Finding The Greedy, Prodigal, and Suicidal Contracts at Scale

Ivica Nikolić
School of Computing, NUS
Singapore

Aashish Kolluri
School of Computing, NUS
Singapore

Ilya Sergey
University College London
United Kingdom

Prateek Saxena
School of Computing, NUS
Singapore

Aquinas Hobor
Yale-NUS College
School of Computing, NUS
Singapore

ABSTRACT

Smart contracts—stateful executable objects hosted on blockchains like Ethereum—carry billions of dollars worth of coins and cannot be updated once deployed. We present a new systematic characterization of a class of *trace vulnerabilities*, which result from analyzing multiple invocations of a contract over its lifetime. We focus attention on three example properties of such trace vulnerabilities: finding contracts that either lock funds indefinitely, leak them carelessly to arbitrary users, or can be killed by anyone. We implemented Maian, the first tool for specifying and reasoning about trace properties, which employs inter-procedural symbolic analysis and concrete validator for exhibiting real exploits. Our analysis of nearly one million contracts flags 34, 200 (2, 365 distinct) contracts vulnerable, in 10 seconds per contract. On a subset of 3, 759 contracts which we sampled for concrete validation and manual analysis, we reproduce real exploits at a true positive rate of 89%, yielding exploits for 3, 686 contracts. Our tool finds exploits for the infamous Parity bug that indirectly locked $200 million US worth in Ether, which previous analyses failed to capture.

1 INTRODUCTION

Cryptocurrencies feature a distributed protocol for a set of computers to agree on the state of a public ledger called the blockchain. The ledgers map accounts or addresses with quantities of virtual coins. Miners, or the computing nodes, facilitate recording the state of a payment network, encoding transactions that transfer coins from one address to another. A significant number of blockchain protocols exist, and as of writing the market value of the associated coins exceeds $300 billion US, creating a lucrative attack target.

Smart contracts extend the idea of a blockchain to a compute platform for decentralized execution of general-purpose applications. Contracts are programs that run on blockchains: their code and state is stored on the ledger, and they can send and receive coins. Smart contracts have been popularized by the Ethereum blockchain. Recently, sophisticated applications of smart contracts have arisen, especially in the area of token management due to the development of the ERC20 token standard. This standard allows the uniform management of custom tokens, enabling, *e.g.*, decentralized exchanges and complex wallets. Today, over a million smart contracts operate on the Ethereum network, and this count is growing.

Smart contracts offer a particularly unique combination of security challenges. Once deployed they cannot be upgraded or patched, unlike traditional consumer device software. Secondly, they are written in a new ecosystem of languages and runtime environments (e.g., for Ethereum, the Ethereum Virtual Machine and its programming language called Solidity). Contracts are relatively difficult to test, especially since their runtimes allow them to interact with other smart contracts and external off-chain services; they can be invoked repeatedly by transactions from a large number of users. Third, since currency and coins on a blockchain often have significant value, attackers are highly incentivized to find and exploit bugs in contracts that process or hold them directly for profit. The attack on the DAO contract cost the Ethereum community $60 million US; and several more recent ones have had impact of a similar scale [1].

In this work, we present a systematic characterization of a class of vulnerabilities that we call *trace* vulnerabilities. Unlike many previous works that have applied static and dynamic analyses to find bugs in contracts automatically [24, 25, 32, 38], our work focuses on detecting vulnerabilities across a sequence of invocations of a contract. We label vulnerable contracts with three categories — greedy, prodigal, and suicidal — which either lock funds indefinitely, leak them to arbitrary users, or be susceptible to by killed by any user. These properties capture many well-known examples of known anecdotal bugs [1, 10, 17], but broadly cover a class of examples that were not known in prior work or public reports. More importantly, our characterization allows us to concretely check for bugs by running the contract, which aids determining confirmed true positives.

We build an analysis tool called Maian for finding these vulnerabilities directly from the bytecode of Ethereum smart contracts, without requiring source code access. In total, across the three categories of vulnerabilities, Maian has been used to analyze 970, 898 contracts live of the public Ethereum blockchain. Our techniques suffice to find the infamous Parity bug that indirectly caused 200 million dollars worth of Ether, which is not found by previous analyses (in part because its detection requires two contract invocations). A total of 34, 200 (2, 365 distinct) contracts are flagged as potentially buggy by Maian. As in the case of the Parity bug, they may put a larger amount to risk, since contracts interact with one another. For 3, 759 contracts we tried to concretely validate, Maian has found over 3, 686 confirmed vulnerabilities with 89% true positive rate. All vulnerabilities are uncovered on average within 10 seconds of analysis per contract.

Contributions. We make the following contributions:

- We identify three classes of *trace* vulnerabilities, which can be captured as properties of execution traces — potentially infinite

sequence of invocations of a contract. Previous techniques and tools [32] are not designed to find these bugs because they only model behavior for a single call to a contract.
- We provide high-order properties to check which admit a mechanized symbolic analysis procedure for detection. We fully implement MAIAN, a tool for symbolic analysis of smart contract bytecode (without access to source code).[1]
- We test close to one million contracts, finding thousands of confirmed true positives within a few seconds of analysis time per contract. Testing trace properties with MAIAN is practical.

2 PROBLEM

We define a new class of trace vulnerabilities, showing three specific examples of properties that can be checked in this broader class. We present our approach and tool to reason about the class of trace vulnerabilities.

2.1 Ethereum Smart Contracts

In the Ethereum blockchain, smart contracts are a type of accounts that hold executable code called a bytecode. A contract performs actions according to the instructions specified by its bytecode. Such an action, called a contract invocation, occurs when an Ethereum account sends a transaction (that contains input data) to the contract. Therefore, a single transaction to a contract triggers one execution of its bytecode according to the provided input data. Smart contract can also be invoked by other contract with a message call, which is implemented as a bytecode instruction. Contracts can be executed repeatedly over their lifetime. An *execution trace* is a sequence of consecutive contract invocations.

Ethereum accounts (also known as addresses), both smart contracts and normal accounts (called externally owned), hold some amount of Ether, which is the currency of Ethereum. Smart contract may receive Ether from other accounts when invoked, and can send their Ether to other accounts with message calls. Contracts can also be removed from the blockchain. This is called killing a contract and results in completely erasing contract's logic from the blockchain and sending its Ether to a predetermined address.

All the actions a contract takes, including sending Ether and getting killed, occur only when specific bytecode instructions are executed during its invocation. The Ethereum Virtual Machine (EVM) is the engine that interprets and executes the bytecode of smart contracts when invoked. The bytecode instructions are low-level and often too complex to be used for directly programming common logic. Thus, most of smart contracts are written in Solidity, a high-level programming language for Ethereum smart contracts, and later compiled to bytecode and deployed on the blockchain. In Solidity, a contract can send out its Ether with operations such as `send`, `call`, `transfer`, while it can be killed with `suicide`, `selfdestruct`.

For clarity reasons, in the paper we provide examples of smart contracts in Solidity. However, we note that all of our analysis applies to smart contract specified directly in bytecode.

