

# 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 and 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 precisely 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 dollars 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. Prototypically, these distributed ledgers map accounts or addresses (the public half of a cryptographic key pair) 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 now exist, and as of writing the market value of the associated coins is over \$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,<sup>1</sup> unlike traditional consumer device software. Secondly, they are written in a new ecosystem of languages and runtime environments, the *de facto* standard for which is 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 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 as *trace vulnerabilities*. Unlike many previous works that have applied static and dynamic analyses to find bugs in contracts automatically [2–5], our work focuses on detecting vulnerabilities across a long 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

<sup>1</sup>Other than by “hard forks”, which are essentially decisions of the community to change the protocol and are extremely rare.

be susceptible to be killed by any user. Our precisely defined properties capture many well-known examples of known anecdotal bugs [1, 6, 7], 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 are powerful enough to find the infamous Parity bug that indirectly caused 200 million dollars worth of Ether, which is not found by previous analyses. A total of 34,200 (2,365 distinct) contracts are flagged as potentially buggy, directly carry the equivalent of millions of dollars worth of Ether. 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 a execution traces — potentially infinite sequence of invocations of a contract. Previous techniques and tools [3] are not designed to find these bugs because they only model behavior for a single call to a contract.
- We provide formal 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).
- 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 Background on Smart Contracts

Smart contracts in Ethereum run on Ethereum Virtual Machine (EVM), a stack-based execution runtime [8].

Different source languages compile to the EVM semantics, the predominant of them being Solidity [9]. A smart contract embodies the concept of an autonomous agent, identified by its program logic, its identifying address, and its associated balance in Ether. Contracts, like other addresses, can receive Ether from external agents storing it in their `balance` field; they can also send Ether to other addresses via transactions. A smart contract is created by the *owner* who sends an initializing transaction, which contains the contract bytecode and has no specified recipient. Due to the persistent nature of the blockchain, once initialized, the contract code cannot be updated. Contracts live perpetually unless they are explicitly terminated by executing the `SUICIDE` bytecode instruction, after which they are no longer invocable or called *dead*. When alive, contracts can be invoked many times. Each invocation is triggered by sending a transaction to the contract address, together with input data and a fee (known as *gas*) [8]. The mining network executes separate instances of the contract code and agrees on the outputs of the invocation via the standard blockchain consensus protocol, *i.e.*, Nakamoto consensus [10, 11]. 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.

The EVM allows contract functions to have local state, while the contracts may have global variables stored on the blockchain. Contracts can invoke other contracts via message calls; outputs of these calls, considered to be a part of the same transaction, are returned to the caller during the runtime. Importantly, calls are also used to send Ether to other contracts and non-contract addresses. The `balance` of a contract can be read by anyone, but is only updated via calls from other contracts and externally initiated transactions.

Contracts can be executed repeatedly over their lifetime. A transaction can run one *invocation* of the contract and an execution *trace* is a (possibly infinite) sequence of runs of a contract recorded on the blockchain. Our work shows the importance of reasoning about execution traces of contracts with a class of vulnerabilities that has not been addressed in prior works, and provides an automatic tool to detect these issues.

### 2.2 Contracts with Trace Vulnerabilities

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

Note that any characterization of bugs must be taken with a grain of salt, since one can always argue that the

```

1 function payout(address[] recipients,
2               uint256[] amounts) {
3     require(recipients.length==amounts.length);
4     for (uint i = 0; i < recipients.length; i++) {
5         /* ... */
6         recipients[i].send(amounts[i]);
7     }

```

Figure 1: Bounty contract; payout leaks Ether.

exposed behavior embodies intent — as was debated in the case of the DAO bug [6]. Our characterization of vulnerabilities is based, in part, on anecdotal incidents reported publicly [6, 7, 12]. To the best of our knowledge, however, our characterization is the first to precisely define checkable properties of such incidents and measure their prevalence. Note that there are several valid reasons for contracts for being killable, holding funds indefinitely under certain conditions, or giving them out to addresses not known at the time of deployment. For instance, a common security best practice is that when under attack, a contract should be killed and should return funds to a trusted address, such as that of the owner. Similarly, 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 (when under attack), to addresses that have sent Ether to it 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— which is not an owner, has never deposited Ether in the contract, and has provided no data that is difficult to fabricate by an arbitrary observer—we deem this as a vulnerability. We are interested in finding such contracts, which we call as *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. In the contract, the function `payout` sends to a list of recipients specified amounts of Ether. It is clear from the function definition that the recipients and the amounts are provided as inputs, and anybody can call the function (i.e., the function does not have restrictions on the sender). The message sender of the transaction is not checked for; the only check is on the size of lists. Therefore, any user can invoke this function with a list of recipients of her choice, and completely drain its Ether.

