
# TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts

**Johannes Krupp and Christian Rossow,** ***CISPA, Saarland University, Saarland Informatics Campus***

https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

**This paper is included in the Proceedings of the 27th USENIX Security Symposium.**

**August 15–17, 2018 • Baltimore, MD, USA**

ISBN 978-1-931971-46-1

**Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.**

# TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts

Johannes Krupp
*CISPA, Saarland University, Saarland Informatics Campus*

Christian Rossow
*CISPA, Saarland University, Saarland Informatics Campus*

## Abstract

Cryptocurrencies like Bitcoin not only provide a decentralized currency, but also provide a programmatic way to process transactions. Ethereum, the second largest cryptocurrency next to Bitcoin, is the first to provide a Turing-complete language to specify transaction processing, thereby enabling so-called *smart contracts*. This provides an opportune setting for attackers, as security vulnerabilities are tightly intertwined with financial gain. In this paper, we consider the problem of automatic vulnerability identification and exploit generation for smart contracts. We develop a generic definition of vulnerable contracts and use this to build TEETHER, a tool that allows creating an exploit for a contract given only its binary bytecode. We perform a large-scale analysis of all 38,757 unique Ethereum contracts, 815 out of which our tool finds working exploits for—completely automated.

## 1 Introduction

Cryptocurrencies are widely regarded as one of the most disruptive technologies of the last years. Their central value proposition is providing a decentralized currency—not backed by banks, but built on concepts of cryptography and distributed computing. This is achieved by using a *blockchain*, a publicly verifiable append-only data structure in which all transactions are recorded. This data structure is maintained by a peer-to-peer network. All nodes of this network follow a consensus protocol that governs the processing of transactions and keeps the blockchain in a consistent state. Furthermore, the consensus protocol guarantees that the blockchain cannot be modified by an attacker, unless they control a significant fraction of computation power in the entire network.

In 2009, the first cryptocurrency, Bitcoin [22], was launched. Since then, it has seen an unprecedented hype and has grown to a market capitalization of over 150 billion USD [1]. Although Bitcoin remains the predominant cryptocurrency, it also inspired many derivative systems. One of the most popular of these is Ethereum, the second largest cryptocurrency by overall market value as of mid 2018 [1].

Ethereum heavily extends the way consensus protocols handle transactions: While Bitcoin allows to specify simple checks that are to be performed when processing a transaction, Ethereum allows these rules to be specified in a Turing-complete language. This makes Ethereum the number one platform for so-called *smart contracts*.

A smart contract can be seen quite literally as a contract that has been formalized in code. As such, smart contracts can for example be used to implement fundraising schemes that automatically refund contributions unless a certain amount is raised in a given time, or shared wallets that require transactions to be approved of by multiple owners before execution. In Ethereum, smart contracts are defined in a high-level, JavaScript-like language called Solidity [2] and is then compiled into a bytecode representation suitable for consumption by the Ethereum Virtual Machine (EVM). Parties can interact with this contract through transactions in Ethereum. The consensus protocol guarantees correct contract execution in the EVM.

Of course, increased complexity comes at the cost of increased risk—Ethereum's Turing-complete Solidity is more error-prone than the simple checks that can be specified in Bitcoin. To make matters worse, once deployed, smart contracts are immutable and cannot be patched or updated. This causes an unparalleled coupling of software vulnerabilities and financial loss. In fact, since the inception of Ethereum in 2015, several cases of smart contract vulnerabilities have been observed [3, 4], causing a loss of tens of millions USD. As Ethereum is becoming more and more popular and valuable, the impact of smart contract vulnerabilities will only increase.

In this work, we tackle the problem of automatic vulnerability discovery and, more precisely, automatic ex-

ploit generation. Our attacker model assumes a regular Ethereum user without special capabilities whose goal it is to steal Ether from a given contract. Towards this, we first give a generic definition of contract vulnerabilities. Our definition is based on the observation that value transfer from one account (a contract) to another can only occur under few and well-defined conditions. In particular, we identify four critical, low-level EVM instructions that are necessarily involved in a value transfer: One used to create regular transactions (`CALL`), one for contract termination (`SELFDESTRUCT`), and two that can allow for code injection (`CALLCODE`, `DELEGATECALL`).

We propose a methodology to find vulnerable execution traces in a contract and employ symbolic execution to automatically create an exploit. Our approach is as follows: We search for certain critical paths in a contract's control flow graph. Specifically, we identify paths that lead to a critical instruction, where the instruction's arguments can be controlled by an attacker. Once a path is found, we leverage symbolic execution to turn this path into a set of constraints. Using constraint solving we can then infer the transactions an attacker has to perform to trigger the vulnerability. The special execution environment of smart contracts make this a non-trivial task. Most notably we show how to handle hash values symbolically, which are used extensively in smart contracts.

To demonstrate the utility of our methodology, we finally perform a large-scale analysis of 38,757 unique contracts extracted from the blockchain. teEther finds exploits for 815 (2.10%) of those—completely automated, without the need for human intervention or manual validation, and not requiring source code of contracts. Due to code sharing this puts the funds of at least 1,731 accounts at risk. Furthermore, a case-study indicates, that many of the underlying vulnerabilities are caused by the design choices of Solidity and misunderstandings about the EVM's execution model.

We summarize our core contributions as follows:

1. We provide a generic definition of vulnerable contracts, based on low-level EVM instructions (Section 3).
2. We develop a tool teEther that provides end-to-end exploit generation from a contract's bytecode only. To this end, we tackle several EVM-specific challenges, such as novel methodologies to handle hash values symbolically (Section 4).
3. We provide a large-scale vulnerability analysis of 38,757 unique contracts extracted from the Ethereum blockchain (Section 5).

# 2 Background

Ethereum is the second largest consensus-based transaction system next to Bitcoin, with a current market capitalization of over 110 billion USD [1]. Ethereum is often described as a second-generation blockchain, due to its support of so-called *smart contracts*—accounts controlled only by code which can handle transactions fully autonomously. In this section, we give a description of smart contracts, the Ethereum virtual machine, as well as the Ethereum execution model.

## 2.1 Transaction System

At the very core, Ethereum provides a public ledger for a new cryptocurrency called *Ether*. It provides a mapping between accounts—identified by a 160-bit address—and their balance. This ledger is backed by a network of mutually distrusting nodes, so-called miners. Users can submit transactions to the network in order to transfer Ether to other users or to invoke smart contracts. Miners will then process these transactions and, using a consensus protocol, agree on the outcome thereof. A processing fee is paid to the miner for each transaction to prevent resource exhaustion attacks on the network as well as to incentivize miners to process as many transactions as possible. All processed transactions are kept in a blockchain, a public hash-based append-only log, which allows anyone to verify the current state of the system.

## 2.2 Smart Contracts

A *smart contract* is a special type of Ethereum account that is associated with a piece of code. Like regular accounts, smart contracts can hold a balance of Ether. Additionally, smart contracts also have a (private) storage—a key-value store with 256-bit keys and 256-bit values. This storage is only "private" in the sense that it cannot be read or modified by other contracts, only by the contract itself. Furthermore, the storage is not *secret*. In fact is only cryptographically secured against external modifications. As all transactions are recorded in the public blockchain, the contents of a contract's private storage can be easily reconstructed by analyzing all transactions.

### 2.2.1 The Ethereum Virtual Machine (EVM)

The code of a smart contract is executed in a special purpose virtual machine, the Ethereum Virtual Machine (EVM). The EVM is a stack-based virtual machine with a wordsize of 256 bit. Besides arithmetic and control-flow instructions, the EVM also offers special instructions to access fields of the current transaction,

modify the contract's private storage, query the current blockchain state, and even create further transactions[1].

The EVM only provides integer arithmetic and cannot handle floating point values. To be able to denote values smaller than 1 Ether, balance is expressed in *Wei*, the smallest subdenomination of Ether. 1 Ether $= 10^{18}$ Wei.