[1] Link to the implementation omitted for the double-blind submission.

```
function payout(address[] recipients,
                uint256[] amounts) {
 require(recipients.length==amounts.length);
 for (uint i = 0; i < recipients.length; i++) {
    /* ... */
    recipients[i].send(amounts[i]);
  }}
```

Figure 1: `Bounty` contract; `payout` leaks Ether.

2.2 Contracts with Trace Vulnerabilities

While trace vulnerabilities are a broader class, we focus our attention on three example properties to check of contract traces. Specifically, we flag contracts which (a) release Ether to arbitrary addresses carelessly, (b) can be killed by arbitrary addresses, and (c) have no way to release Ether after a certain execution state.

Note that any characterization of bugs must be taken with a grain of salt, since one can always argue that the exposed behavior embodies intent — as was debated in the case of the DAO bug [10]. Our characterization of vulnerabilities is based, in part, on anecdotal incidents reported publicly [2, 10, 17]. To the best of our knowledge, however, our characterization is the first to precisely define checkable properties of such incidents and measure their prevalence. There are several valid reasons for contracts for being killable or giving them out to addresses not known at the time of deployment. For instance, benign contracts such as bounties or games, often hold funds for long periods of time (until a bounty is awarded) and release them to addresses that are not known statically. Our characterization admits these benign behaviors and flags egregious violations described next, for which we are unable to find justifiable intent.

Prodigal Contracts. Contracts often return funds to owners (accounts that deployed them), to addresses that have sent Ether to them in past (*e.g.*, in lotteries), or to addresses that exhibit a specific solution (*e.g.*, in bounties). However, when a contract gives away Ether to an arbitrary address, we deem this as a vulnerability. We are interested in finding such contracts, which we call *prodigal*.

Consider the `Bounty` contract with code fragment given in Figure 1. This contract collects Ether from different sources and rewards bounty to a selected set of recipients. The function `payout` sends to a list of recipients specified amounts of Ether. From its definition, it is clear that the recipients and the amounts are specified by the inputs, and anybody can call the function (*i.e.*, the function does not have a restriction on the sender). Therefore, any user can invoke this function, and send all of contract's Ether to addresses of her choice.

The above contract requires a single function invocation to leak its Ether. However, there are examples of contracts which need two or more invocations (calls with specific arguments) to cause a leak. Such examples are presented in Section 5.

Suicidal Contracts. A contract often enables a security fallback option of being killed by its owner (or trusted addresses) in emergency situations like when being drained of its Ether due to attacks, or when malfunctioning. However, if a contract can be killed by *any* arbitrary account, we consider it vulnerable and call it *suicidal*.

The recent *Parity* fiasco[1] is a concrete example of such type of a contract. A supposedly innocent Ethereum account [33] killed

```
1  function initMultiowned(address[] _owners,
2                          uint _required){
3    if (m_numOwners > 0) throw;
4    m_numOwners = _owners.length + 1;
5    m_owners[1] = uint(msg.sender);
6    m_ownerIndex[uint(msg.sender)] = 1;
7    m_required = _required;
8    /* ... */
9  }
10
11 function kill(address _to) {
12     uint ownerIndex = m_ownerIndex[uint(msg.sender)];
13     if (ownerIndex == 0) return;
14     var pending = m_pending[sha3(msg.data)];
15     if (pending.yetNeeded == 0) {
16       pending.yetNeeded = m_required;
17       pending.ownersDone = 0;
18     }
19     uint ownerIndexBit = 2**ownerIndex;
20     if (pending.ownersDone & ownerIndexBit == 0) {
21       if (pending.yetNeeded <= 1)
22         suicide(_to);
23       else {
24         pending.yetNeeded--;
25         pending.ownersDone |= ownerIndexBit;
26       }
27     }
28 }
```

Figure 2: Simplified fragment of `ParityWalletLibrary` contract, which can be killed.

a library contract on which the main *Parity* contract relies, thus rendering the latter non-functional and locking all its Ether. To understand the *suicidal* side of the library contract, focus on its shortened code fragment given in Figure 2. To kill the contract, an arbitrary account invokes two different contract functions: one to set the ownership,[2] and one to actually kill it. That is, the account first calls `initMultiowned`, providing empty array for `_owners`, and zero for `_required`. (This effectively means that the contract has no owners and that nobody has to agree to execute a specific contract function.) Then the account invokes the contract function `kill`. This function needs `_required` number of owners to agree to kill the contract, before the actual **`suicide`** command at line 22 is executed. However, since in the previous call to `initMultiowned`, the value of `_required` was set to zero, **`suicide`** is executed, and thus the contract is killed.

Greedy Contracts. We refer to contracts that remain alive and lock Ether indefinitely, allowing it be released under no conditions, as *greedy*. In the example of the `Parity` contract, many other `multisigWallet`-like contracts which held Ether, used functions from the `Parity` library contract to release funds to their users. After the `Parity` library contracts was killed, the wallet contracts could no longer access the library, thus became greedy. This vulnerability resulted in locking of $200M US worth of Ether indefinitely!

Greedy contracts can arise out of more direct errors as well. The most common such errors occur in contracts that accept Ether but either completely lack instructions that send Ether out (e.g.

[2]The bug would have been prevented has the function `initMultiowned` been properly initialized by the authors.

```
1  contract AddressReg{
2    address public owner;
3    mapping (address=>bool) isVerifiedMap;
4    function setOwner(address _owner){
5      if (msg.sender==owner)
6          owner = _owner;
7    }
8    function AddressReg(){ owner = msg.sender; }
9    function verify(address addr){
10     if (msg.sender==owner)
11       isVerifiedMap[addr] = true;
12   }
13   function deverify(address addr){
14     if (msg.sender==owner)
15       isVerifiedMap[addr] = false;
16     }
17   function hasPhysicalAddress(address addr)
18             constant returns(bool){
19      return isVerifiedMap[addr];
20   }
21 }
```

Figure 3: `AddressReg` contract locks Ether.

bytecode instructions corresponding to **`send`**, **`call`**, **`transfer`**), or such instructions are not reachable. An example of contract that lacks instructions that release Ether, that has already locked Ether is given in Figure 3.

Posthumous Contracts. When a contract is killed, its code and global variables are cleared from the blockchain, thus preventing any further execution of its code. However, all killed contracts continue to receive transactions. Although such transactions can no longer invoke the code of the contract, if Ether is sent along them, it is added to the contract balance, and similarly to the above case, it is locked indefinitely. Killed contract or contracts that do not contain any code, but have non-zero Ether we call *posthumous*. It is the onus of the sender to check if the contract is alive before sending Ether, and evidence shows that this is not always the case. Because posthumous contracts require no further static analysis beyond that for identifying suicidal contracts, we do not treat this as a separate class of bugs. We merely list all posthumous contracts on the live Ethereum blockchain we have found in Section 5.