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. Examples of such

```

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.

contracts 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, which would make it to execute the `SUICIDE` instruction, 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 user [13] 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 the shortened code fragment of this contract given in Figure 2. To kill the contract, the user invokes two different functions: one to set the ownership,<sup>2</sup> and one to actually kill the contract. That is, the user 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 user invokes the 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

<sup>2</sup>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.

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. `send`, `call`, `transfer`), or such instructions are not reachable. An example of contract that lacks commands 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

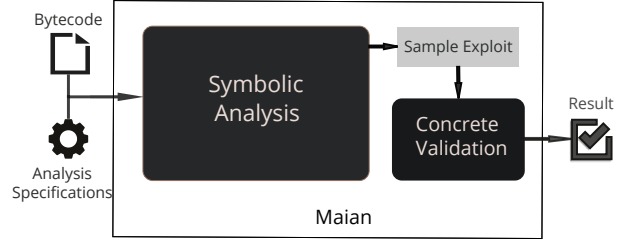


Figure 4: MAIAN

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 [2,5,14–17].

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 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 [3]. 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 concrete validation component takes the inputs generated by symbolic analysis compo-



ment 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 Execution Model and Trace Properties

A life cycle of a smart contract can be represented by a sequence of the contract’s states, which describe the values of the contract’s fields, as well as its balance, interleaved with instructions and *irreversible* actions it performs modifying the global context of the blockchain, such transferring Ether or committing suicide. One can consider a contract to be *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 qualified as *trace-safety* ones, meaning that “during a final execution nothing bad happens”. Proving the absence of some other high-level bugs will, however, require establishing a statement of a different kind, namely, “something good must eventually happen”. Such properties are known as *liveness* ones and require reasoning about progress in executions. An example of such property would be an assertion that a contract can always execute a finite number of steps in order to perform an action of interest, such as transferring money, in order to be considered non-greedy.

In this section, we formally define the execution model of Ethereum smart contracts, allowing one to pinpoint the vulnerabilities characterised in Section 2.2. The key idea of our bug-catching approach is to formulate the erroneous behaviours as predicates of observed contract *traces*, rather than individual configurations and instruction invocations, occurring in the process of an execution. By doing so, we are able to (a) capture the prodigal/suicidal contracts via conditions that relate the unwelcome agents gaining, at some point, access to a contract’s funds or suicide functionality by finding a way around a planned semantics, and (b) respond about repeating behavioural patterns in the contract life cycles, allowing us to detect greedy contracts.

#### 3.1 EVM Semantics and Execution Traces

We begin with defining contract execution traces by adopting a low-level execution semantics of an EVM-like language in the form of ETHERLITE-like calculus [2]. ETHERLITE implements a small-step stack machine, operating on top of a global configuration of the blockchain, which used to retrieve contract codes and

ascribe Ether balance to accounts, as well as manipulations with the local contract configuration. As customary in Ethereum, such agent is represented by its address *id*, and might be a contract itself. For the purpose of this work, we simplify the semantics of ETHERLITE by eliding the executions resulting in exceptions, as reasoning about such is orthogonal to the properties of interest. Therefore, the configurations  $\delta$  of the ETHERLITE abstract machine are defined as follows:

Configuration	$\delta \triangleq \langle A, \sigma \rangle$
Execution stack	$A \triangleq \langle M, id, pc, s, m \rangle \cdot A \mid \varepsilon$
Message	$m \triangleq \{sender \mapsto id; value : \mathbb{N}; data \mapsto \dots\}$
Blockchain state	$\sigma \triangleq id \mapsto \{bal : \mathbb{N}; code? \mapsto M; f? \mapsto v\}$

That is, a contract execution configuration consists of an activation record stack *A* and a blockchain context  $\sigma$ . An activation record stack *A* is a list of tuples  $\langle M, id, pc, s, m \rangle$ , where *id* and *M* are the address and the code of the contract currently being executed, *pc* is a program counter pointing to the next instruction to be executed, *s* is a local operand stack, and *m* is the last message used to invoke the contract execution. Among other fields, *m* stores the identity of the *sender*, the amount *value* of the ether being transferred (represented as a natural number), as well as auxiliary fields (*data*) used to provide additional arguments for a contract call, which we will be omitting for the sake of brevity. Finally, a simplified context  $\sigma$  of a blockchain is encoded as a finite partial mapping from an account *id* to its balance and contract code *M* and its mutable state, mapping the field names *f* to the corresponding values,<sup>3</sup> which both are optional (hence, marked with ?) and are only present for contract-storing blockchain records. We will further refer to the union of a contract’s fields entries  $f \mapsto v$  and its balance entry  $bal \mapsto z$  as a *contract state*  $\rho$ .