In addition to the 256 bit word stack and the persistent storage the EVM also provides a byte-addressable memory, which serves as an input and output buffer to various instructions. For example, the `SHA3` instruction, which computes a Keccak-256 hash over a variable length data, reads its input from memory, where both the memory location and length of the input are provided via two stack arguments. Content of this memory is not persisted between contract executions, and the memory will always be set to all zero at the beginning of contract execution.

### 2.2.2 Solidity

Smart contracts are usually written in Solidity [2], a high-level language similar to JavaScript, and then compiled to EVM bytecode. For ease of readability, we will use Solidity in examples, however, our analysis is based on EVM bytecode only and completely Solidity-agnostic.

Smart contracts can be created by anyone by sending a special transaction to the zero address. After creation, the code of a contract is immutable, which means that smart contracts cannot be updated or patched. While some attempts have been made to create "updatable" contracts that only act as a front-end and delegate actual execution to another, updatable contract address, in most cases creating a new contract with updated code and transferring funds is the only option—given that funds can still be reclaimed from the old contract[2].

An example smart contract is given in Figure 1. This smart contract models a wallet, which allows to deposit and withdraw money (`deposit`, `withdraw`) as well as to transfer ownership of the wallet (`changeOwner`). In Solidity, a function with the same name as the contract is considered a constructor (`Wallet`). The constructor code is only executed once during contract creation and is not part of the contract code afterwards.

Furthermore, Solidity has the concept of *modifiers*. Modifiers are special functions with a placeholder (`_`) that allow to "wrap" other functions. Modifiers are often used to implement sanity or security checks. For instance, the example contract defines a modifier `onlyOwner`, which checks if the sender of the current transaction is equal to the stored owner of the wallet. Only if the check succeeds the actual function is executed. This is used to ensure that only the owner of the wallet can perform withdraw money or transfer ownership.

```
1  contract Wallet{
2    address owner;
3
4    // constructor
5    function Wallet(){
6      owner = msg.sender;
7    }
8
9    modifier onlyOwner{
10     require(msg.sender == owner);
11     _;
12   }
13
14   function changeOwner(address newOwner)
15   onlyOwner {
16     owner = newOwner;
17   }
18
19   function deposit()
20   payable {
21   }
22
23   function withdraw(uint amount)
24   onlyOwner {
25     owner.transfer(amount);
26   }
27 }
```

Figure 1: A contract that models a wallet.

### 2.2.3 Transactions

In Ethereum, all interactions between accounts happens through transactions. The most important fields of a transaction are `to`, `sender`, `value`, `data`, and `gas`. `sender` and `to` are the sender and receiver of a transaction respectively. In a normal transaction between two regular accounts, `value` denotes the amount to be transferred while `data` can be used as a payment reference. A simple function call on a smart contract on the other hand is a transaction with a `value` of 0 and `data` the input data to the contract. By convention, Solidity uses the first four bytes of `data` as a function identifier, followed by the function arguments. The function identifier is computed as the first four bytes of the Keccak-256 hash of the function's signature. E.g., to call the `withdraw` function, `data` would consist of the bytes `2e1a7d4d`, followed by the amount to be withdrawn in Wei as a 256-bit word. Functions marked as `payable`, as for example `deposit` in Figure 1 can also be invoked through transactions with a non-zero `value`. In this case, the transferred value is

[1] The original Ethereum paper [25] distinguishes between *transactions*, which are signed by regular accounts, and *messages*, which are not. For simplicity we will refer to both as transactions in this paper.

[2] `https://np.reddit.com/r/ethereum/comments/316b6b/fuck_i_just_send_all_my_ether_to_a_new_contract/`

added to the contract's balance.

The concept of "functions" and "modifiers" only exists at the level of Solidity, not at the bytecode-level of the EVM. At EVM level, a smart contract is just a single string of bytecode, and execution always starts at the first instruction. This is why compiled contracts usually start with a sequence of branches, each comparing the first four bytes of `data` to the contract's function signatures.

A transaction also specifies the transaction fee a miner gets for processing the transaction. To this end, Ethereum uses the concept of "gas": Every instruction that is executed by a miner in order to process the transaction consumes a certain amount of gas. Gas consumption depends on the instruction type to model the actual work performed by the miner. For example, a simple addition consumes 3 units of gas, whereas access to the contract's storage consumes 20000. The transaction field `gas` therefore specifies the maximum amount of gas that may be consumed in processing the transaction. When this limit is exceeded, processing of the transaction is aborted. However, the processing fee is still deducted.

## 2.3 Notation

Keeping our terminology close to the formal specification of Ethereum [25], we use the following notation: We use $\mu$ to denote an EVM machine state with memory $\mu_m$ and stack $\mu_s$. Furthermore, we use $I$ to refer to a transaction's execution environment, in particular, we use $I_d$ as the `data` field of the transaction and $I_v$ as its `value`. Finally, $S$ refers to a contract's storage.

## 2.4 Attacker Model

For the attacks considered in this paper we do not require special capabilities from the attacker. An attacker needs only be able to (i) obtain a contract's bytecode (in order to generate an exploit) and (ii) to submit transactions to the Ethereum network (to execute the exploit). The fact that both of these are trivial to accomplish serves to stress the severity of the attacks found by our tool TEETHER.

## 2.5 Ethical Considerations

On the one hand, we believe that raising awareness of critical vulnerabilities in smart contracts is fundamentally important to maintain the trust of their manifold users. Our methodology thus represents a step forward and allows users to check their contracts for critical flaws that may lead to financial loss. On the other hand, describing a detailed methodology, and in particular, releasing a tool to automatically find *and exploit* flaws in contracts may ask for abuse. Yet we argue this is the right way of going forward, as "security by obscurity" has proven ineffective since long. Furthermore, especially the fully automated creation of exploits allows to easily validate whether the found vulnerabilities are actually real—an important step to show the effectiveness and accuracy of any bug finding mechanism.

A fundamental downside of largely anonymous blockchain networks like Ethereum, however, is that we cannot reach out to owners of vulnerable contracts. This is in stark contrast to responsible disclosure processes in open-source software projects that have dedicated points of contact. For Ethereum accounts and contracts, such contacts do not exist. We discussed several approaches to tackle this problem, including but not limited to (i) public disclosure of all vulnerable accounts such that they can remedy the problem (yet revealing exactly to the public which contracts are vulnerable); (ii) temporarily transfer ("steal") money from vulnerable contracts into secure contracts until the owner has fixed the problem (yet rendering the old contract unavailable, causing money loss due to transaction fees, and being illegal). In the end, we deemed none of these options optimal, and decided to refrain from mentioning particular vulnerable contracts in public. If contract owners are in doubt and can prove their ownership to us, we will responsibly disclose the generated exploit to them. We aim to release TEETHER 180 days after publication of this paper, to give contract owners sufficient time fixing their contracts until others can easily reproduce our work by re-executing our tool.

# 3 Smart Contract Vulnerabilities

Smart contracts usually enforce certain control over who is allowed to interact with them. A particularly important guarantee is that contracts only allow "authorized" Ethereum accounts to receive coins that are stored in the contract. In this context, a contract is *vulnerable*, if it allows an attacker to transfer Ether from the contract to an attacker-controlled address. From such vulnerable contracts, an attacker can steal all (or at least parts of) the Ether stored in them, which can result in a total loss of value for the contract owner.

We now describe how one can identify such vulnerabilities in Ethereum contracts. Our idea is to statically analyze a contract's code to reveal critical code parts that might be abused to steal Ether stored in a contract. To this end, we describe how the aforementioned vulnerabilities map to EVM instructions.

## 3.1 Critical Instructions

We identify four critical EVM instructions, one of which must necessarily be executed in order to extract Ether from a contract. These four instructions fall into two categories: Two instructions cause a direct transfer, and two

instructions allow arbitrary Ethereum bytecode to be executed within a contract's context.

#### 3.1.1 Direct value transfer

Two of the EVM instructions described in Ethereum's formal specification [25] allow the transfer of value to a given address: `CALL` and `SELFDESTRUCT`.[3] The `CALL` instruction performs a regular transaction, with the following stack arguments:

1. `gas` – the amount of gas this transaction may consume
2. `to` – the beneficiary of the transaction
3. `value` – the number of Wei (i.e., $10^{-18}$ Ether) that will be transferred by this call

4.-7. `in offset`, `in size`, `out offset`, `out size` – memory location and length of call data respectively returned data.