2.3 Our Approach

Each run of the contract, called an invocation, may exercise an execution path in the contract code under a given input context. Note that prior works have considered bugs that are properties of *one* invocation, ignoring the chain of effects across a *trace* of invocations [7, 25, 26, 29, 30, 38]. We develop a tool that uses systematic techniques to find contracts that violate specific properties of traces. The violations are either:

(a) of *safety* properties, asserting that there *exists* a trace from a specified blockchain state that causes the contract to violate certain conditions; and
(b) of *liveness* properties, asserting whether some actions *cannot* be taken in *any* execution starting from a specified blockchain state.

We formulate the three kinds of vulnerable contracts as these safety and liveness trace properties in Section 3. Our technique of finding

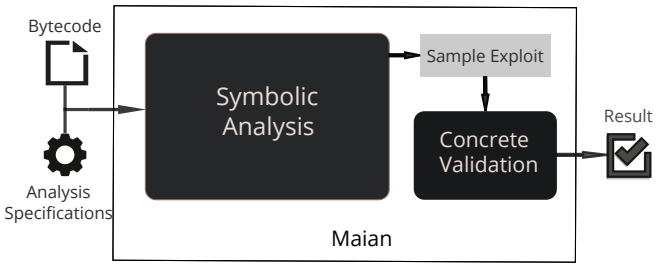

Figure 4: Maian

vulnerabilities, implemented as a tool called Maian and described in Section 4, consists of two major components: symbolic analysis and concrete validation. The symbolic analysis component takes contract bytecode and analysis specifications as inputs. The specifications include vulnerability category to search for and depth of the search space, which further we refer to as *invocation depth*, along with a few other analysis parameters we outline in Section 4. To develop our symbolic analysis component, we implement a custom Ethereum Virtual Machine, which facilitates symbolic execution of contract bytecode [32]. With every contract candidate, our component runs possible execution traces symbolically, until it finds a trace which satisfies a set of predetermined properties. The input context to every execution trace is a set of symbolic variables. Once a contract is flagged, the component returns concrete values for these variables. Our final step is to run the contract concretely and validate the result for true positives; this step is implemented by our concrete validation component. The component takes the inputs generated by symbolic analysis component and checks the exploit of the contract on a private fork of Ethereum blockchain. Essentially, it is a testbed environment used to confirm the correctness of the bugs. As a result, at the end of validation the candidate contract is determined as true or false positive, but the contract state on main blockchain is not affected since no changes are committed to the official Ethereum blockchain.

3 TRACE VULNERABILITIES

We consider three types of bugs in smart contracts which are exploitable via execution traces and which belong to two standard categories. The first category regards a contract *buggy* with respect to a certain class of unwelcome high-level scenarios (*e.g.*, "leaking" funds) if some of its finite execution traces fail to satisfy a certain condition. Trace properties characterised this way are traditionally regarded as *safety*, meaning that "during the execution nothing bad happens". The second category is related to contracts where proving the absence of some other high-level bugs requires establishing a statement of a different kind, namely, "something good must eventually happen". Such properties are known as *liveness* and require reasoning about progress in executions.

In this section, we introduce the execution model of Ethereum smart contracts and define the three types of bugs.

3.1 EVM Semantics and Traces

In Ethereum, a smart contract is identified by its 160-bit address. For each contract, the blockchain stores its three distinguished fields: *balance* represents the amount of Ether in possession, *code* specifies the program logic of the contract in bytecode, and *storage* is allocated to save global variables of the program.

The *code* field is immutable[3] – once a contract is deployed on the blockchain its logic cannot be updated. Its bytecode is run on Ethereum Virtual Machine (EVM), a stack-based execution runtime [42]. Different source languages compile to the EVM semantics, the predominant of them being Solidity [41]. A run of the code, *i.e.*, invocation of the smart contract, is triggered by initiating a transaction (a *call*) with a message to a contract, referred to via its address, so the message's payload includes input arguments for the contract's call and a fee (known as *gas*) [42]. The mining network executes replicated instances of the contract code and agrees on the outputs of the invocation via the standard blockchain consensus protocol, *i.e.*, Nakamoto consensus [31, 35]. The result of the computation is replicated via the blockchain and grants a transaction fee to the miners as per block reward rates established periodically. Contracts can invoke other contracts via message calls usually implemented as the bytecode instruction `CALL`; outputs of these calls, considered to be a part of the same transaction, are returned to the caller during the runtime. The invoked contracts can find their `CALLER`, *i.e.*, they have access to the account (contract) that sends the transaction (message call), and `CALLVALUE`, *i.e.*, the amount of sent Ether.

During the execution of the bytecode, the EVM may change the contract *storage*, *i.e.*, the values of the global variables used in the bytecode. If the execution successfully completes, the updated *storage* is written to the blockchain. Thus the field *storage* is mutable; its value can change according to properly executed bytecode instructions. The execution of a bytecode is proper, if it reaches the instructions `STOP` or `RETURN`. On the other hand, the execution may "throw" if it reaches a non-existing instruction code or invalid jump destination. In such a case, it terminates and all the global updates are reverted.

The *balance* of a contract can be read by anyone, but it is updated via calls to and from other contracts (*i.e.*, by executing the `CALL` instruction) or via transactions send to the contract. Contracts live perpetually unless they are explicitly terminated (or killed) by executing the `SUICIDE` bytecode instruction, which clears their *storage* and *code* fields from the blockchain, and sends their *balance* to an account specified as a parameter of the instruction.

When alive, contracts can be invoked many times. Further we consider contract invocations via transactions, *i.e.*, an externally owned account sends a transaction (with possibly non-zero Ether amount) to the contract address. The transaction contains some data which is passed as an input to the the contract's *code*. When such a transaction is mined, it gets executed by the EVM. This engine takes the contract's *code*, the provided input data, as well as the *storage* of the contract. It executes the *code* and, if properly, writes the updated values of the contract's *storage*, possibly clears its *code* field if the contract is killed, and it updates all *balance* fields

[3]The *code* field may change only if a contract is killed – see further.

(of all accounts to which the contract sent Ether according to the executed instructions) on the blockchain.

It is critical to understand that a contract invocation depends as well on the *storage* of the contract. Hence, we reason about the security of contracts not only depending on their *code*, but on their value of the blockchain. Thus we talk about blockchain state $\sigma(C)$ of a contract C, *i.e.*, the value of its three fields on the blockchain.

Finally, instead of focusing on a single invocation, we can talk about a *trace* of invocations, *i.e.*, a consecutive sequence of transactions, invoking calls to a contract from the same Ethereum account (*caller*). An *invocation depth* of a trace is the number of transactions in the trace. Below, we focus on the traces whose all transactions are from the same caller, and are mined one after another and that there are no other transactions (from other callers) mined in between. Zero-Ether traces are composed of transactions that do not send any Ether.

3.2 Safety Violations

Given the notion of contract traces we can define the first two types of vulnerable smart contracts, namely *prodigal* and *suicidal*. The two bugs are due to safety violations, *i.e.*, execution of specially constructed traces reach bytecode instructions that violate certain properties expected from secure smart contracts.

Definition 3.1 (Prodigal contracts). A contract C at blockchain state $\sigma(C)$ is called *prodigal* if an *arbitrary* account A can send a zero-Ether trace to C, which when executed results in transfer of Ether from the C to A.