Figure 5 presents selected rules for a smart contract execution in ETHERLITE.<sup>4</sup> The rules for storing and loading values to/from a contract’s field *f* are standard. Upon calling another account, a rule CALL is executed, which required the amount of Ether *z* to be transferred to be not larger than the contract *id*’s current balance, and changes the activation record stack and the global blockchain context accordingly. Finally, the rule SUICIDENONEMPTYSTACK provides the semantics for the SUICIDE instruction (for the case of a non-empty activation record stack), in which case all funds of the terminated contract *id* are transferred to the caller’s *id*’.

An important addition we made to the semantics of ETHERLITE are execution *labels*, which allow to distin-

<sup>3</sup>For simplicity of presentation, we treat all contract state as *persistent*, eliding operations with *auxiliary* memory, such as MLOAD/MSTORE.

<sup>4</sup>The remaining rules can be found in the work by Luu *et al.* [2].

$$\begin{array}{c}
\text{SSTORE} \\
\hline
M[pc] = \text{SSTORE} \quad \sigma' = \sigma[id][f \mapsto v] \\
\langle \langle M, id, pc, f \cdot v \cdot s, m \rangle \cdot A, \sigma \rangle \xrightarrow{\text{sstore}(f, v)} \langle \langle M, id, pc + 1, s, m \rangle \cdot A, \sigma' \rangle \\
\\
\text{SLOAD} \\
\hline
M[pc] = \text{SLOAD} \quad v = \sigma[id][f] \\
\langle \langle M, id, pc, f \cdot s, m \rangle \cdot A, \sigma \rangle \xrightarrow{\text{sload}(f, v)} \langle \langle M, id, pc + 1, v \cdot s, m \rangle \cdot A, \sigma \rangle \\
\\
\text{CALL} \\
\hline
M[pc] = \text{CALL} \quad \sigma[id][bal] \geq z \\
s = id' \cdot z \cdot args \cdot s' \quad a = \langle M, id, pc + 1, s', m \rangle \\
m' = \{sender \mapsto id; value \mapsto z; data \mapsto args\} \quad M' = \sigma[id'] [code] \\
\sigma' = \sigma[id][bal \mapsto \sigma[id][bal] - z] \quad \sigma'' = \sigma'[id'] [bal \mapsto \sigma'[id'] [bal] + z] \\
\langle \langle M, id, pc, s, m \rangle \cdot A, \sigma \rangle \xrightarrow{\text{call}(id', m')} \langle \langle M', id', 0, \epsilon, m' \rangle \cdot a \cdot A, \sigma'' \rangle \\
\\
\text{SUICIDENONEMPTYSTACK} \\
\hline
M[pc] = \text{SUICIDE} \quad s = id' \cdot s' \quad a = \langle M', pc', s'', m' \rangle \\
\sigma' = \sigma[id'] [bal \mapsto (\sigma[id'] [bal] + \sigma[id] [bal])] \quad \sigma'' = \sigma'[id'] [bal \mapsto 0] \\
\langle \langle M, id, pc, s, m \rangle \cdot A, \sigma \rangle \xrightarrow{\text{suicide}(id')} \langle \langle M', id', pc', 1 \cdot s'', m' \rangle \cdot A, \sigma'' \rangle
\end{array}$$

Figure 5: Selected execution rules of ETHERLITE.

guish between specific transitions being taken, as well as their parameters, and are defined as follows:

$$\ell \triangleq \text{sstore}(f, v) \mid \text{sload}(f, v) \mid \text{call}(id, m) \mid \text{suicide}(id) \mid \dots$$

For instance, a transition label of the form  $\text{call}(id, m)$  captures the fact that a currently running contract has transferred control to another contract  $id$ , by sending it a message  $m$ , while the label  $\text{suicide}(id)$  would mean a suicide of the current contract, with transfer of all of its funds to the account (a contract's or not)  $id$ .

With the labelled operational semantics at hand, we can now provide a definition of partial contract execution *traces* as sequences of interleaved contract states  $\rho_i$  and transition labels  $\ell_j$  as follows:

**Definition 3.1** (Projected contract trace). A *partial* projected trace  $t = \widehat{\tau}_{id}(\sigma, m)$  of a contract  $id$  in an initial blockchain state  $\sigma$  and an incoming message  $m$  is defined as a sequence  $[\langle \rho_0, \ell_0 \rangle, \dots, \langle \rho_n, \ell_n \rangle]$ , such that for every  $i \in \{0 \dots n\}$ ,  $\rho_i = \sigma_i[id]_{bal, \bar{f}}$ , where  $\sigma_i$  is the blockchain state at the  $i^{\text{th}}$  occurrence of a configuration of the form,  $\langle \langle \bullet, id, \bullet, \bullet, \bullet \rangle, \sigma_i \rangle$  in an execution sequence starting from the configuration  $\langle \langle \sigma[id][code], id, 0, \epsilon, m \rangle \cdot \epsilon, \sigma \rangle$ , and  $\ell_i$  is a label of an immediate next transition.

In other words,  $\widehat{\tau}_{id}(\sigma, m)$  captures the states of a contract  $id$ , interleaved with the transitions taken “on its behalf” and represented by the corresponding labels, starting from the initial blockchain  $\sigma$  and triggered by the message  $m$ . The notation  $\sigma[id]_{bal, \bar{f}}$  stands for a projection to the corresponding components of the contract entry in  $\sigma$ . States and transitions of contracts *other* than  $id$  and involved into the same execution are, thus, ignored.

Given a (partial) projected trace  $\widehat{\tau}_{id}(\sigma, m)$ , we say that it is *complete*, if it corresponds to an execution, whose

last configuration is  $\langle \epsilon, \sigma' \rangle$  for some  $\sigma'$ . The following definition captures the behaviors of multiple subsequent transactions with respect to a contract of interest.

**Definition 3.2** (Multi-transactional contract trace). A contract trace  $t = \tau_{id}(\sigma, \bar{m}_i)$ , for a sequence of messages  $\bar{m}_i = m_0, \dots, m_n$ , is a concatenation of single-transaction traces  $\widehat{\tau}_{id}(\sigma_i, m_i)$ , where  $\sigma_0 = \sigma$ ,  $\sigma_{i+1}$  is a blockchain state at the end of an execution starting from a configuration  $\langle \langle \sigma[id][code], id, 0, \epsilon, m_i \rangle \cdot \epsilon, \sigma_i \rangle$ , and all traces  $\widehat{\tau}_{id}(\sigma_i, m_i)$  are complete for  $i \in \{0, \dots, n-1\}$ .

As stated, the definition does *not* require a trace to end with a complete execution at the last transaction. For convenience, we will refer to the last element of a trace  $t$  by  $\text{last}(t)$  and to its length as  $\text{length}(t)$ .