Thus, if an attacker can control the second stack argument (`to`) when a `CALL` instruction is executed with a non-zero third stack argument, they can cause the contract to transfer value to an address under their control.

The `SELFDESTRUCT` instruction is used to terminate a contract. This will cause the contract to be deleted, allowing no further calls to this contract. `SELFDESTRUCT` takes a single argument – an address where all remaining funds of this contract will be transferred to. If an attacker can cause execution of a `SELFDESTRUCT` instruction while controlling the topmost stack element, he can obtain all the contract's funds as well as cause a permanent Denial-of-Service of this contract.

#### 3.1.2 Code injection

While `CALL` and `SELFDESTRUCT` are the only two instructions that allow an attacker to directly transfer funds from a contract to a given address, this does not imply that contracts lacking these two instructions are not vulnerable. In order to facilitate libraries and code-reuse, the EVM provides the `CALLCODE` and `DELEGATECALL` instructions, which allow the execution of third party code in the context of the current contract. `CALLCODE` closely resembles `CALL`, with the only exception that it does not perform a transaction to `to`, but rather to the current contract itself *as if it had the code of* `to`. I.e. the beneficiary of the new transaction remains the same, but it will be processed using `to`'s code. `DELEGATECALL` does the same, but persists the original values of `sender` and `value`, i.e., instead of creating a new internal transaction, it modifies the current transaction and "delegates" handling to another contract's code. Consequently, `value` is omitted from the arguments of `DELEGATECALL`.

[3] Additionally, the `CREATE` instruction allows to create a new contract and transfer value to it. However, this would require an attacker to have control over the resulting contract to receive the coins. Therefore, we will not consider `CREATE` for the remainder of this work.

```
1 PUSH20 <attacker-controlled address>
2 SELFDESTRUCT
```

Figure 2: EVM "shellcode"

If an attacker controls the second stack element (`to`) of either `CALLCODE` or `DELEGATECALL`, they can "inject" arbitrary code into a contract. By deploying the snippet from Figure 2 to a new contract, and subsequently issuing a `CALLCODE` or `DELEGATECALL` in the vulnerable contract to this new contract, the original contract can be destructed and all funds transferred to the attacker.

#### 3.1.3 Vulnerable State

Summarizing, this systematic analysis of the Ethereum instructions allows us to precisely define when a contract is in a vulnerable state:

**Definition 1** (Critical Path)**.** *A critical path is a potential execution trace that either*

1. *leads to the execution of a `CALL` instruction with a non-zero third stack element where the second stack argument can be externally controlled,*
2. *leads to the execution of a `SELFDESTRUCT` instruction where the first stack argument can be externally controlled, or*
3. *leads to the execution of either a `CALLCODE` or `DELEGATECALL` instruction where the second stack argument can be externally controlled.*

**Definition 2** (Vulnerable State)**.** *A contract is in a* vulnerable state*, if a transaction can lead to the execution of a critical path.*

We will call a transaction that exploits a contract in vulnerable state by one of the critical instructions as a *critical transaction*.

### 3.2 Storage

While it is obvious that a contract in vulnerable state is vulnerable according to our intuition that attackers can steal Ether, the converse it not necessarily true. Consider, for example, the contract given in Figure 3. As long as `vulnerable` is set to false, this contract is not in a vulnerable state, as the `transfer`-statement—and

```
1  contract Stateful{
2    bool vulnerable = false;
3    function makeVulnerable(){
4      vulnerable = true;
5    }
6    function exploit(address attacker){
7      require(vulnerable);
8      attacker.transfer(this.balance);
9    }
10 }
```

Figure 3: Stateful contract

its corresponding `CALL` instruction—cannot be reached due to the preceding `require`. Only after a call to `makeVulnerable` the `vulnerable` variable is set and a vulnerable state is reached. Yet, intuitively, this contract is vulnerable. We thus have to extend our definition to also include a notion of state that captures modifications made to a contract's storage.

The only instruction that allows to modify storage is `SSTORE`. A transaction that performs a storage modification therefore always executes a `SSTORE` instruction. We can therefore define state-changing transactions.

**Definition 3** (State Changing Path)**.** *A* state changing path *is a potential execution trace that contains at least one `SSTORE` instruction.*

**Definition 4** (State Changing Transaction)**.** *A transaction is* state changing *if its execution trace is a state changing path.*

Combining this with Definition 2 allows us to give the following definition

**Definition 5** (Vulnerable)**.** *A contract is* vulnerable *if there exists a (possibly empty) sequence of* state changing *transactions that lead to a* vulnerable state.

From this it immediately follows that a successful exploit always consists of a sequence of state changing transactions followed by a critical transaction.

## 4 Automatic Exploitation

In this section we present TEETHER, our tool for automatic exploit generation for smart contracts.

### 4.1 Overview

Figure 4 shows the overall architecture of TEETHER. In a first step, the CFG-recovery module disassembles the EVM bytecode and reconstructs a control flow graph (CFG). Next, this CFG is scanned for critical instructions

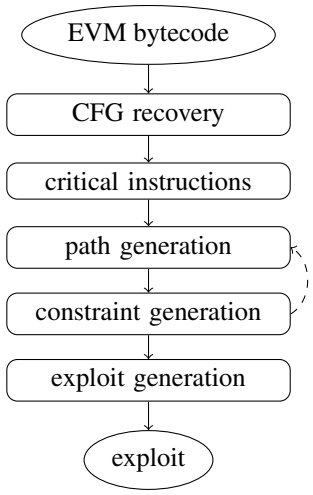


Figure 4: Architecture of TEETHER

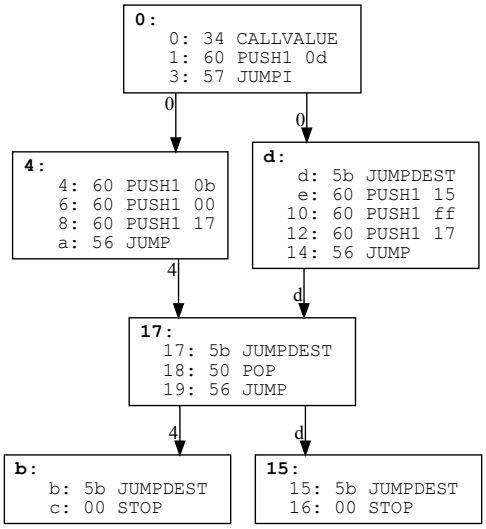


Figure 5: An example CFG with dependent edges

and for state changing instructions. The path generation module explores paths from the root of the CFG leading to these instructions, from which the constraint generation module creates a set of path constraints through symbolic execution. Finally, the exploit generation module solves the combined constraints of critical paths and state changing paths to produce an exploit.

### 4.2 CFG Recovery

Reconstructing a control flow graph from EVM bytecode is a non-trivial task. This is due to the fact that the EVM only provides control flow instructions with indirect jumps. Both the conditional `JUMPI` and the unconditional `JUMP` read the jump target from the top-most stack element. While the jump target can be trivially inferred

in some cases, such as **PUSH2** <addr>**; JUMP**, it becomes less obvious in other cases. For example, consider the JUMP instruction at address 19 in Figure 5. Here, the JUMP instruction is used similar to x86's `ret` instruction, to resume execution at a "return address" that the caller pushed on the stack before the function call.

To address this, TEETHER uses backward slicing to iteratively reconstruct the CFG. Initially, the CFG contains only trivial edges, i.e., those matching the above pattern as well as fall-through cases for JUMPI. All other JUMP and JUMPI instructions are marked as *unresolved*. Next, an unresolved JUMP or JUMPI is selected and the set of (path-sensitive) backward slices of its jump target is computed. If a full backward slice can be found, it is executed to compute the jump target, the newly found edge is added to the CFG, and the corresponding jump instruction marked as *resolved*. Since introduction of a new edge can lead to possibly new backward slices of jumps within the newly connected subtree, all JUMP and JUMPI instructions in this subtree are again marked as *unresolved*. This process is repeated until no new edges are found and all jump instructions marked as *resolved*.