In short, prodigal contracts, without receiving, send Ether to an arbitrary account. (Note, we simulate an arbitrary account by assuming its address is any fixed 160-bit string A). To detect if a contract is prodigal, we try to build an execution trace in which all of the transaction have `CALLVALUE = 0` and the last transaction triggers one of the bytecode instructions that transfer Ether to A. More specifically, we assume that in all of the transactions the execution of the last transaction should either:

- reach the `CALL` instruction with recipient being the transaction `CALLER` and the transfer amount non-zero, and afterwards reach a normal stopping instruction such as `STOP` or `RETURN`. This assures that the contract sends some Ether to A and afterwards does not throw (otherwise, the whole transaction is ignored and the Ether transfer is reverted); or
- reach the `SUICIDE` instruction with recipient being the `CALLER`. Such instruction will immediately kill the contract and transfer all of its funds to A

Definition 3.2 (Suicidal contracts). A contract C at blockchain state $\sigma(C)$ is called *suicidal* if an arbitrary account can send a trace to C, which when executed, kills the contract.

The definitions of suicidal and prodigal contracts are similar, and so are their detection techniques. To check if a contract is suicidal, we try to build a trace where the last transaction has to only reach the `SUICIDE` instruction in the bytecode.

3.3 Liveness Violations

A contract at a certain blockchain state is considered locking, if no execution trace will trigger release of its Ether. Since disproving liveness properties of this kind with a finite counterexample is impossible in general, we formulate our definition as an *under-approximation* of the property of interest, considering only traces up to a certain depth:

Definition 3.3 (Greedy contracts). A contract C at blockchain state $\sigma(C)$ with a non-zero balance is called *k-greedy* if execution of any trace with invocation depth k for C and sent by any account, does not result in transfer of Ether from C (to any account).

Interestingly, the definition of a greedy contract is dual to the notion of a prodigal, that is, the contract will not release its Ether regardless of the sender of the transactions. To detect greedy contracts we show that executions of all traces with up to k invocations do not reach the instructions that transfer Ether such as `CALL`.

4 THE ALGORITHM AND THE TOOL

Maian is a symbolic analyzer for smart contract execution traces, for the properties defined in Section 3. It takes as input a contract in its bytecode form and contract's state at concrete block value from the Ethereum blockchain, flagging contracts with bugs outlined in Section 2.2. Depending on the category of bugs, Maian either tries to build or shows an absence of particular type of traces according to conditions from Section 3. To reason about traces, the tool executes them symbolically. For the sake of tractability of the analysis, it does not keep track of the entire blockchain context σ (including the state of other contracts), treating only the contract's transaction inputs and certain block parameters as symbolic. To reduce the number of false positives and confirm concrete exploits for vulnerabilities, Maian calls its *concrete validation* routine, which we outline in Section 4.2.

4.1 Symbolic Analysis

Our work concerns finding properties of traces that involve multiple invocations of a contract. We leverage static symbolic analysis to perform this step in a way that allows reasoning across contract calls and across multiple blocks. We start our analysis given a contract bytecode and a starting concrete context capturing values of the blockchain. Maian reasons about values read from input transaction fields and block parameters[4] in a symbolic way—specifically, it denotes the set of all concrete values that the input variable can take as a symbolic variable. It then symbolically interprets the relationship of other variables computed in the contract as a symbolic expression over symbolic variables. For instance, the code `y := x + 4` results in a symbolic value for `y` if `x` is a symbolic expression; otherwise it is executed as concrete value. Conceptually, one can imagine the analysis as maintaining two memories mapping variables to values: one is a symbolic memory mapping variables to their symbolic expressions, the other mapping variables to their concrete values.

Execution Path Search. The symbolic interpretation searches the space of all execution paths in a trace with a depth-first search. The search is a best effort to increase coverage and find property violating traces. Our goal is neither to be sound, *i.e.*, search all possible paths at the expense of false positives, nor to be provably complete,

[4]Those being `CALLVALUE`, `CALLER`, `NUMBER`, `TIMESTAMP`, `BLOCKHASH`, `BALANCE`, `ADDRESS`, and `ORIGIN`.

i.e., have only true positives at the expense of coverage [16]. From a practical perspective, we make design choices that strike a balance between these two goals.

The symbolic execution starts from the entry point of the contract, and considers all functions which can be invoked externally as an entry point. More precisely, the symbolic execution starts at the first instruction in the bytecode, proceeding sequentially until the execution path ends in terminating instruction. Such instruction can be valid (*e.g.*, `STOP, RETURN`), in which case it assumes to have reached the end of the contract invocations, and thus restart the symbolic execution again from the first bytecode instruction to simulate the next invocation. On the other hand, the terminating instruction can be invalid (*e.g.*, non-existing instruction code or invalid jump destination), in which case it terminates the search down this path and backtracks in the depth-first search procedure to try another path. When execution reaches a branch, Maian concretely evaluates the branch condition if all the variables used in the conditional expression are concrete. This uniquely determines the direction for continuing the symbolic execution. If the condition involves a symbolic expression, Maian queries an external SMT solver to check for the satisfiability of the symbolic conditional expression as well as its negation. Here, if the symbolic conditional expression as well as its negation are satisfiable, both branches are visited in the depth-first search; otherwise, only the satisfiable branch is explored in the depth first search. On occasions, the satisfiability of the expression cannot be decided in a pre-defined timeout used by our tool; in such case, we terminate the search down this path and backtrack in the depth-first search procedure to try another path. We maintain a symbolic path constraint which captures the conditions necessary to execute the path being analyzed in a standard way. Maian implements support for 121 out of the 133 bytecode instructions in Ethereum's stack-based low-level language. More precisely, it supports all but the instructions that cannot be realized with the symbolic execution engine (for instance, the `CREATE` instruction, which deploys a new contract). When the tool encounters an unsupported instruction, it proceeds on a best effort basis by stopping exploration of that branch but continues to explore other contract branches via backtracking.

At a call instruction, control follows transfer to the target. If the target of the transfer is a symbolic expression, Maian backtracks in its depth-first search. Calls outside a contract, however, are not simulated and returns are marked symbolic. Therefore, Maian depth-first search is inter-procedural, but not inter-contract.

Handling data accesses. The memory mappings, both symbolic and concrete, record all the contract memory as well blockchain storage. During the symbolic interpretation, when a global or blockchain storage is accessed for the first time on a path, its concrete value is read from the main Ethereum blockchain into local mappings. This ensures that subsequent reads or writes are kept local to the path being presently explored.

The EVM machine supports a flat byte-addressable memory, and each address has a bit-width of 256 bits. The accesses are in 32-byte sized words which Maian encodes as bit-vector constraints to the SMT solver. Due to unavailability of source code, Maian does not have any prior information about higher-level datatypes in the memory. All types default to 256-bit integers in the encoding used by Maian. Furthermore, Maian attempts to recover more advanced types such as dynamic arrays by using the following heuristic: if a symbolic variable, say x, is used in constant arithmetic to create an expression (say $x + 4$) that loads from memory (as an argument to the `CALLDATALOAD` instruction), then it detects such an access as a dynamic memory array access. Here, Maian uses the SMT solver to generate k concrete values for the symbolic expression, making the optimistic assumption that the size of the array to be an integer in the range $[0, k]$. The parameter k is configurable, and defaults to 2. Apart from this case, whenever accesses in the memory involve a symbolic address, Maian does not do alias analysis and simply terminates the explored path, backtracking in its depth-first search.