## 3.2 Characterising Safety Violations

The notion of contract traces allows us to formally capture the definitions of buggy behaviors, described previously in Section 2.2. First, we turn our attention to the prodigal/suicidal contracts, which can be uniformly captured by the following higher-order trace predicate.

**Definition 3.3** (Leaky contracts). A contract with an address  $id$  is considered to be *leaky* with respect to predicates  $P$ ,  $R$  and  $Q$ , and a blockchain state  $\sigma$  (denoted  $\text{leaky}_{P,R,Q}(id, \sigma)$ ) iff there exists a sequence of messages  $\bar{m}_i$ , such that for a trace  $t = \tau_{id}(\sigma, \bar{m}_i)$ :

1. the *precondition*  $P(\sigma[id][code], t_0, m_0)$  holds,
2. the *side condition*  $R(t_i, m_0)$  holds for all  $i < \text{length}(t)$ ,
3. the *postcondition*  $Q(t_n, m_0)$  holds for  $t_n = \text{last}(t)$ .

Definition 3.3 of leaky contracts is *relative* with respect to a current state of a blockchain: a contract that is currently leaky may stop being such in the future. Also, notice that the “triggering” initial message  $m_0$  serves as an argument for all three parameter predicates. We will now show how two behaviors observed earlier can be encoded via specific choices of  $P$ ,  $R$ , and  $Q$ .<sup>5</sup>

**Prodigal contracts.** A contract is considered prodigal if it sends Ether, immediately or after a series of transitions (possibly spanning multiple transactions), to an arbitrary sender. This intuition can be encoded via the following choice of  $P$ ,  $R$ , and  $Q$  for Definition 3.3:

$$\begin{aligned}
P(M, \langle \rho, \ell \rangle, m) &\triangleq m[\text{sender}] \notin \text{im}(\rho) \wedge m[\text{value}] = 0 \\
R(\langle \rho, \ell \rangle, m) &\triangleq \text{True} \\
Q(\langle \rho, \ell \rangle, m) &\triangleq \ell = \text{call}(m[\text{sender}], m') \wedge m'[\text{value}] > 0 \\
&\vee \ell = \text{delegatecall}(m[\text{sender}]) \\
&\vee \ell = \text{suicide}(m[\text{sender}])
\end{aligned}$$

According to the instantiation of the parameter predicates above, a prodigal contract is exposed by a trace that

<sup>5</sup>In most of the cases, it is sufficient to take  $R \triangleq \text{True}$ , but in Section 6 we hint certain properties that require a non-trivial side condition.

is triggered by a message  $m$ , whose sender does *not* appear in the contract's state ( $m[sender] \notin \text{im}(\rho)$ ), *i.e.*, it is not the owner, and the Ether payload of  $m$  is zero. To expose the erroneous behavior of the contract, the postcondition checks that the transition of a contract is such that it transfer funds or control (*i.e.*, corresponds to CALL, DELEGATECALL or SUICIDE instructions [8]) with the recipient being the sender of the initial message. In the case of sending funds via CALL we also check that the amount being transferred is non zero. In other words, the initial caller  $m[sender]$ , unknown to the contract, got himself some funds without any monetary contribution! In principle, we could ensure minimality of a trace, subject to the property, by imposing a non-trivial side condition  $R$ , although this does not affect the class of contracts exposed by this definition.

**Suicidal contracts.** A definition of a suicidal contract is very similar to the one of a prodigal contract. It is delivered by the following choice of predicates:

$$\begin{aligned} P(M, \langle \rho, \ell \rangle, m) &\triangleq \text{SUICIDE} \in M \wedge m[sender] \notin \text{im}(\rho) \\ R(\langle \rho, \ell \rangle, m) &\triangleq \text{True} \\ Q(\langle \rho, \ell \rangle, m) &\triangleq \ell = \text{suicide}(m[sender]) \end{aligned}$$

That is, a contract is suicidal if its code  $M$  contains the SUICIDE instruction and the corresponding transition can be triggered by a message sender, that does not appear in the contract's state at the moment of receiving the message, *i.e.*, at the initial moment  $m[sender] \notin \text{im}(\rho)$ .

### 3.3 Characterising Liveness Violations

A contract is considered *locking* at a certain blockchain state  $\sigma$ , if at any execution originating from  $\sigma$  prohibits certain transitions to be taken. 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 final traces up to a certain length:

**Definition 3.4** (Locking contracts). A contract with an address  $id$  is considered to be *locking* with respect to predicates  $P$  and  $R$ , the transaction number  $k$ , and a blockchain state  $\sigma$  (denoted  $\text{locking}_{P,R,k}(id, \sigma)$ ) *iff* for all sequences of messages  $\overline{m}_i$  of length less or equal than  $k$ , the corresponding trace  $t = \tau_{id}(\sigma, \overline{m}_i)$  satisfies:

1. the *precondition*  $P(\sigma[id][code], t_0, m_0)$ ,
2. the *side condition*  $R(t_i, m_0)$  for all  $i \leq \text{length}(t)$ .

Notice that, unlike Definition 3.3, this Definition does not require a postcondition, as it is designed to under-approximate potentially infinite traces, up to a certain length  $k$ ,<sup>6</sup> so the “final state” is irrelevant.

<sup>6</sup>We discuss viable choices of  $k$  in Section 5.