In the example in Figure 5, two backward slices can be found for the JUMP instruction at address 19, (**PUSH1** 0b) and (**PUSH1** 15), which allows to introduce two out-edges for basic block 17, 17 $\rightarrow$ b and 17 $\rightarrow$ 15.

#### 4.2.1 Dependent edges

Another problem that arises from indirect jumps is the problem that a path in the CFG does not necessarily correspond to a valid execution trace. E.g. the path 0 $\rightarrow$ 4 $\rightarrow$ 17 $\rightarrow$ 15, while seemingly plausible from the CFG, can never occur in an actual execution, as the edge 17 $\rightarrow$ 15 can only be taken if 17 was entered from d.

To reduce the number of invalid paths considered in further analyses, TEETHER uses an approach we call *dependent edges*. For this, edges are annotated with a basic block-level summary of their backward slices. In a forwards exploration, a path may be extended by an edge iff one of its annotations is fully contained in the path. Referring to Figure 5, the path 0 $\rightarrow$ 4 $\rightarrow$ 17 may only be extended via 17 $\rightarrow$ b. Likewise, in a backwards exploration these annotations form a set of path requirements, restricting the exploration to subpaths that can still reach all required basic blocks. For example, a backwards analysis starting from 15 $\rightarrow$ 17 has collected the requirement set {b} and may not take the back-edge 17 $\rightarrow$ 4 as b is not an ancestor of 4.

### 4.3 Path Generation

The resulting CFG is then scanned for CALL, CALLCODE, DELEGATECALL, and SELFDESTRUCT instructions. For each found instruction, the set of backward slices of its critical argument is computed. As we require that this argument is potentially attacker controlled, slices are then filtered for those containing instructions whose results can be directly (ORIGIN, CALLER, CALLVALUE, CALLDATALOAD, CALLDATASIZE, CALLDATACOPY) or indirectly (SLOAD, MLOAD) controlled by an attacker.

Each of the remaining slices defines an instruction subsequence of a critical path. To find critical paths, TEETHER explores paths using A* [15], where the cost of a path is defined as the number of branches this path traverses in the CFG. As every branch in the CFG corresponds to an additional path constraint, this allows TEETHER to explore less-constrained paths first. This captures the intuition that a path with fewer constraints is easier to satisfy. To focus on critical paths only, after every step we check whether all remaining instructions of at least one critical slice can still be reached from the current path. If no critical slice can be reached in full, further exploration of the path is discarded.

State changing paths are found in a similar fashion by searching for SSTORE instructions. As a state change can be useful for an attacker even without controlling the address or value written (e.g., Figure 3), no backward slices need to be computed in this case. Thus, the A* search only has to check whether a SSTORE instruction can be reached on the current path.

### 4.4 Constraint Generation

The constraint generation module runs in lockstep with the path generation. Once a path is found, the path constraint generation module tries to execute the path symbolically in order to collect a set of path constraints. To this end, TEETHER uses a symbolic execution engine based on Z3 [13]. Fixed-size elements, such as the call value or the caller's address are modelled using fixed-size bitvector expressions, variable-length elements, such as the call data, the memory, and the storage are modeled using Z3's array expressions.

Whenever a conditional branch (JUMPI) is encountered whose condition $\mu_s[1]$ is a symbolic expression, both the jump target and the fall through target are compared to the next address defined by the given path, and a new constraint of the form $\mu_s[1] \neq 0$ respectively $\mu_s[1] = 0$ is added accordingly.

#### 4.4.1 Infeasible Paths

Not all paths generated by the path generation module necessarily correspond to feasible execution traces. Consider for example the code given in Figure 6. Here the path generation module will eventually output the path 1 $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 5 $\rightarrow$ 6. However, when executing this

```
1  int x = 0;
2  if(msg.value > 0){
3    x = 1;
4  }
5  if(x!=0){
6    msg.sender.transfer(this.balance);
7  }
```

Figure 6: Infeasible Paths Example

path, the value of x at line 5 will always be a concrete value and, since the path skipped the assignment in line 3, will have value 0. Thus the branch to line 6 will not be taken going directly to line 7 instead, leading to a mismatch between the program counter (7) and the next step of the intended path (6). Therefore, we consider a path *infeasible*, as soon as the program counter deviates from the desired path. To prevent expensive symbolic execution of further paths that would also be infeasible due to the same conditions, we extract a minimal infeasible subpath. As such deviations can only occur following a JUMP or JUMPI instruction, we consider the backward slices of the last executed instruction. These slices contain all instructions contributing to the jump target and in case of JUMPI also to the branch condition. The minimal infeasible subpath is then the subpath of the execution trace starting from the first instruction that is contained in any of the slices. In case a value loaded from memory or storage is contained in the path, the entire execution trace is taken as the minimal infeasible subpath, to keep the analysis sound. This minimal infeasible subpath is then passed back to the path generation module, which will stop exploring paths containing this subpath.

#### 4.4.2 Hash Computation

While symbolic translation of most EVM instructions is relatively straight-forward, special care has to be taken to symbolically model the EVM's SHA3 instruction. The SHA3 instruction takes a memory region as input (specified through two arguments, address and size) and computes the Keccak-256 hash over the memory contents stored therein. This instruction is, for example, used by the Solidity compiler for the mapping data structure, which provides a key-value store. Accessing a value stored in a mapping is commonly implemented by computing the Keccak-256 hash of the key and using the resulting value as an index into the contract's storage. Since such mappings are a common data structure in Ethereum contracts, TEETHER needs to be able to reason about such storage accesses, which requires a symbolic modeling of the SHA3 instruction.

To this end, whenever we want to symbolically execute a SHA3 instruction, we introduce a new symbolic 256-bit variable to model the result of the Keccak-256 computation. At the same time we record the relation between this new variables and the input data given to the SHA3 instruction. We will later show in Section 4.5.1 how this mapping can be used to solve path constraints which include hash-dependent constraints.

#### 4.4.3 Symbolic-Length Memory Access

Another issue of symbolic execution is that some EVM instructions can copy to/from variable-length elements. For example, the SHA3 instruction can compute hashes over variable length data. Similarly, the CALLDATACOPY instruction, which copies bytes from the given call data into memory, operates on variable-length data. This makes symbolic execution non-trivial, as the length is not a concrete value but a symbolic expression instead. TEETHER uses two approaches to address these issues.

First, whenever data of symbolic length is copied to memory, e.g., when using CALLDATACOPY, we use Z3's If expression to model conditional assignments. For example, a common pattern seen in smart contracts is copying the entire input data into memory. TEETHER will execute this using assignments of the form

$$\mu'_m[a+i] \leftarrow \mathrm{If}(i < l, I_d[b+i], \mu_m[a+i])$$

where $a = \mu_s[0]$, $b = \mu_s[1]$, and $l = \mu_s[2]$. To keep the number of generated expressions reasonable, we perform assignments only up to a pre-configured upper limit for $i$ (256 in our experiments).

Second, if data of symbolic length is read from memory, we will return a new symbolic read object. Similarly to mapping of Keccak-256 results to their respective input data, we also keep a mapping from symbolic read objects to their address, their length, and the memory-state when the read occurs. This allows us to later resolve the value of a symbolic read object.

#### 4.4.4 Constraint Results

The final output of the constraint generation module for a given path $p$ is a tuple $R^p = (\mu, S, I, C, H, M)$, where

- $\mu$ is the symbolic machine state after execution
- $S$ is the symbolic storage of the contract after execution
- $I$ is the symbolic environment in which path $p$ is executed
- $C$ is a set of constraints that must be fulfilled to execute path $p$

$H$ is a mapping of Keccak-256 result variables to their respective input data

$M$ is a mapping of symbolic read objects to their address, length, and memory state

We assume that both $\mu$ and $S$ also capture the entire history of the respective states after every instruction.