Handling non-deterministic inputs. Contracts have several sources of non-deterministic inputs such as the block timestamp, *etc.* While these are treated as symbolic, they are not exactly under the control of the external users. Maian does not use their concrete values because it still needs to reason about invocations of the contract across multiple invocations, *i.e.*, at different blocks.

Flagging Violations. When the depth-first search in the space of the contract execution reaches a state where the desired safety property is violated, it flags the contract as a buggy *candidate*. The symbolic path constraint, along with the necessary property conditions, are asserted for satisfiability to the SMT solver. We use Z3 [9] as our solver, which provides concrete values so satisfy an input formula. We use these values as the concrete data for symbolic inputs, including the symbolic transaction data.

Bounding the path search space. Maian takes the following steps to bound the search in the (potentially infinite) path space. First, the call depth is limited to the constant called `max_call_depth`, which defaults to 3 but can be configured for empirical tests. Second, we limit the total number of jumps or control transfers on one path explored to a configurable constant `max_cfg_nodes`, default set to 60. This is necessary to avoid being stuck in loops, for instance. Third, we set a `timeout` of 10 seconds per call to our SMT solver. Lastly, the total time spent on a contract is limited to configurable constant `max_analysis_time`, default set to 300 seconds.

Pruning. To speed up the state search, we implement pruning with memorization. Whenever the search encounters that the particular configuration (*i.e.*, contract storage, memory, and stack) has been seen before, it does not further explore that part of the path space.

4.2 Concrete Validation

In the concrete validation step, Maian creates a private fork of the original Ethereum blockchain with the last block as the input context. It then runs the contract with the concrete values of the transactions generated by the symbolic analysis to check if the property holds in the concrete execution. If the concrete execution fails to exhibit a violation of the trace property, we mark the contract as a *false positive*; otherwise, the contract is marked as a *true positive*. To implement the validating framework, we added a new functionality to the official go-ethereum package [15] which allows us to fork the Ethereum main chain at a block height of our choice. Once we fork the main chain, we mine on that fork without connecting to any peers on the Ethereum network, and thus we are able to mine our own transactions without committing them to the main chain.

Prodigal Contracts. The validation framework checks if a contract indeed leaks Ether by sending to it the transactions with inputs provided by the symbolic analysis engine. The transactions are sent by one of our accounts created previously. Once the transactions are executed, the validation framework checks whether the contract has sent Ether to our account. If a verifying contract does not have Ether, our framework first sends Ether to the contract and only then runs the exploit.

Suicidal Contracts. In a similar fashion, the framework checks if a contract can be killed after executing the transactions provided by the symbolic analysis engine on the forked chain. Note, once a contract is killed, its bytecode is reset to `'0x'`. Our framework uses precisely this test to confirm the correctness of the exploit.

Greedy Contracts. A strategy similar to the above two cannot be used to validate the exploits on contracts that lock Ether. However, during the bug finding process, our symbolic execution engine checks firsthand whether a contract accepts Ether. The validation framework can, thus, check if a contract is true positive by confirming that it accepts Ether and does not have `CALL`, `CALLCODE`, `DELEGATECALL`, or `SUICIDE` opcodes in its bytecode. In Section 5 we give examples of such contracts.

5 EVALUATION

We analyzed 970, 898 smart contracts, obtained by downloading the Ethereum blockchain from the first block until block number 4, 800, 000. Ethereum blockchain has only contract bytecodes. To obtain the original (Solidity) source codes, we refer to the Etherscan service [13] and obtain source for 9, 825 contracts. Only around 1% of the contracts have source code, highlighting the utility of Maian as a bytecode analyzer.

Recall that our concrete validation component can analyze a contract from a particular block height where the contract is alive (*i.e.*, initialized, but not killed). To simplify the validation process for a large number of contracts flagged by the symbolic analysis component, we perform our concrete validation at block height of 4, 499, 451, further denoted as `BH`. At this block height, we find that most of the flagged contracts are alive, including the Parity library contract [1] that our tool successfully finds. This contract was killed at a block height of 4, 501, 969. All contracts existing on blockchain at a block height of 4, 499, 451 are tested, but only contracts that are alive at `BH` are concretely validated.[5]

Experimental Setup and Performance. Maian supports parallel analysis of contracts, and scales linearly in the number of available cores. We run it on a Linux box, with 64-bit Ubuntu 16.04.3 LTS, 64GB RAM and 40 CPUs Intel(R) Xeon(R) E5-2680 v2@2.80GHz. In most of our experiments we run the tool on 32 cores. On average, Maian requires around 10.0 seconds to analyze a contract for the three aforementioned bugs: 5.5 seconds to check if a contract is prodigal, 3.2 seconds for suicidal, and 1.3 seconds for greedy.

5.1 Results

Table 1 summarizes the contracts flagged by Maian. Given the large number of flagged contracts, we select a random subset for concrete validation, and report on the true positive rates obtained. We report the number of distinct contracts, calculated by comparing the hash of the bytecode; however, all percentages are calculated on the original number of contracts (with duplicates).

Category	**#Candidates flagged** (*distinct*)	Candidates without source	#Validated	% of true positives
Prodigal	**1504** (*438*)	1487	1253	*97*
Suicidal	**1495** (*403*)	1487	1423	*99*
Greedy	**31,201** (*1524*)	31,045	1083	*69*
Total	34, 200 (2, 365)	34, 019	3, 759	89

Table 1: Results for invocation depth 3 at block height `BH`. Column 1 reports number of flagged contracts, and the distinct ones among these. Column 2 shows the number of flagged without source code. Column 3 is the subset we sampled for concrete validation. Column 4 reports true positive rates; the total here is the average TP rate weighted by the number of validated contracts.

Prodigal contracts. Our tool has flagged 1, 504 candidates contracts (438 distinct) which may leak Ether to an arbitrary Ethereum address, with a true positive rate of around 97%. At block height `BH`, 46 of these contracts hold some Ether. The concrete validation described in Section 4.2 succeeds for exploits for 37 out of 46 — these are true positives, whereas 7 are false positives. The remaining 2 contracts leak Ether to an address different from the caller's address. Note that all of the 37 true positive contracts are alive as of this writing. For ethical reasons, no exploits were done on the main blockchain.

Of the remaining 1, 458 contracts which presently do not have Ether on the public Ethereum blockchain, 24 have been killed and 42 have not been published (as of block height `BH`). To validate the remaining alive contracts (in total 1392) on a private fork, first we send them Ether from our mining account, and find that 1, 183 contracts can receive Ether.[6] We then concretely validate whether these contract leak Ether to an arbitrary address. A total of 1, 156 out of 1, 183 (97.72%) contracts are confirmed to be true positives; 27 (2.28%) are false positives.