**Greedy contracts.** In order to specify a property asserting that in an interaction with up to  $k$  transactions, a contract does not allow to release its funds, we instantiate the predicates from Definition 3.4 as follows:

$$\begin{aligned} P(M, \langle \rho, \ell \rangle, m) &\triangleq \rho[bal] > 0 \\ R(\langle \rho, \ell \rangle, m) &\triangleq \neg \left( \begin{array}{l} \ell = \text{call}(m[sender], m') \wedge m'[value] > 0 \\ \vee \ell = \text{delegatecall}(m[sender]) \\ \vee \ell = \text{suicide}(m[sender]) \end{array} \right) \end{aligned}$$

Intuitively, the definition of a greedy contract is dual to the notion of a prodigal one, as witnessed by the above formulation: at any trace starting from an initial state, where the contract holds a non-zero balance, no transition transferring the corresponding funds (*i.e.*, matched by the side condition  $R$ ) can be taken, no matter what is the *sender's* identity. That is, this definition covers the case of contract's *owner* as well: *no one* can withdraw any funds from the contract.

## 4 The Algorithm and the Tool

MAIAN is a symbolic analyzer for smart contract execution traces, for the properties defined in Section 3. It operates by taking as input a contract in its byte-code form and a concrete starting block value from the Ethereum blockchain as the input context, flagging contracts that are outlined in Section 2.2. When reasoning about contract traces, MAIAN follows the ETHERLITE rules, described in Section 3.1, executing them symbolically. During the execution, which starts from a contract state satisfying the precondition of property of interest (*cf.* Definitions 3.3 and 3.4), it checks if there exists an execution trace which violates the property and a set of candidate values for input transactions that trigger the property violation. For the sake of tractability of the analysis, it does not keep track of the entire blockchain context  $\sigma$  (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 byte-code and a starting concrete context capturing values of the blockchain. MAIAN reasons about values read from

input transaction fields and block parameters<sup>7</sup> 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, *i.e.*, have only true positives at the expense of coverage [18]. 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 we assume to have reached the end of some contract function, and thus restart the symbolic execution again from the first bytecode instruction to simulate the next function call. On the other hand, the terminating instruction can be invalid (*e.g.*, non-existing instruction code or invalid jump destination), in which case we terminate the search down this path and backtrack 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.

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 makes no attempt at alias analysis and simply terminates the path being search and backtracks 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, these are not exactly under the control of the external users. MAIAN does not use their concrete values as it needs to reason about invocations of the contract across multiple invocations, *i.e.*, at different blocks.

**Flagging Violations.** Finally, when the depth-first search in the space of the contract execution reaches a state where the desired 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

<sup>7</sup>Those being CALLVALUE, CALLER, NUMBER, TIMESTAMP, BLOCKHASH, BALANCE, ADDRESS, and ORIGIN.



use Z3 [19] as our solver, which provides concrete values that make the input formula satisfiable. We use these values as the concrete data for our 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 [20] 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, 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 4,799,998, which is the last block as of December 26, 2017. Ethereum blockchain has only contract bytecodes. To obtain the original (Solidity) source codes, we refer to the Etherscan service [21] 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 our 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.<sup>8</sup>

**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.

**Contract Characteristics.** The number of contracts has increased tenfold from Dec, 2016 to Dec, 2017 and 176-fold since Dec, 2015. However, the distribution of Ether balance across contracts follows a skewed distribution. Less than 1% of the contracts have more than 99% of the Ether in the ecosystem. This suggests that a vulnerability in any one of these high-profile contracts can affect a

<sup>8</sup>We also concretely validate the flagged candidates which were killed before BH as well.

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

Table 1: Final results using invocation depth 3 at block height BH. Column 1 reports number of flagged contracts, and the distinct among these. Column 2 shows the number of flagged which have no 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.

large fraction of the entire Ether balance. Note that contracts interact with each other, therefore, a vulnerability in one contract may affect many others holding Ether, as demonstrated by the recent infamous Parity library which was used by wallet contracts with \$200 million US worth of Ether [1].

## 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).

**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.<sup>9</sup> 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 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, SUICIDE, or DELEGATECALL).
- (ii) Contracts that accept Ether, and in their bytecode have at least one of CALL, 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 ob-

<sup>9</sup>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 6: A prodigal contract.

```

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

tain true positive rate among greedy contracts of 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 4,799,998 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 6, we give an example of a prodigal contract. The function `tap` seems to lock Ether because the condition at 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`. Note, on a byte-code level, the EVM can only load chunks of 32 bytes of input data. At line 3 in `tap` the first 20 bytes of `nickname` are assigned to the global variable `prev`, while 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

```

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

the semantic of `SUICIDE` instruction enforce it to send all of its balance to an address provided to the instruction. In Figure 7, the contract Thing [22] is inherited from a base contract Mortal. The contract implements a review system in which public reviews an ongoing topic. Among others, the contract 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.

**Suicidal contracts.** A contract can be killed by exploiting an unprotected `SUICIDE` instruction. A trivial example is a public kill function which hosts the suicide instruction. Sometimes, `SUICIDE` is protected by a weak condition, such as in the contract Dividend given in Figure 8. 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` instruction 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. For instance, the contract Mortal given in Figure 7 checks whether it is the owner that calls the `kill` function. Hence, it requires an attacker to become the owner of the contract to kill it. So, this contract requires two invocations to be killed: one call to the function `mortal` used to become an owner of the con-

```

1 contract SimpleStorage {
2   uint storedData; address storedAddress;
3   event flag(uint val, address addr);
4
5   function set(uint x, address y) {
6     storedData = x; storedAddress = y;
7   }
8   function get() constant
9     returns(uint retVal, address retAddr) {
10    return (storedData, storedAddress);
11  }
12 }

```

Figure 9: A contract that locks Ether.

tract and one call to the function `kill` to kill the contract. A more secure contract would leverage the `mortal` function to a constructor so that the function is called only once when the contract is deployed. Note, the recent Parity bug similarly also requires two invocations [1].

**Greedy contracts.** The contract `SimpleStorage`, given in Figure 9, 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.

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

- (i) Our 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 one value (say `true`) when the call succeeds.
- (ii) MAIAN may assign values to symbolic variables related to block state (*e.g.*, `timestamp` and `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 11, the `_guess` value depends on the values of block parameters, which cannot be forced to take on the concrete values found by our analyzer.

```

1 function confirmTransaction(uint tId)
2   ownerExists(msg.sender) {
3   confirmations[tId][msg.sender] = true;
4   executeTransaction(tId);
5 }
6 function executeTransaction(uint tId) {
7   // In case of majority
8   if (isConfirmed(tId)) {
9     Transaction tx = transactions[tId];
10    tx.executed = true;
11    if (tx.destination.call.value(tx.value)
12        (tx.data))
13      /*...*/
14    }
15 }

```

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

```

1 function RandomNumber() returns(uint) {
2   /*...*/
3   last = seed^(uint(sha3(block.blockhash(
4     block.number), nonces[msg.sender]))*0
5     x000b0007000500030001);
6 }
7 function Guess(uint _guess) returns (bool) {
8   if (RandomNumber() == _guess) {
9     if (!msg.sender.send(this.balance)) throw;
10    /*...*/
11  } /*...*/
12 }

```

Figure 11: False positive, flagged as a prodigal contract.

**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 10, 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) Our 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 predetermined value unlike in the case of the contract in Figure 11. 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



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.

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 5.9 million US dollars<sup>10</sup> according to the exchange rate at the time of this writing. In addition, 6,239 Ether (7.5 million US dollars) is locked inside posthumous contracts currently on the blockchain, of which 313 Ether (379,940 US dollars) have been 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.

## 6 Related Work

**Dichotomy of smart contract bugs.** The early work by Delmolino *et al.* [24] 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 safety and security identify potential bugs, related to the concurrent transactional executions [25], mishandled exceptions [2], overly extensive gas consumption [14] and implementations of fraudulent financial schemes [26].<sup>11</sup>

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.

<sup>10</sup>Calculated at 1,210 USD/Eth [23].

<sup>11</sup>See the works [27, 28] for a survey of known contract issues.

**Reasoning about smart contracts.** OYENTE [2, 3] was the first symbolic execution-based tool that provided analysis targeting several specific issues: (a) mishandled exceptions, (b) transaction-ordering dependence, (c) timestamp dependence and (d) reentrancy [29], thus remedying the corner cases of Solidity/EVM semantics (a) as well as some programming anti-patterns (b)–(d).

Other tools for symbolic analysis of EVM and/or EVM have been developed more recently: MANTICORE [17], MYTHRILL [15, 16], SECURIFY [5], and KEVM [30, 31], 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. A very recent work by Grossman *et al.* [32] 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 [4], 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 [19, 33, 34]. ZEUS does not provide a general framework for checking trace properties, or under-approximating liveness properties.

Various versions of EVM semantics [8] were implemented in Coq [35], Isabelle/HOL [36, 37], F\* [38], Idris [39], and Why3 [40, 41], 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.

Several contract languages were proposed recently that distinguish between *global actions* (*e.g.*, sending Ether or terminating a contract) and *instructions* for ordinary computations [42, 43], for the sake of simplified reasoning about contract executions. For instance, the work on the contract language SCILLA [43] shows how to encode in Coq [44] and formally prove a property, which is very similar to a contract being *non-leaky*, as per Definition 3.3 instantiated with a non-trivial side condition *R*.

## 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. Analyzing 970,898 contracts, our new tool MAIAN flags thousands of contracts vulnerable at a high true positive rate.

## References

- [1] A. Akentiev, “Parity multisig github.” [Online]. Available: <https://github.com/paritytech/parity/issues/6995>
- [2] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *CCS*. ACM, 2016, pp. 254–269.
- [3] “Oyente: An Analysis Tool for Smart Contracts,” 2018. [Online]. Available: <https://github.com/melonproject/oyente>
- [4] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: Analyzing safety of smart contracts,” in *NDSS*, 2018, to appear.
- [5] “Securify: Formal Verification of Ethereum Smart Contracts,” 2018. [Online]. Available: <http://securify.ch/>
- [6] M. del Castillo, “The DAO Attacked: Code Issue Leads to \$60 Million Ether Theft,” June 17, 2016.
- [7] “Governmental’s 1100eth jackpot payout is stuck because it uses too much gas.” [Online]. Available: <https://www.reddit.com/r/ethereum/comments/4ghzhv/>
- [8] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger.” [Online]. Available: <https://ethereum.github.io/yellowpaper/paper.pdf>
- [9] *Solidity: High-Level Language for Implementing Smart Contracts*. [Online]. Available: <http://solidity.readthedocs.io/>
- [10] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008. [Online]. Available: <http://bitcoin.org/bitcoin.pdf>
- [11] G. Pirlea and I. Sergey, “Mechanising blockchain consensus,” in *CPP*. ACM, 2018, pp. 78–90.
- [12] J. Alois, “Ethereum Parity Hack May Impact ETH 500,000 or \$146 Million,” 2017.
- [13] “The guy who blew up parity didn’t know what he was doing.” [Online]. Available: <https://www.reddit.com/r/CryptoCurrency/comments/7beos3/>
- [14] T. Chen, X. Li, X. Luo, and X. Zhang, “Under-optimized smart contracts devour your money,” in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*, 2017, pp. 442–446.
- [15] “Mythril,” 2018. [Online]. Available: <https://github.com/b-mueller/mythril/>
- [16] B. Mueller, “How Formal Verification Can Ensure Flawless Smart Contracts,” January 2018. [Online]. Available: <https://goo.gl/9wUFE1>
- [17] “Manticore,” 2018. [Online]. Available: <https://github.com/trailofbits/manticore>
- [18] P. Godefroid, “Higher-order test generation,” in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11, 2011.
- [19] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.
- [20] “Go-ethereum.” [Online]. Available: <https://github.com/ethereum/go-ethereum>
- [21] “Etherscan verified source codes.” [Online]. Available: <https://etherscan.io/contractsVerified>
- [22] “Contract mortal.” [Online]. Available: <https://etherscan.io/address/0x4671ebe586199456ca28ac050cc9473cbac829eb#code>
- [23] “Etherscan.” [Online]. Available: <https://etherscan.io/>
- [24] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi, “Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab,” in *FC 2016 International Workshops*, ser. LNCS, vol. 9604. Springer, 2016, pp. 79–94.
- [25] I. Sergey and A. Hobor, “A Concurrent Perspective on Smart Contracts,” in *1st Workshop on Trusted Smart Contracts*, ser. LNCS, vol. 10323. Springer, 2017, pp. 478–493.
- [26] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, “Dissecting ponzi schemes on ethereum: identification, analysis, and impact,” *CoRR*, vol. abs/1703.03779, 2017.
- [27] N. Atzei, M. Bartoletti, and T. Cimoli, “A Survey of Attacks on Ethereum Smart Contracts (SoK),” in *POST*, ser. LNCS, vol. 10204. Springer, 2017, pp. 164–186.
- [28] ConsenSys Diligence, “Ethereum Smart Contract Security Best Practices,” 2018. [Online]. Available: <https://consensys.github.io/smart-contract-best-practices>
- [29] E. G. Sirer, “Reentrancy Woes in Smart Contracts.” [Online]. Available: <http://hackingdistributed.com/2016/07/13/reentrancy-woes/>
- [30] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, “KEVM: A Complete Semantics of the Ethereum Virtual Machine,” *Tech. Rep.*, 2017.
- [31] G. Rosu, “ERC20-K: Formal Executable Specification of ERC20,” December 2017. [Online]. Available: <https://runtimeverification.com/blog/?p=496>
- [32] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, “Online detection of effectively callback free objects with applications to smart contracts,” *PACMPL*, vol. 2, no. POPL, pp. 48:1–48:28, 2018.
- [33] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The SeaHorn Verification Framework,” in *CAV, Part I*, ser. LNCS, vol. 9206. Springer, 2015, pp. 343–361.
- [34] K. L. McMillan, “Interpolants and Symbolic Model Checking,” in *VMCAI*, ser. LNCS, vol. 4349. Springer, 2007, pp. 89–90.
- [35] Y. Hirai, “Ethereum Virtual Machine for Coq (v0.0.2),” Published online on 5 March 2017. [Online]. Available: <https://goo.gl/DxYFwK>
- [36] —, “Defining the Ethereum Virtual Machine for Interactive Theorem Provers,” in *1st Workshop on Trusted Smart Contracts*, ser. LNCS, vol. 10323. Springer, 2017, pp. 520–535.
- [37] S. Amani, M. Bégel, M. Bortin, and M. Staples, “Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL,” in *CPP*. ACM, 2018, pp. 66–77.
- [38] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, “Formal verification of smart contracts: Short paper,” in *PLAS*. ACM, 2016, pp. 91–96.
- [39] J. Pettersson and R. Edström, “Safer Smart Contracts through Type-Driven Development,” Master’s thesis, Chalmers University of Technology, Sweden, 2016.
- [40] C. Reitwiesner, “Formal verification for solidity contracts,” 2015. [Online]. Available: <https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts>
- [41] J. Filliâtre and A. Paskevich, “Why3 - Where Programs Meet Provers,” in *ESOP*, ser. LNCS, vol. 7792. Springer, 2013, pp. 125–128.
- [42] “Bamboo,” 2018. [Online]. Available: <https://github.com/pirapira/bamboo>
- [43] I. Sergey, A. Kumar, and A. Hobor, “Scilla: a smart contract intermediate-level language,” *CoRR*, vol. abs/1801.00687, 2018.
- [44] Coq Development Team, *The Coq Proof Assistant Reference Manual - Version 8.7*, 2018. [Online]. Available: <http://coq.inria.fr/>