As discussed, sometimes it will be necessary to perform a sequence of multiple state changing transactions followed by an exploiting transaction. For this we define the combined constraint result given by a path sequence $\vec{v} = p_0, \ldots, p_n$ as $R^{\vec{v}}$. Let $\mu_0$ and $S_0$ denote the initial state of a path, then $R^{\vec{v}} = (\vec{\mu}, \vec{S}, \vec{I}, C, H, M)$, where

$$\vec{\mu} = \mu^{p_0}, \ldots, \mu^{p_n}$$
$$\vec{S} = S^{p_0}, \ldots, S^{p_n}$$
$$\vec{I} = I^{p_0}, \ldots, I^{p_n}$$
$$C = \bigcup_{i=0}^{n} C^{p_i} \cup \bigcup_{i=0}^{n-1} \{S_0^{p_{i+1}} = S^{p_i}\} \cup \{S_0^{p_0} = \hat{S}\}$$
$$H = \bigcup_{i=0}^{n} H^{p_i}$$
$$M = \bigcup_{i=0}^{n} M^{p_i}$$

Note the introduction of additional constraints $S_0^{p_{i+1}} = S^{p_i}$ and $S_0^{p_0} = \hat{S}$, which encode that the state changes performed by path $p_i$, are still present at the beginning of path $p_{i+1}$. Storage at the beginning of the first path $p_0$ is equal to the last state $\hat{S}$ stored in the blockchain. We will use the notation $\mu^*$ and $S^*$ to refer to the symbolic machine state and storage *just* before execution of the critical instruction in the final path.

In order to only create meaningful path combinations, we only prepend a state changing path $p$ to a path sequence $\vec{v}$, if any of the paths in $\vec{v}$ may read from storage entries modified by $p$. To this end, TEETHER also records every storage accesses that is performed during symbolic execution of a path. A path sequence $\vec{v}$ may read from the storage modifications made by a path $p$ iff there exists a write to address $e$ in $p$ and a read from address $f$ in $\vec{v}$ such that either

1. Both $e$ and $f$ are concrete values and $e = f$
2. At least one of $e$ and $f$ is a symbolic expression, and neither depend on a Keccak-256 result
3. Both $e$ and $f$ are symbolic expression dependent on Keccak-256 results $h_e$ and $h_f$ respectively, both are structurally identical, i.e., have an identical AST, and the hash results could potentially be equal, i.e., their input data has at least the same length, $\|H^p[h_e]\| = \left\|H^{\vec{v}}[h_f]\right\|$.

TEETHER tries to create an exploit based on a single path first, before trying larger path sequences. For our experiments, we explored path sequences up to length three, consisting of at most two state-changing paths and one final critical path.

## 4.5 Exploit Generation

The final stage of TEETHER is the exploit generation module, which checks the combined path constraints generated in the previous step for satisfiability with respect to Keccak-256 results and symbolic read objects. If a path sequence with satisfiable combined path constraints is found, this module will output a list of transactions that lead to exploitation of the smart contract. Otherwise, the next path sequence is requested and tested.

Before checking satisfiability of a combined result, we first encode the attacker's goals using additional constraints. The first goal is to transfer funds or code execution to an attacker-controlled address $a$. We achieve this by adding a constraint $\mu_s^*[1] = a$ (`CALL`, `CALLCODE`, `DELEGATECALL`) or $\mu_s^*[0] = a$ (`SELFDESTRUCT`). The second goal of the attacker is to make profit. While this is not an issue in the cases of `CALLCODE`, `DELEGATECALL`, and `SELFDESTRUCT`, as here all funds of the smart contract can be transferred to the attacker, additional checks are needed in case of a `CALL`-based exploit. This is especially true since some of the necessary transactions might require transferring Ether *to* the contract first. We thus require that the value transmitted in the final `CALL` instruction is greater than the sum of all values sent to the contract. As value is specified by the third stack argument to `CALL`, formally this gives

$$\mu_s^*[2] > \sum_{i=0}^{n} I_v^{p_i}$$

### 4.5.1 Satisfying Assignment

Having assembled the combined path constraints of a path sequence, including their state inter-dependencies and the attacker's goals, the next step is to find a satisfying assignment, which will give us concrete values to build the transactions required for successful exploitation. We leverage the constraint solver Z3. Yet we cannot simply pass our set of collected constraints *as is*, as the constraint solver is unaware of the special semantics of Keccak-256 results and symbolic-read objects.

To overcome this problem we apply the iterative approach shown in Figure 7. The algorithm keeps a set $Q$ of *unresolved* variables, which is initially set to all elements of $H$ and $M$. As long as this queue is non-empty, we compute the subset $D$ of constraints that is not dependent on any of the variables in $Q$ and use a constraint solver to

$Q \leftarrow H \cup M$
$\mathbb{A} \leftarrow \emptyset$
**while** $\|Q\| > 0$ **do**
  $D \leftarrow \{c \in C | \text{Vars}(c) \cap Q = \emptyset\}$
  $\mathbb{A} \leftarrow \text{Sat}(D)$
  **for all** $x \in Q$ **do**
    **if** $x \in H$ **then**
      $e \leftarrow H[x]$
      **if** $e \cap Q = \emptyset$ **then**
        $v_e \leftarrow \mathbb{A}(e)$
        $v_x \leftarrow \text{Keccak-256}(v_e)$
        $C \leftarrow C \cup \{e = v_e, x = v_x\}$
        $Q \leftarrow Q \setminus \{x\}$
      **end if**
    **else if** $x \in M$ **then**
      $a, l, \mu_m \leftarrow M[x]$
      **if** $(\text{Vars}(a) \cup \text{Vars}(l)) \cap Q = \emptyset$ **then**
        $v_a \leftarrow \mathbb{A}(a)$
        $v_l \leftarrow \mathbb{A}(l)$
        $v_x \leftarrow \mathbb{A}(\mu_m[v_a : v_a + v_l])$
        $C \leftarrow C \cup \{a = v_a, l = v_l, x = v_x\}$
        $Q \leftarrow Q \setminus \{x\}$
      **end if**
    **end if**
  **end for**
**end while**
**return** $\text{Sat}(C)$

Figure 7: Iterative Constraint Solving Algorithm

find a satisfying variable assignment $\mathbb{A}$ for $D$. Next, the algorithm attempts to resolve unresolved variables from $Q$. A variable can be resolved, if it does not depend on other unresolved variables. To resolve a Keccak-256 result, we first evaluate the hash's input data expression (according to $H$) in the assignment $\mathbb{A}$. This gives us a concrete value for the input data, over which we can then compute a Keccak-256 hash. To "fix" this relation between Keccak-256 result variable and input data, we add two new constraints that bind the input-data to its current valuation and the Keccak-256 result variable to the computed hash value. A symbolic-memory read object is resolved similarly by computing concrete value for the start address and length. Once a variable has been resolved, it is removed from $Q$. This process is repeated until all variables are resolved.

The key insight here is that, since the mappings $H$ and $M$ define dependencies between the elements of $H$ and $M$ and the variables involved in their corresponding expressions, they also implicitly define a topological ordering on $H$ and $M$. Furthermore, as these mappings can never define a cycle, this ordering is well-defined.

Consider, for example, the Solidity statement `sha3(sha3(msg.sender))` which takes the address of the message sender and hashes it twice. This will lead to two entries in $H$, $h_0$ and $h_1$ with $H[h_0] = I_s$ and $H[h_1] = h_0$, which gives the dependency chain $h_1 \rightarrow h_0 \rightarrow I_s$. This means we first have to fix the value of $I_s$ to compute $h_0$, which will then allow us to compute $h_1$.

#### 4.5.2 Exploiting Transactions

If a satisfying assignment $\mathbb{A}$ can be found, TEETHER will then output a list of transactions $t_0, \ldots, t_n$ an attacker would have to perform in order to exploit the contract. Transaction value and data content for each transaction $t_i$ are given by

$$\texttt{value}_i = \mathbb{A}(I_v)$$
$$\texttt{data}_i = \mathbb{A}(I_d)$$

### 4.6 Implementation

TEETHER is implemented in 4,300 lines of Python, using Z3 [13] as constraint solver. We will release TEETHER as open source 180 days after paper publication.