For each of the 24 contracts killed by the block height `BH`, the concrete validation proceeds as follows. We create a private test fork of the blockchain, starting from a snapshot at a block height where the contract is alive. We send Ether to the contract from one of our addresses address, and check if the contract leaks Ether to an arbitrary address. We repeat this procedure for each contract, and find that all 24 candidate contracts are true positives.

Suicidal contracts. Maian flags 1, 495 contracts (403 distinct), including the `ParityWalletLibrary` contract, as found susceptible to being killed by an arbitrary address, with a nearly 99% true positive rate. Out of 1, 495 contracts, 1, 398 are alive at `BH`. Our concrete validation engine on a private fork of Ethereum confirm that 1, 385 contracts (or 99.07%) are true positives, *i.e.*, they can be killed by any arbitrary Ethereum account, while 13 contracts (or 0.93%) are false positives. The list of true positives includes the recent `ParityWalletLibrary` contract which was killed at block height 4, 501, 969 by an arbitrary account. Of the 1, 495 contracts

[5] We also concretely validate flagged candidates that were killed before `BH` as well.

[6] These are live and we could update them with funds in testing.

```
1 bytes20 prev;
2 function tap(bytes20 nickname) {
3     prev = nickname;
4     if (prev != nickname) {
5       msg.sender.send(this.balance);
6     }
7 }
```

Figure 5: A prodigal contract.

flagged, 25 have been killed by BH; we repeat the procedure described previously and confirmed all of them as true positives.

Greedy contracts. Our tool flags 31,201 greedy candidates (1,524 distinct), which amounts to around 3.2% of the contracts present on the blockchain. The first observation is that MAIAN deems all but these as accepting Ether but having states that release them (not locking indefinitely). To validate a candidate contract as a true positive one has to show that the contract does not release/send Ether to any address for any valid trace. However, concrete validation may not cover all possible traces, and thus it cannot be used to confirm if a contract is greedy. Therefore, we take a different strategy and divide them into two categories:

(i) Contracts that accept Ether, but in their bytecode do not have any of the instructions that release Ether (such instructions include CALL, CALLCODE, SUICIDE, or DELEGATECALL).

(ii) Contracts that accept Ether, and in their bytecode have at least one of CALL, CALLCODE, SUICIDE or DELEGATECALL.

MAIAN flagged 1,058 distinct contracts from the first category. We validate that these contracts can receive Ether (we send Ether to them in a transaction with input data according to the one provided by the symbolic execution routine). Our experiments show that 1,057 out of 1,058 (*e.g.*, 99.9%) can receive Ether and thus are true positives. On the other hand, the tool flagged 466 distinct contracts from the second category, which are harder to confirm by testing alone. We resort to manual analysis for a subset of these which have source code. Among these, only 25 have Solidity source code. With manual inspection we find that none of them are true positive — some traces can reach the CALL code, but MAIAN failed to reach it in its path exploration. The reasons for these are mentioned in the Section 5.3. By extrapolation (weighted average across 1,083 validated), we obtain true positive rate among greedy contracts of approximately 69%.

Posthumous Contracts. Recall that posthumous are contracts that are dead on the blockchain (have been killed) but still have non-zero Ether balance. We can find such contracts by querying the blockchain, *i.e.*, by collecting all contracts without executable code, but with non-zero balance. We found 853 contracts at a block height of 4,800,000 that do not have any compiled code on the blockchain but have positive Ether balance. Interestingly, among these, 294 contracts have received Ether after they became dead.

5.2 Case Studies: True Positives

Apart from examples presented in section 2.2, we now present true and false positive cases studies. Note that we only present the contracts with source code for readability. However, the fraction of flagged contracts with source codes is very low (1%).

Prodigal contracts. In Figure 5, we give an example of a prodigal contract. Note that the function tap seems to lock Ether since the

```
1  contract Mortal {
2    address public owner;
3    function mortal() {
4      owner = msg.sender;
5    }
6    function kill() {
7      if (msg.sender == owner){
8        suicide(owner);
9      }
10   }
11 }
12 contract Thing is Mortal { /*...*/ }
```

Figure 6: The prodigal contract `Thing`, derived from `Mortal`, leaks Ether to any address by getting killed.

```
1  function withdraw() public returns (uint) {
2    Record storage rec = records[msg.sender];
3    uint balance = rec.balance;
4    if (balance > 0) {
5      rec.balance = 0;
6      msg.sender.transfer(balance);
7      Withdrawn(now, msg.sender, balance);
8    }
9    if (now - lastInvestmentTime > 4 weeks) {
10     selfdestruct(funder);
11   }
12   return balance; }
```

Figure 7: The `Dividend` contract can be killed by invoking `withdraw` if the last investment has been made at least 4 weeks ago.

condition in line 4, semantically, can never be true. However, the compiler optimization of Solidity allows this condition to pass when an input greater than 20 bytes is used to call the function tap. The EVM always loads 32 bytes from the input data and decodes it according to the type of argument. In this case, the first 20 bytes of nickname are assigned to the global variable prev, thus neglecting the remaining 12 bytes. The error occurs because EVM at line 4, correctly nullifies the 12 bytes in prev, but not in nickname. Thus if nickname has non-zero values in these 12 bytes then the inequality is true. This contract so far has lost 5.0001 Ether to different addresses on real Ethereum blockchain.

A contract may also leak Ether by getting killed since the semantic of SUICIDE instruction enforce it to send all of its balance to an address provided to the instruction. In Figure 6, the contract Thing[28] is inherited from a base contract Mortal. This contract implements a review system in which public reviews an ongoing topic. Among others, it has a kill function inherited from its base contract which is used to send its balance to its owner if its killed. The function mortal, supposedly a constructor, is misspelled, and thus anyone can call mortal to become the owner of the contract. Since the derived contract Thing inherits functions from contract Mortal, this vulnerability in the base contract allows an arbitrary Ethereum account to become the owner of the derived contract, to kill it, and to receive its Ether. Hence, a trace composed of two functions calls, to mortal and to kill, makes the contract prodigal.

```
contract SimpleStorage {
  uint storedData; address storedAddress;
  event flag(uint val, address addr);

  function set(uint x, address y) {
    storedData = x; storedAddress = y;
  }
  function get() constant
    returns(uint retVal, address retAddr) {
    return (storedData,storedAddress);
  }
}
```

Figure 8: A contract that locks Ether.

Suicidal contracts. A contract can be killed by exploiting an unprotected SUICIDE instruction. A trivial example is a public kill function which hosts this instruction. Sometimes, SUICIDE is protected by a weak condition, such as in the contract Dividend given in Figure 7. This contract allows users to buy shares or withdraw their investment. The logic of withdrawing investment is implemented by the withdraw function. However, this function has a self_destruct command which can be executed once the last investment has been made more than 4 weeks ago. Hence, if an investor calls this function after 4 weeks of the last investment, all the funds go to the owner of the contract and all the records of investors are cleared from the blockchain. Though the Ether is safe with the owner, there would be no record of any investment for the owner to return ether to investors.

In the previous example, one invocation of withdraw function was sufficient to kill the contract. There are, however, contracts which require two or more function invocations to be killed, i.e. they require a trace to be killed. For example, the mentioned Parity bug requires a trace composed of two function invocations [1]. It is interesting to note that the bug produced by Maian is slightly different from the actual bug used to kill the Parity library contract[7].

Greedy contracts. The contract SimpleStorage, given in Figure 8, is an example of a contract that locks Ether indefinitely. When an arbitrary address sends Ether along with a transaction invoking the set function, the contract balance increases by the amount of Ether sent. However, the contract does not have any instruction to release Ether, and thus locks it on the blockchain.

The **payable** keyword has been introduced in Solidity recently to prevent functions from accepting Ether by default, *i.e.*, a function not associated with **payable** keyword throws if Ether is sent in a transaction. However, although this contract does not have any function associated with the **payable** keyword, it accepts Ether since it had been compiled with an older version of Solidity compiler (with no support for **payable**).

5.3 Case Studies: False Positives

We manually analyze cases where Maian's concrete validation fails to trigger the necessary violation with the produced concrete values, if source code is available.

[7] Maian produced simpler inputs to the functions: instead of using one so-called multi-owner address for approving actions as in the original bug, Maian used zero multi-owners.

```
function confirmTransaction(uint tId)
  ownerExists(msg.sender) {
  confirmations[tId][msg.sender] = true;
  executeTransaction(tId);
}
function executeTransaction(uint tId) {
// In case of majority
  if (isConfirmed(tId)) {
    Transaction tx = transactions[tId];
    tx.executed = true;
    if (tx.destination.call.value(tx.value) (tx.data))
      /*...*/
  }}
```

Figure 9: False positive, flagged as a greedy contract.

```
function RandomNumber() returns(uint) {
  /*...*/
  last =
    seed^(uint(sha3(block.blockhash(block.number),
     nonces[msg.sender])) * 0x000b00070005000300001);
}
function Guess(uint _guess) returns (bool) {
  if (RandomNumber() == _guess) {
    if (!msg.sender.send(this.balance)) throw;
    /*...*/
  }
  /*...*/
}
```

Figure 10: False positive, flagged as a prodigal contract.

Prodigal and Suicidal contracts. In both of the classes, false positives arise due to two reasons:

(i) The tool performs inter-procedural analysis within a contract, but does not transfer control in cross-contract calls. For calls from one contract to a function of another contract, Maian assigns symbolic variables to the return values. This is imprecise, because real executions may only return a single value (say true) when the call succeeds.

(ii) Maian may assign values to symbolic variables related to block state (*e.g.*, blocknumber) in cases where these values are used to decide the control flow. Thus, we may get false positives because those values may be different at the concrete validation stage. For instance, in Figure 10, the _guess value depends on the values of block parameters, which cannot be forced to take on the concrete values found by our analyzer.

Greedy contracts. The large share of false positives is attributed to two causes:

(i) Detecting a trace which leads to release of Ether may need three or more function invocations. For instance, in Figure 9, the function confirmTransaction has to be executed by the majority of owners for the contract to execute the transaction. Our default invocation depth is the reason for missing a possible reachable state.

(ii) The tool is not able to recover the subtype for the generic **bytes** type in the EVM semantics.

(iii) Some contracts release funds only if a random number (usually generated using transaction and block parameters) matches a

Inv. depth	Prodigal	Suicidal	Greedy
1	131	127	682
2	156	141	682
3	157	141	682
4	157	141	682

Table 2: The table shows number of contracts flagged for various invocation depths. This analysis is done on a random subset of $25,000$–$100,000$ contracts.

predetermined value unlike in the case of the contract in Figure 10. In that contract the variable _guess is also a symbolic variable, hence, the solver can find a solution for condition on line 7. If there is a concrete value in place of _guess, the solver times out since the constraint involves a hash function (hard to invert by the SMT solver).

5.4 Summary and Observations

The symbolic execution engine of Maian flags 34, 200 contracts. With concrete validation engine or manual inspection, we have confirmed that around 97% of prodigal, 97% of suicidal and 69% of greedy contracts are true positive. The importance of analyzing the bytecode of the contracts, rather than Solidity source code, is demonstrated by the fact that only 1% of all contracts have source code. Further, among all flagged contracts, only 181 have verified source codes according to the widely used platform Etherscan, or in percentages only 1.06%, 0.47% and 0.49%, in the three categories of prodigal, suicidal, and greedy, respectively. We refer the reader to Table 1 for the exact summary of these results.

Furthermore, the maximal amount of Ether that could have been withdrawn from prodigal and suicidal contracts, before the block height BH, is nearly 4, 905 Ether, or 3.4 million US dollars[8] according to the exchange rate at the time of this writing. In addition, 6, 239 Ether (4.3 million US dollars) is locked inside posthumous contracts currently on the blockchain, of which 313 Ether (216, 000 US dollars) were sent to dead contracts after they have been killed.

Finally, the analysis given in Table 2 shows the number of flagged contracts for different invocation depths from 1 to 4. We tested 25, 000 contracts being for greedy, and 100, 000 for remaining categories, inferring that increasing depth improves results marginally, and an invocation depth of 3 is an optimal tradeoff point. Table 2 clearly shows that reasoning about contract traces, rather than a single contract invocation, reveals more vulnerabilities of prodigal and suicidal type. Compared to a single invocation, analysis based on two invocations detects an additional $10\% - 20\%$ contracts with potential bugs. Besides this quantitative increase, there is as well a particular qualitative increase of flagged contracts. Specifically, contracts that can be exploited by executing a two-invocation trace on average tend to be more complex and thus finding the vulnerability manually requires more effort.

Note, we have contacted the Ethereum Foundation for an ethical disclosure procedure, and we have given then the full list of found vulnerable contracts.

[8] Calculated at 693 USD/ETH [12].

6 RELATED WORK

Security and safety properties of smart contracts have received a lot of attention since several costly bugs and exploits took place [2, 10].

Dichotomy of smart contract bugs. The majority of the bugs in Ethereum-style smart contracts are due to the de-facto high-level implementation language, Solidity [41], whose runtime behaviour that diverge from the "intuitive understanding" of the language by the developers.

The early work by Delmolino *et al.* [11] distinguishes the following classes of problems: (a) contracts that do not refund their users, (b) missing encryptions of sensitive user data and (c) lack of incentives for the users to take certain actions. The property (a) is the closest to our notion of *greedy*. While that outlines the problem and demonstrates it on series of simple examples taught in a class, they do not provide a systematic approach for detection of smart contracts prone to this issue. Later works on contract security identify potential bugs, related to the concurrent transactions [39], mishandled exceptions [25], overly extensive gas consumption [7] and implementations of fraudulent financial schemes [5].[9]

In contrast to all those work, which focus on bad implementation practices or misused language semantics, we believe, our characterisation of several classes of contract bugs, such as greedy, prodigal, *etc*, is novel, as they are stated in terms of properties execution traces rather than particular instructions taken/states reached.