## 5 Evaluation

To demonstrate the utility of TEETHER, we downloaded a snapshot of the Ethereum blockchain and scanned it for contracts. Using a snapshot from Nov 30 2017, we found a total of 784,344 contracts. Interestingly, many contracts share the same bytecode, with the most popular code being shared by 247,654 contracts. On the other hand, 32,401 contracts were only deployed on a single address. Removing duplicates left us with a total number of 38,757 unique contracts. We executed TEETHER on all these 38,757 contracts. To avoid the situation that our code analysis gets stuck too long in a single contract, we allowed up to 30 minutes for CFG reconstruction plus 30 minutes for finding each a `CALL`, `CALLCODE`, `DELEGATECALL`, and `SELFDESTRUCT`-based exploit. We furthermore assumed a contract's storage was empty at the beginning, such that we can treat duplicate contracts the same. All experiments were performed on a virtualized Intel Xeon E5-2660 system with 16 threads and 192 GB of memory, however, we never observed a memory usage of more than 32 GB.

### 5.1 Results

For 33,195 (85.65%) contracts, the analysis finished within the given time limit. Out of these, TEETHER was able to generate an exploit for 815 (2.10%), which we will analyze in detail below. To put this into perspective, about two thirds of all contracts, 24,331 or

| | CALL | CALLCODE | DELEGATECALL | SELFDESTRUCT | Contracts |
|---|---|---|---|---|---|
| exploit | 547 | 2 | 8 | 298 | 815 |
| independent | 413 | 2 | 8 | 241 | 630 |
| dependent | 134 | 0 | 0 | 57 | 189 |
| critical path | 7,039 | 6 | 60 | 2,357 | 8,049 |
| no critical path | 25,689 | 37,826 | 37,748 | 34,533 | 24,331 |
| Sum | 33,275 | 37,834 | 37,816 | 37,188 | 33,195 |

Table 1: Detailed exploit generation results

62.78%, do not even expose a single critical path. In other words, these contracts either do not contain any CALL, CALLCODE, DELEGATECALL, or SELFDESTRUCT instructions, or do so only with non-attacker controllable arguments. Further 8,049 (20.77%) contracts did have a critical path, but we were not able to exploit it. While some of these can be false negatives due to TEETHER's limitations, like the restricting the transaction sequences to maximum three, or limitations of the underlying constraint solver, we believe the majority of these cases are actually true negatives, as our definition of critical paths is broad. We will discuss this issue in detail in Section 6.

Table 1 shows a breakdown of analysis results per vulnerability type. While many contracts were found vulnerable to CALL- or SELFDESTRUCT-based exploits, only a small number of CALLCODE- and DELEGATECALL-based exploits were found. However, also the number of contracts having a critical CALLCODE or DELEGATECALL path is significantly lower compared with CALL or SELFDESTRUCT. Interestingly, some contracts exposed multiple vulnerabilities so that TEETHER generated a total of 855 exploits targeting 815 different contracts.

The 855 exploits can be grouped into two classes: As the target contract can send further transactions to other, third-party contracts during execution, the outcome of an exploit might be dependent upon the results returned by these transactions. We will call such exploits *dependent*. In contrast, in an *independent* exploit, the execution of the target contract does not depend on further transactions to non-attacker-controlled addresses. 134 (24.50%) of the 547 CALL-based exploits and 57 (19.13%) of the 298 SELFDESTRUCT-based exploits are dependent, leaving 413 respectively 241 independent exploits. As TEETHER can only create path constraints for a single contract, we will only consider independent exploits in the following.

As said before, many contract addresses share the same contract code. Therefore, while the 664 independent exploits only target 630 *different* contracts, in total, 1,731 contract accounts are affected.

## 5.2 Validation

To verify that the exploits generated by TEETHER do in fact work as intended, for ethical and jurisdictional reasons we refrain from testing them on the actual blockchain. While there are no technical limitations to buying Ether and performing the attacks on the main network of Ethereum, we chose to evaluate the generated exploits on private test networks only. We thus modeled an attack on the actual blockchain as close as possible.

Since every contract account has its own storage that can influence the execution, we validate every exploit against every affected account individually, leading to a total of 1,769 $(exploit, account)$ combinations. To this end, we create a fresh test Ethereum network (i.e., a separate blockchain) containing three accounts: The contract under test, a regular account to model the attacker, and a third contract whose code will be used in CALLCODE and DELEGATECALL exploits. The attacker's account and the contract account are given an initial balance of 100 and 10 Ether, respectively. Additionally, we also ensure that the contract's storage content in our test network agrees with the one from our snapshot of the actual Ethereum blockchain. The network is then run using the unmodified official Ethereum Go client [5], whose scripting interface will also be used to submit the exploit transactions.

To reduce computation time by allowing tests on several non-unique contracts at once, we computed the exploit assuming that the contract's storage was set to zero. The first step in evaluation is thus to repeat TEETHER's constraint and exploit generation stages by supplying the contract's actual storage content. Unfortunately, creating an updated exploit fails for 84 (5.71%) of the CALL-based and 28 (9.69%) of the SELFDESTRUCT-based exploits, which means that the generated exploit was a false positive. Note that while the analysis performed by TEETHER is sound in general, this assumption is the only thing breaking soundness in our evaluation. We further discuss this issue in Section 6

If generation of the updated exploit succeeded, we submit its transaction to our test network. To prevent transaction reordering, we wait until the miner processed each transaction before submitting the next. After the

| | CALL | CALLCODE | DELEGATECALL | SELFDESTRUCT | Total | |
|---|---|---|---|---|---|---|
| successful exploit | 1,301 | 1 | 7 | 255 | 1,564 | (88.41%) |
| failed exploit | 85 | 1 | 1 | 6 | 93 | (5.26%) |
| failed update | 84 | 0 | 0 | 28 | 112 | (6.33%) |
| Sum | 1,470 | 2 | 8 | 289 | 1,769 | (100.00%) |

Table 2: Validation results

last transaction has been processed, we check the final balance of the attacker's account. As the attacker's goal is to extract Ether from the target account we call the exploit successful if the final balance is greater than the 100 Ether that we preallocated to it. In order to minimize interference due to processing fees we set the gas price in our test network to 0, i.e., no processing fee is deducted.

The results for all tested 1,769 exploits are given in Table 2. Overall, a large fraction (88.41%) of the generated exploits works as expected: Once all exploit transactions have been processed, the attacker has successfully stolen Ether and increased their own balance.

Overall, 205 exploits (11.59%) failed for mainly two reasons. As mentioned earlier, 112 (6.33%) of all exploits failed in the update stage due to the mismatch in storage between the initial exploit generation and the exploit re-computation on the actual storage contents. To better understand why the exploit did not succeed in the remaining 93 cases, we further analyzed the constraints they induce. About half of these can be attributed to differences between our test network and the actual blockchain. For example, some of these exploits result in constraints based on the current block number or the balance of another account. As we base our test network on a custom genesis block, the current block number will be low when executing the contract, whereas the actual Ethereum blockchain has been constantly growing since 2015 and currently contains over 5,000,000 blocks. Similarly, as our test network only contains three accounts, retrieving another account's balance will always return 0, as these accounts do not exist in our network.

## 5.3 Case Studies

In an effort to shed some light onto the cause of these vulnerabilities, we manually reviewed all vulnerable contracts for which users had uploaded Solidity source code to `etherscan.io`. However, as this was the case for only 44 (3%) contracts, these findings do not provide a comprehensive list of contract vulnerabilities, but rather serve as a case-study. Finally, to protect contracts that are still "live", we only provide a description of the vulnerabilities we found, but do not publish addresses of vulnerable contracts.

Vulnerabilities we found in these contracts can be classified into four categories:

1. Erroneous visibility: Per default, Solidity functions are publicly accessible, unless marked with the keyword `internal`. This can lead to unintended exposure of contract functionalities. For example, one of the 44 contracts implements a betting functionality with a dedicated function to handle a draw. However, this function is not marked as `internal` and can be called directly to transfer funds to arbitrary addresses.

2. Erroneous constructor: In Solidity, a function with the same name as the contract itself serves as the contract's constructor. In contrast to regular functions, the constructor does not become part of the contract's compiled code and is only executed once during contract creation. However, as Solidity does not provide a special keyword to mark the constructor, functions that were meant to be constructors can become regular functions due to ignoring case-sensitivity, spelling mistakes, or oversight during refactoring operations such as renaming. The analyzed contracts contain examples of both, simple mistakes (e.g. Figure 8) and cases where the contract was presumably renamed without renaming the constructor (e.g. contract `MyContract_v1` with constructor `MyContract`).

3. Semantic confusion: Another class of vulnerable contracts stem from different misunderstandings of Ethereum's execution model. For example, these contracts seemingly confuse the contract's total balance (`this.balance`) with the value held by the current transaction (`msg.value`). Other cases neglect the fact that a contract's storage is publicly readable and thus should not be used to store secrets.

4. Logic flaws: The final class of vulnerabilities we observed is caused by logic flaws. For example, the excerpt given in Figure 9 is a flawed implementation of the classical `onlyOwner` modifier, but has an inverted condition. Contrary to the intended behaviour, this allows all marked functions to be called by anyone *but* the owner.

Interestingly, the first three of these categories can be almost exclusively attributed to Solidity. While vulnera-

```
1  contract Owned {
2      address public owner;
3      function owned() {
4          owner = msg.sender;
5      }
6      modifier onlyOwner {
7          if (msg.sender != owner) throw;
8          _;
9      }
10     //...
11 }
```

Figure 8: Erroneous constructor

```
1  modifier  onlyOwner() {
2      require(msg.sender != owner);
3      _;
4  }
```

Figure 9: Flawed `onlyOwner` modifier

bilities due to logic flaws are also common in other domains, others could be prevented through modifications of Solidity. For example, making functions **`internal`** *by default* would eliminate the first category. Likewise, the second category could be eliminated by introducing a dedicated keyword for constructors.

# 6 Discussion

While the evaluation results are promising and our tool has identified several hundreds of vulnerable contracts, there are cases in which our current implementation fails to create working exploits. In this section we discuss some of the underlying assumptions and limitations, both of TEETHER and of the evaluation we performed.

## 6.1 Critical Path Definition

One potential limitation of TEETHER is the broad definition of a critical path, specifically of potentially attacker-controlled instructions. Our definition states that a critical path is a path that contains a slice of a critical instruction which contains at least one potentially attacker-controlled instruction (cf. Definition 1). The inclusion of `SLOAD` and `MLOAD` into the potentially attacker-controlled instructions makes this criterion apply to many paths, even though the corresponding storage or memory locations may never be writable by an attacker. This, in turn, may cause irrelevant paths to be considered in the path generation. While this does not pose a conceptual problem, it can cause a significant increase in computation time and thus lead to a larger number of timeouts. This problem could be alleviated by performing additional checks to match `SLOAD` and `MLOAD` to previous writes to create a more precise definition of critical paths, thereby limiting the number of paths considered.

## 6.2 Inter-Contract Exploits

Furthermore, our current implementation of TEETHER focuses on intra-contract exploits. In fact, however, a contract may call other contracts, and by supporting this inter-contract communication one could find additional exploits. For example, the bug in Parity's multi-signature wallet [6] that allowed an attacker to take over multiple wallets, splits core functionality between two contracts. Whereas one contract acts as the actual Wallet, the other is the support library. Only by combining these two contracts TEETHER could find an exploit of this documented vulnerability. In fact, with all relevant code in a single contract, our tool can indeed find the vulnerability and create a working exploit (see Appendix A).

## 6.3 Evaluation

As described in Section 5, our evaluation initializes the contract's storage to an empty state when we start searching for exploits. This allows us to combine the analysis of contracts that share the same code, reducing the number of tool runs from 784,344 to only 38,757 and has reduced the overall runtime by roughly factor 20. However, this comes at the cost of imprecise results. As we already have observed in 112 cases, an exploit that would work against a contract with empty storage might not work against the same contract with filled storage. Conversely, our current evaluation might also *miss* exploits that only work if the storage contains certain entries. However, this is not a fundamental limitation of TEETHER and can be solved by retrieving the actual storage state from the real Ethereum blockchain, and reapplying it to our local testbed. While it would require to treat all collapsed non-unique contracts separately, as each address has its own storage state, the results obtained would be sound.

# 7 Related Work

In this section we discuss related work in the areas of smart contract analysis and automatic exploitation, and how they relate to the work presented here.

## 7.1 Smart Contract Analysis

Analysis of smart contracts has been an area of active research for the past few years. In a similar vein to the work present herein, Luu et al. [20] presented OYENTE,

a tool to detect certain vulnerabilities like transaction-ordering dependence or reentrancy. However, their work is substantially different from ours in two ways: Firstly, OYENTE only considers a very specific set of vulnerabilities, many of which can also only be exploited by a malicious miner or a by colluding with a miner colluding. In contrast, we give a general vulnerability definition that can be exploited by a much weaker attacker—in fact, anyone with an Ethereum account. Secondly, the goal of OYENTE is only to *detect* a vulnerability. This means that the report generated by OYENTE have to be painstakingly verified on a case-by-case basis. Our tool, on the other hand, is designed to automatically provide an exploit once a vulnerability is found. Validation is then often as easy as executing the exploit transactions and checking the final balance.

Atzei et al. [7] provide a survey on attacks against Ethereum smart contracts, giving a taxonomy and discussing attacks and flaws that have been observed in the wild. While not all attacks they consider provide a monetary benefit to the attacker, some of the attacks presented therein are a special case of the vulnerabilities considered by TEETHER. For example, the multiplayer games attack described in their paper can also be identified and be exploited by our tool—fully automated.

In an effort to support further vulnerability analyses, Matt Suiche has proposed a decompiler [24]. Also, Zhou et al. [26] developed Erays, a tool for reverse engineering smart contracts able to produce high-level pseudocode from compiled EVM code. Yet in contrast to our work, both of these rely on manual contract inspection (although at a higher code abstraction).

Aside from security vulnerabilities, Delmolino et al. [19] describe several pitfalls that can lead to logic flaws in smart contracts. In a similar vein, several works consider the problem of designing good contracts, e.g. Mavridou et al. [21] or Chen et al. [12].

Fröwis and Böhme [14] performed an analysis on trust-dependencies between contracts, revealing that contracts oftentimes rely on further external contracts. This also implies that a vulnerable contract may put other, dependent contracts at risk.

Complementary to vulnerability detection there have also been advances towards verification of smart contracts. The work by Bhargavan et al. [9] presents EVM* and Solidity* that provide a direct translation of a subset of EVM bytecode and Solidity into F* respectively, which can then be used for further verification.

ZEUS, recently presented by Kalra et al. [18], provides a framework to check smart contracts written in Solidity against a user-defined policy. Both contract source code and policy are compiled together into an LLVM-based intermediate representation, which is then further analysed statically, leveraging existing LLVM-IR-based verification tools. Based on this, they analyze 1,524 Ethereum contracts for policy violations against a list of known bugs (including the ones considered by OYENTE). Additionally, they also use ZEUS to check a subset of contracts against contract-specific fairness properties.

Like OYENTE, ZEUS also requires access to a contract's source code, whereas our tool works given only compiled EVM bytecode. Furthermore, in contrast to our tool, ZEUS requires user-interaction to define a policy, which is often contract specific. Finally, a policy violation found by ZEUS does not imply practical exploitability of the contract in question, whereas our tool outputs exploits that can be easily validated.

Finally, Breidenbach et al. [10] proposed using bug bounties to incentivize security analyses of smart contracts. Specifically, they designed a framework that encodes the process of identifying exploits and paying rewards into a smart contract itself, thereby guaranteeing fairness between the bounty payer and the bug finder.

## 7.2 Automatic Exploitation

Another area that is related to our work is the research field of automatic exploitation. Many tools have been proposed that can create specific classes of exploits under certain conditions. Notable examples are: Q, presented by Schwartz et al. [23], can transform a x86 software exploit into another exploit that still works under harder constraints (e.g., Address Space Layout Randomization and W^X). AEG by Avgerinos et al. [8] and MAYHEM by Cha et al. [11] both provide means to create a control flow hijacking exploit using buffer overflows or format string attacks from source code and compiled binaries, respectively. Huang et al. [17] extends the considered attack surface by including the operating system and libraries a compiled binary uses at runtime, and work by Hu et al. [16] considers non-control-flow hijacking exploits by modelling data-oriented exploits.

While all of these share the general idea of symbolic execution, constraint generation, and resolution to generate an exploit—as does the work presented herein—there are major differences. The most obvious difference is that the execution environment of the EVM does not provide an equivalent to buffer overflows or format string exploits. As such, the considered exploits are substantially different. Furthermore, all works mentioned rely on preconditioning, i.e., providing a starting point to the path exploration, most often in the form of a crashing input. In contrast to this, our work can create an exploit only based in the compiled contract's code without further input. Finally, there are also challenges specific to the EVM that do not apply to previous work, primarily handling and resolution of hash-values, which are an integral part of many smart contracts.

## 8 Conclusion

We have presented a generic definition of vulnerable contracts and a methodology for automatic exploit generation based on this definition. In a large-scale analysis encompassing 38,757 contracts from the Ethereum blockchain, teEther identified 815 as vulnerable. Furthermore, teEther successfully generated 1,564 working exploits against Ethereum accounts that use these contracts. This illustrates that smart contract security should be taken seriously, especially as these exploits are fully anonymous and trivial to conduct—they only require an Ethereum account. Exploit generation, as we have shown, can be fully automated.

Over the last years, Ethereum has seen a rapid and steady increase in value. Should this trend continue into the future, smart contract exploitation will only become more lucrative, and in turn, seeking protection will become even more important. Our methodology and especially concrete tools such as teEther can help in finding, understanding, and preventing exploits *before* they cause losses. Finally, our systematic analysis of the real Ethereum blockchain has revealed that the problem of highly-critical vulnerabilities in smart contracts is way larger than anecdotal evidence might suggest.

## Acknowledgements

We would like to thank the anonymous reviewers for their comments. Furthermore, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 700176 ("SISSDEN").

## References

[1] `https://coinmarketcap.com`. Accessed Feb 1st, 2018.

[2] `https://soliditylang.com/documentation/language-specifications.html`. Accessed Feb 1st, 2018.

[3] `https://blog.consensusx.com/dissecting-the-two-malicious-ethereum-messages-that-cost-30m-but-couldve-cost-100m-155e023a9500`. Accessed Feb 1st, 2018.

[4] `https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html`. Accessed Feb 1st, 2018.

[5] `https://github.com/ethereum/go-ethereum`. Accessed Feb 1st, 2018.

[6] `https://paritytech.io/the-multi-sig-hack-a-postmortem/`. Accessed Feb 1st, 2018.

[7] Atzei, N., Bartoletti, M., and Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust (POST'17)* (2017).

[8] Avgerinos, T., Cha, S. K., Hao, B. L. T., and Brumley, D. Automatic exploit generation. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS'11)*.

[9] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 11th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS'16)* (2016).

[10] Breindenbach, L., Daian, P., Tramèr, F., and Juels, A. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)* (2018).

[11] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)* (2012).

[12] Chen, T., Li, X., Luo, X., and Zhang, X. Under-optimized smart contracts devour your money. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'17)* (2017).

[13] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)* (2008).

[14] Fröwis, M., and Böhme, R. In code we trust? In *Proceedings of the First International Workshop on Cryptocurrencies and Blockchain Technology (CBT'17)* (2017).

[15] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (1968).

[16] Hu, H., Chua, Z. L., Adrian, S., Saxena, P., and Liang, Z. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)* (2015).

[17] Huang, S.-K., Huang, M.-H., Huang, P.-Y., Lu, H.-L., and Lai, C.-W. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability 63*, 1 (2014).

[18] Kalra, S., Goel, S., Dhawan, M., and Sharma, S. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS'18)* (2018).

[19] Kevin Delmolino, Mitchell Arnett, A. K. A. M., and Shi, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. Cryptology ePrint Archive, Report 2015/460, 2015. `https://eprint.iacr.org/2015/460`.

[20] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making Smart Contracts Smarter. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (2016).

[21] Mavridou, A., and Laszka, A. Designing secure ethereum smart contracts: A finite state machine based approach. *arXiv preprint arXiv:1711.09327* (2017).

[22] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`, 2009.

[23] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security'11)* (2011).

[24] Suiche, M. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF CON 25* (2017).

[25] Wood, G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. `http://gavwood.com/Paper.pdf`, 2014.

[26] Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A., and Bailey, M. Erays: Reverse engineering ethereum's opaque smart contracts. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)* (2018).

```
contract MultiOwned{
  uint public m_numOwners;
  uint public m_required;
  uint[256] m_owners;
  mapping(uint => uint) m_ownerIndex;
  mapping(bytes32 => PendingState) m_pending;
  bytes32[] m_pendingIndex;
  struct PendingState { uint yetNeeded; uint ownersDone; uint index; }
  modifier onlymanyowners(bytes32 _operation) {
    if (confirmAndCheck(_operation)) _;
  }
  function confirmAndCheck(bytes32 _operation) internal returns (bool) {
    uint ownerIndex = m_ownerIndex[uint(msg.sender)];
    if (ownerIndex == 0) return;
    var pending = m_pending[_operation];
    if (pending.yetNeeded == 0) {
      pending.yetNeeded = m_required;
      pending.ownersDone = 0;
      pending.index = m_pendingIndex.length++;
      m_pendingIndex[pending.index] = _operation;
    }
    uint ownerIndexBit = 2**ownerIndex;
    if (pending.ownersDone & ownerIndexBit == 0) {
      if (pending.yetNeeded <= 1) {
        delete m_pendingIndex[m_pending[_operation].index];
        delete m_pending[_operation];
        return true;
      }else{
        pending.yetNeeded--;
        pending.ownersDone |= ownerIndexBit;
      }
    }
  }
  function initMultiowned(address[] _owners, uint _required) {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
      m_owners[2 + i] = uint(_owners[i]);
      m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
  }
  function pay(address to, uint amount) onlymanyowners(sha3(msg.data)){
    to.transfer(amount);
  }
}
```

Figure 10: Minimal example of the Parity-Wallet Bug

## A Parity-Wallet Bug

Figure 10 shows a minimal working example of the Parity-Wallet Bug in a single contract. Lines 1-44 are taken verbatim from the original Parity wallet[4].

We ran TEETHER on this contract with the goal to produce an exploit transferring 1 Ether from the contract (address `0x400...000`) to the attacker (address `0x012...567`). TEETHER produces the following exploit in 26.74 seconds:

```
------------------------------------------------
Transaction 1
------------------------------------------------
from:  0x0123456789abcdef0123456789abcdef01234567
to:    0x4000000000000000000000000000000000000000
data:  c57c 5f60 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000
value: 0

------------------------------------------------
Transaction 2
------------------------------------------------
from:  0x0123456789abcdef0123456789abcdef01234567
to:    0x4000000000000000000000000000000000000000
data:  c407 6876 0000 0000 0000 0000 0000 0000
       0123 4567 89ab cdef 0123 4567 89ab cdef
       0123 4567 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0de0 b6b3
       a764 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
value: 0
```

The first transaction of this exploit calls function `initMultiowned` (`c57c5f60`) with all-zeros as arguments, i.e., an empty `_owners`-array and 0 as `_required`. This function will re-initialize the contract's owner information, setting `m_numOwners` to 1 and adding **`msg.sender`**, the attacker, to `m_owners[]` as the sole owner.

The second transaction then calls `pay` (`c4076876`), with the attacker's address (`0x012...567`) as `to` and $10^{18} = 0$xde0b6b3a7640000 (1 Ether in Wei) as `amount`. As the attacker has been set as the sole owner by the previous transaction, the function `confirmAndCheck` called by the `onlymanyowners` modifier will return **`true`**, allowing the function to proceed and leading to the transfer of 1 Ether to the attacker.

[4] https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol#L284