Reasoning about smart contracts. Several tools have been proposed to automatic detection of vulnerabilities in smart contracts, as well as for formal contract verification.

Oyente [25, 32] was the first tool that provided analysis targeting several specific issues: (a) mishandled exceptions, (b) transaction-ordering dependence, (c) timestamp dependence and (d) reentrancy [40], thus remedying the corner cases of Solidity/EVM semantics as well as some programming anti-patterns.

Other tools for symbolic analysis of EVM and/or Solidity have been developed more recently: Manticore [26], Mythrill [29, 30], Securify [38], and KEVM [21, 37], all focusing on detecting *low-level* safety violations and vulnerabilities, such as integer overflows, reentrancy, and unhandled exceptions, *etc*, neither of them requiring reasoning about contract execution traces. While it does not seem impossible to extend all these frameworks for handling trace-based properties discussed in this work, this has not been done yet, thus we cannot conduct a formal comparison. A very recent work by Grossman *et al.* [19] similar to our in spirit and providing a dynamic analysis of execution traces, focuses exclusively on detecting *non-callback-free* contracts (*i.e.*, prone to reentrancy attacks)—a vulnerability that is by now well studied.

Concurrently with our work, Kalra *et al.* developed Zeus [24], a framework for automated verification of smart contracts using abstract interpretation and symbolic model checking, accepting user-provided *policies* to verify for. Unlike Maian, Zeus conducts policy checking at a level of LLVM-like intermediate representation of a contract, obtained from Solidity code, and leverages a suite of standard tools, such as off-the-shelf constraint and SMT solvers [9, 20, 27]. Although Zeus also flags some contracts as "suicidal" (due to incorrect uses of `selfdestruct`), it does not provide a framework for

[9] See the works [4, 8] for a survey of known contract issues.

checking other trace properties, or under-approximating liveness properties, *i.e.*, for detecting prodigal or greedy contracts.

Various versions of EVM semantics [42] were implemented in Coq [23], Isabelle/HOL [3, 22], F* [6, 18], Idris [34], and Why3 [14, 36], followed by subsequent mechanised contract verification efforts. However, none of those efforts considered trace properties in the spirit of what we defined in Section 3.

7 CONCLUSION

We characterize vulnerabilities in smart contracts that are checkable as properties of an entire execution trace (possibly infinite sequence of their invocations). We show three examples of such trace vulnerabilities, leading to greedy, prodigal and suicidal contracts. and built a symbolic analysis tool MAIAN to find these. Analyzing 970, 898 contracts, MAIAN flags thousands of contracts vulnerable at a high true positive rate. At a scale of nearly one million contracts, MAIAN flags thousands of contracts as vulnerable, and successfully generates exploits for 69–99% of the subset we sample for validation.

REFERENCES

[1] Anthony Akentiev. 2018. Parity Multisig Github. (2018). https://github.com/paritytech/parity/issues/6995

[2] JD Alois. 2017. Ethereum Parity Hack May Impact ETH 500,000 or $146 Million. (2017).

[3] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *POST (LNCS)*, Vol. 10204. Springer, 164–186.

[5] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2017. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *CoRR* abs/1703.03779 (2017).

[6] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.

[7] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*. 442–446.

[8] ConsenSys Diligence. 2018. Ethereum Smart Contract Security Best Practices. (2018). https://consensys.github.io/smart-contract-best-practices

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

[10] Michael del Castillo. June 17, 2016. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. (June 17, 2016).

[11] Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. 2016. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. In *FC 2016 International Workshops (LNCS)*, Vol. 9604. Springer, 79–94.

[12] Etherscan 2018. (2018). https://etherscan.io/

[13] Etherscan verified source codes 2018. Etherscan verified source codes. (2018). https://etherscan.io/contractsVerified

[14] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (LNCS)*, Vol. 7792. Springer, 125–128.

[15] Go-ethereum 2018. (2018). https://github.com/ethereum/go-ethereum

[16] Patrice Godefroid. 2011. Higher-order Test Generation. In *PLDI*. ACM, 258–269.

[17] Governmental bug 2018. GovernMental's 1100ETH jackpot payout is stuck because it uses too much gas. (2018). https://www.reddit.com/r/ethereum/comments/4ghzhv/

[18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST (LNCS)*, Vol. 10804. Springer, 243–269.

[19] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28.

[20] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV, Part I (LNCS)*, Vol. 9206. Springer, 343–361.

[21] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Daejun Park, Yi Zhang, Brandon Moore, and Grigore Rosu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *CSF*. IEEE. To appear.

[22] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *1st Workshop on Trusted Smart Contracts (LNCS)*, Vol. 10323. Springer, 520–535.

[23] Yoichi Hirai. 2017. Ethereum Virtual Machine for Coq (v0.0.2). Published online on 5 March 2017. (2017). https://goo.gl/DxYFwK

[24] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing Safety of Smart Contracts. In *NDSS*. To appear.

[25] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.

[26] Manticore 2018. (2018). https://github.com/trailofbits/manticore

[27] Kenneth L. McMillan. 2007. Interpolants and Symbolic Model Checking. In *VMCAI (LNCS)*, Vol. 4349. Springer, 89–90.

[28] Mortal 2018. Contract mortal. (2018). https://etherscan.io/address/0x4671ebe586199456ca28ac050cc9473cbac829eb#code

[29] Bernhard Mueller. January 2018. How Formal Verification Can Ensure Flawless Smart Contracts. (January 2018). https://goo.gl/9wUFE1

[30] Mythril 2018. (2018). https://github.com/b-mueller/mythril/

[31] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008). http://bitcoin.org/bitcoin.pdf

[32] Oyente 2018. Oyente: An Analysis Tool for Smart Contracts. (2018). https://github.com/melonproject/oyente

[33] Parity bug 2018. The guy who blew up Parity didn't know what he was doing. (2018). https://www.reddit.com/r/CryptoCurrency/comments/7beos3/

[34] Jack Pettersson and Robert Edström. 2016. *Safer Smart Contracts through Type-Driven Development*. Master's thesis. Chalmers University of Technology, Sweden.

[35] George Pîrlea and Ilya Sergey. 2018. Mechanising blockchain consensus. In *CPP*. ACM, 78–90.

[36] Christian Reitwiessner. 2015. Formal Verification for Solidity Contracts. (2015). https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts

[37] Grigore Rosu. December 2017. ERC20-K: Formal Executable Specification of ERC20. (December 2017). https://runtimeverification.com/blog/?p=496

[38] Securify 2018. Securify: Formal Verification of Ethereum Smart Contracts. (2018). http://securify.ch/

[39] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *1st Workshop on Trusted Smart Contracts (LNCS)*, Vol. 10323. Springer, 478–493.

[40] Emin Gün Sirer. 2016. Reentrancy Woes in Smart Contracts. (2016). http://hackingdistributed.com/2016/07/13/reentrancy-woes/

[41] Solidity 2018. *Solidity: High-Level Language for Implementing Smart Contracts*. http://solidity.readthedocs.io/

[42] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf