EthBMC: A Bounded Model Checker for Smart Contracts

Joel Frank, Cornelius Aschermann, and Thorsten Holz,
Ruhr-University Bochum

https://www.usenix.org/conference/usenixsecurity20/presentation/frank

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

ETHBMC: A Bounded Model Checker for Smart Contracts

Joel Frank, Cornelius Aschermann, Thorsten Holz

Ruhr University Bochum

Abstract

The introduction of *smart contracts* has significantly advanced the state-of-the-art in cryptocurrencies. Smart contracts are programs who live on the blockchain, governing the flow of money. However, the promise of monetary gain has attracted miscreants, resulting in spectacular hacks which resulted in the loss of millions worth of currency. In response, several powerful static analysis tools were developed to address these problems. We surveyed eight recently proposed static analyzers for Ethereum smart contracts and found that none of them captures all relevant features of the Ethereum ecosystem. For example, we discovered that a precise memory model is missing and inter-contract analysis is only partially supported.

Based on these insights, we present the design and implementation of ETHBMC, a bounded model checker based on symbolic execution which provides a precise model of the Ethereum network. We demonstrate its capabilities in a series of experiments. First, we compare against the eight aforementioned tools, showing that even relatively simple toy examples can obstruct other analyzers. Further proving that precise modeling is indispensable, we leverage ETHBMC capabilities for automatic vulnerability scanning.We perform a large-scale analysis of roughly 2.2 million accounts currently active on the blockchain and automatically generate 5,905 valid inputs which trigger a vulnerability. From these, 1,989 can destroy a contract at will (so called *suicidal contracts*) and the rest can be used by an adversary to arbitrarily extract money. Finally, we compare our large-scale analysis against two previous analysis runs, finding significantly more inputs (22.8%) than previous approaches.

1 Introduction

Cryptocurrencies have gained considerable traction in both academia and industry since the introduction of Bitcoin in 2008 [42]. The underlying technology, called *blockchain*, was originally designed to be a decentralized peer-to-peer payment protocol without the need for trusted parties [42]. Recently, this technology also found applications in many different areas such as supply chain management, asset transfer, or health care (e.g., [6,38,52,63]). A blockchain is a distributed, append-only ledger maintained by all participants of the network. The participants run a consensus protocol to append new data, so called *blocks*, to the ledger, making transactions in the network possible.

Smart contracts, programs deployed directly on the blockchainallow users to encode complex sets of rules on how and when transactions should happen. For instance, a contract can transfer funds when a specific event takes place. It is even possible that multiple contracts are chained together to express more complicated logic. The idea was first introduced by Szabo in 1997 [57], but the first real-world implementation was provided by Ethereum in 2014 [5]. The actual smart contract is typically written in a high-level language, in the case of Ethereum most often *Solidity* [13]. These high-level languages then get compiled to bytecode which is executed on a transaction-based state machine [64], the Ethereum Virtual Machine (EVM).

This offers a great degree of control and the promise of a multitude of use cases, e.g., state or payment channels [16, 24, 58], decentralized crypto exchanges [19], and multi-signature wallets [49]. On the downside, smart contracts suffer from software failures in a similar way as other kinds of programs do. While in traditional programs this may "only" lead to a crash, in the world of Ethereum a simple bug can have more direct—typically financial—consequences. A good example are the infamous Parity incidents [50,59]. In the first event, an attacker exploited a bug in shared library code to steal over 150,000 worth of Ether, the cryptocurrency behind the Ethereum blockchain. At the time of the hack, this was worth around 30M USD. In the second event, the then-patched library was exploited again, this time rendering over 514,000 Ether (around 155M USD) inaccessible.

Several proposal have been made to detect software faults in an automated way. We surveyed 8 of these automated analysis tools [4,23,33,36,39,41,46,62], both from academia and industry, and found all of them lacking in at least one category:

(i) inter-contract reasoning, (ii) memory modelling, especially memcopy-style operations, or (iii) handling of cryptographic hash functions.

In this paper, we address these shortcomings and present the design and implementation of ETHBMC, an automated analysis framework for smart contracts based on a symbolic executor which employs stronger, more precise reasoning over EVM internals compared to state-of-the-art tools. ETHBMC is designed as a bounded model checker, offering the ability to check predefined models against the smart contract's code. In the case a model gets violated, ETHBMC can automatically generate concrete inputs to ease further analysis (i.e., we generate a chain of transactions which demonstrates the detected vulnerability). As a result, ETHBMC is the first method capable of identifying the Parity vulnerability in a completely automated way. We are even able to generate a second exploit not used in the original attack. To demonstrate the capabilities of our tool, we perform a series of experiments in which we compare our approach to the surveyed analyzers. Our main insight is that the imprecise analysis of other approaches can be impeded by even simple toy examples. Continuing, we leverage ETHBMC capabilities as an automated way to generate exploits, scanning all accounts on the Ethereum blockchain (as of December 2018) generating 5,905 exploits. From these 5,905 exploits, we find that 1,989 could be used to arbitrarily destroy contracts (so called *suicidal contracts*) and the remaining ones can be used to extract money. Additionally, we compare our large-scale analysis with two prior works on this topic. First, we compare our analysis results against teEther [33], the state-of-the-art automatic exploit generation tool. We demonstrate that our approach can find significantly more exploits (22.8%) in less time, while also identifying false positives in teEther. Second, we compare against MAIAN [46], a concolic executor, which can be used to find suicidal contracts, and, again find that ETHBMC finds more exploits. Finally, we perform an ablation study on the techniques ETHBMC introduces to show the improvements in a qualitative way. We systematically disable its features while rescanning vulnerable contracts, giving us insights how the different techniques contribute to the analysis results.

Contributions In summary, we make the following three contributions in this paper:

- We provide a survey of the current state-of-the-art analyzers for the Ethereum network, finding all of them to lack precise reasoning over EVM internals.
- We present the design and implementation of ETHBMC, a bounded model checker which handles the identified issues by more precisely reasoning about the internals of EVM. In particular, we demonstrate that a more precise analysis can be achieved by analyzing symbolic memcopy-style operations, inter-contract communication, and by introducing a new encoding scheme for precisely reasoning about cryptographic hash functions.
- We implemented a prototype of ETHBMC in 13,000 lines of Rust code and demonstrate its capabilities in several experiments. More specifically, we compare ETHBMC against all the previously surveyed tools and we also perform a large-scale analysis of the entire blockchain. We show that ETHBMC can be used in an isolated contract environment to increase analysis precision for single contracts, but also scales to large contract analyses where we need to reason about complex interactions of different contracts.

To foster research on smart contract security, the code of ETHBMC is available at github.com/RUB-SysSec/EthBMC.

2 Background

Before diving into the technical details of our analysis process, we briefly introduce the required background information on cryptocurrencies and the Ethereum Virtual Machine (EVM).

2.1 Cryptocurrencies

In 2008, Satoshi Nakamoto introduced Bitcoin and the concept of the blockchain [42], a decentralized ledger running on a peer-to-peer network. Informally speaking, a blockchain is a public, append-only ledger that stores all events happening within the system. The participant run a consensus protocol which ensures, as long as the majority of the network behaves honestly, that the ledger is correct and secured [1].

Ethereum can, in many ways, be considered a “Bitcoin 2.0”. Introduced by Buterin in 2013, it is a cryptocurrency with a Turing-complete bytecode language to orchestrate value transfer in the system [64]. The participants in the network are identified by a 160-bit address, derived from the public part of an ECDSA asymmetric key pair. These so called accounts might, in the case of Ethereum, also have code attached to them. Such accounts are called *smart contracts*, encoding complex behaviour as bytecode programs. The users can send each other money—in the form of Ether—or execute smart contract code by submitting transactions to the peer-to-peer network and signing them with their private key, thus proving the correctness of the transaction. While the length of the execution of a smart contracts is bounded by a parameter called *gas*, i.e., a fee to guarantee that the program eventually terminates, contracts can achieve quite complex behaviours by either chaining transactions together or using multiple contracts to split up the logic.

2.2 Ethereum Virtual Machine

Ethereum defines a special-purpose, stack-based virtual machine termed the *Ethereum Virtual Machine* (EVM) to determine the outcome of a smart contract execution. Ethereum offers a formal specification in a yellow paper [64] where the entire inner workings of the EVM are defined. The machine operates on bytecode where each operand either pops or pushes values to a data stack, each value having a 256-bit word size. Additionally, the EVM is augmented with several mechanisms tailored towards the cryptocurrency environment.

World State The Ethereum world state is the state of the overall system. For the remainder of this paper we will refer to it as the *environment*. It consists of two parts, a mapping from account addresses to an account state as well as the current block information. The account state is a tuple holding several information, such as the current balance of the account [64]. Additionally, if the account is a smart contract, the account state also contains the fields *code* and *storage*. The code field holds the smart contract's code, while the storage is a persistent memory used for keeping values across multiple contract invocations.

Memory The EVM differentiates between three different types of memory:

- **Storage**: The storage is a persistent key-value store, mapping 256-bit keys to 256-bit values.
- **Calldata**: The data section of a transaction is used to supply user input to contracts. Note that this is a byte-addressable data array and immutable during execution.
- **Execution Memory**: This memory is a volatile byte array which only persists throughout one execution. It is used like a heap in classical computer programs, e.g., to store intermediate results during computation.

This setup creates a Harvard-style architecture with separate instruction and data memory. In addition, the EVM offers memcopy-style operations, e.g., `CALLDATACOPY` which copies part of the calldata to execution memory.

2.3 Symbolic Execution and SMT Solving

While the tools we examine in Section 3.3 are based on multiple different program analysis techniques, ETHBMC is based on symbolic execution, thus we provide a brief introduction. Symbolic execution was originally designed as a software testing technique [30], but has since been adopted by the security community for program analysis (e.g., [7, 8, 54]). Instead of concrete inputs, symbolic execution treats all inputs as symbolic variables, which range over the entire input domain of the program. Intuitively speaking, for a function $f(x)$, instead of considering one concrete execution trace, e.g., $f(10)$, symbolic execution considers an symbolic input φ. Resulting in a symbolic function execution $f(\varphi)$, where φ is of the entire input domain, e.g., a 32-bit integer, thus exploring all possible paths a program can take. When arriving at a branch, e.g., an `if`-statement, execution is forked to explore both possible paths. To keep the explored state space low, symbolic executors encode the current state of the program as well as the *path condition* (e.g., `x <= 3`) as a first-order logic formula and use a Satisfiability Modulo Theory (SMT) solver to check if the program path is feasible, refraining from further exploring impossible ones.

SMT formulas are stated in first-order logic, an extension of propositional logic (also known as boolean logic) which offers multiple different theories for formulating problems [2]. The most relevant for our work are the theory of arrays [21] as well as an extension by Falke et al. [17] for addressing memcopy-like operations. A SMT solver performs proof by enumeration: it tries to find a satisfying (concrete) assignment for the constraint system, thus proving it can be solved. When modeling the execution of a program, this concrete assignments provides an input to the program, which can be used to reach a given state. When we additionally encode fault conditions as logical formulas and we find a satisfying assignment for both (i.e., the execution and the fault condition), this concrete assignment is an input to the program which triggers the corresponding software fault.

3 Challenges in Analyzing Smart Contracts

We now present our review of current state-of-the-art tools. We first illustrate common obstacles encountered during analyzing smart contracts by walking through a series of toy examples. We then expand upon this knowledge and examine the infamous Parity wallet, as this bug represents a real-world example where all patterns intertwine. Finally, we present the systematic review of current state-of-the-art tools, finding that none of them deals with all obstacles we identified.

3.1 Common Obstacles in Smart Contracts

We first walk through the identified obstacles in toy examples to better understand the crucial concepts in isolation.

3.1.1 The Keccak256 Function

The EVM offers a specific instruction for computing a keccak hash over a region of execution memory. Solidity-based smart contracts make intensive use of this instruction when implementing the `mapping` data type, essentially a hash table-like data structure. Moreover, the function can be invoked by a smart contract developer manually, e.g., to implement cryptographic protocols like commitment schemes [10].

```
function solve(uint256 input) {
    if (keccak256(input) == 0x315dd8...)
        selfdestruct(msg.sender);
    }
}
```

Listing 1: The direct use of the keccak function.

Listing 1 demonstrates the plain usage of the keccak function which can be invoked by the `keccak256` keyword. A more "hidden" usage of the function is presented in Listing 2, where the instruction is used to calculate a memory location.

Remember that the storage of the EVM is word addressable memory. Fixed size data types have a fixed memory slot allocated. However, when dealing with dynamic data types, i.e., types whose size can grow during execution, we do not know how many memory slots to allocate. Solidity-based smart contracts resort to calculating the memory offset on the fly. When writing to the `mapping` (line 3), the corresponding memory location gets calculated as $keccak256(k \parallel p)$, where k is the key to the `mapping` (`map`) and p is a constant value chosen at compile time [14]. Note if one could generate a valid hash collision utilizing this scheme, prior values would be overwritten.

```
1  mapping(uint => address) map;
2  function createUser(address addr, uint id) public {
3      map[id] = addr;
4  }
5  function destruct(uint id) public {
6      if (map[id] == msg.sender) {
7          selfdestruct(msg.sender);
8      }
9  }
```

Listing 2: Using the mapping data. type

3.1.2 Memcopy-like Instructions

The EVM cannot access calldata directly, it can only operate on data residing within execution memory, i.e., the input data gets copied. In Listing 3, `string` is an unbounded data type, resulting in the EVM utilizing the `CALLDATACOPY` instruction to copy the entire input to execution memory. This is in contrast to data types with a fixed width (e.g., `uint256`) which can be accessed with a plain read from calldata.

```
function solve(string input) {
    if (input[0] == "A" && input[1] == "B"){
        selfdestruct(msg.sender);
    }
}
```

Listing 3: Memcopy-like operation to access input.

3.1.3 Inter-Contract Communication

Ethereum is a decentralized system, offering the ability for multiple contracts to interact with each another. On the downside, these techniques increase complexity of smart contract systems and might lead to unforeseen (security) consequences. A simple example is provided in Listing 4. During the execution of `Target`, a library contract gets called to simulate a simple interaction between two contracts.

The need for inter-contract analysis is furthered by a recent survey by Kiffer et al. [29] on the current contract topology of Ethereum. They state that most contracts are not deployed by humans, but rather are created by other contracts, making these contracts part of intra-contract interactions.

```
contract Target {
    Library private lib = 0xAABBCC...;
    function solve(uint256 input) {
        if (lib.r(input) == 123) {
            selfdestruct(msg.sender);
        }
    }
}
contract Library {
    function r(uint256 input) returns (uint256) {
        return input;
    }
}
```

Listing 4: A simple interaction between two contracts.

3.2 The Parity Wallet Bug

Based on these examples, we now examine the original Parity wallet bug as a real-world example where all previous issues need to be addressed to obtain a comprehensive analysis. While other types of smart contract vulnerabilities were already studied [4, 29, 36, 46, 62], the question how to detect the Parity incident in an automated fashion remained an unsolved challenge. Note that we only present snippets relevant to this analysis which we simplified for easier reading; a full source code listing can be found online [49].

```
1  contract WalletLibrary {
2    address[256] owners;
3    mapping(bytes => uint256) approvals;
4    function confirm(bytes32 _op) internal bool {
5      /* logic for confirmation */
6    }
7    function initWallet(address[] _owners) {
8      /* initialize the wallet owners */
9    }
10   function pay(address to, uint amount) {
11     if (confirm(keccak256(msg.data)))
12       to.transfer(amount);
13   }
14 }
15 contract Wallet {
16   address library = 0xAABB...;
17   // constructor
18   function Wallet(address[] _owners) {
19     library.delegatecall("initWallet", _owners)
20   }
21   function() payable {
22     library.delegatecall(msg.data);
23 }
```

Listing 5: A simplified source code from the Parity wallet.

The Parity wallet is split across two contracts, a library contract holding the majority of the code base and a client contract deployed by the user. Once deployed, smart contracts are immutable, as a result, when changing (or fixing) a contract, one has to redeploy and thus *repay* for the entire contract. In order to lessen the burden on the user, when splitting up the logic, only the library has to be redeployed. The EVM offers the `DELEGATECALL` instruction, an instruction for using another account's code while executing. The instructions switches the code to be exeuted, while still using the original account context and storage. Consider Listing 5, assume the user Alice wants to use the Parity wallet library. She deploys her client code (line 15-23) with a storage variable containing the library contract's account address (line 16). When later calling her client contract, it *delegates* the transaction to the library code (line 22), forwarding the transaction's calldata (`msg.data`). Note that this also implies that if an attacker can redirect the control flow of a contract to an address of her liking, they has the ability to arbitrarily execute code (e.g., extract all the funds).

Since everyone on the blockchain can call into *any* contract, smart contract developers have invented the concept of the *owner*, a variable which is usually set during contract creation, specifying the address of the contract owner. In the case of the Parity multi-signature wallet, there even exists an array of owners (line 2) initialized during the creation of the wallet (line 7-9). Albeit the variable is defined in the library code, since the execution context resides with the original account, the variables is set on the client contract.

Analysis Hurdles Besides inter-contract communication, the Parity wallet utilizes the keccak function, both as a plain call (line 11) as well as in the `mapping` data type (line 3). When hashing the `msg.data` (line 11), due to its (theoretically) unlimited size, the entire data gets copied to execution memory. Thus a static analyzer must be able to reason about inter-contract communication to analyze the distributed contracts as well as memcopy-like instructions and cryptographic hash functions to thoroughly analyze the `pay` function.

The attacker exploited the fact that the `initWallet` function was not marked as private. In Solidity, this implies that it defaults to *public*, i.e. it is callable by anyone. Thus, the attacker first called the `initWallet` function, making himself the owner, and then transferred all funds of the wallet to his account using the `pay` function. Note that the attacker has to perform two transactions, thus only analyzing `initWallet` is not sufficient since the actual exploit happens in the `pay` function.

3.3 State-of-the-Art Techniques

For our survey of existing methods, we chose a variety of tools based on different principles from the program analysis domain, ranging from data-flow analysis (Securify), over symbolic execution (Manticore, Mythril, MAIAN, Oyente, and teEther), to abstract interpretation (Vandal and MadMax). We cannot give a sufficient introduction to every technique, however, the interested reader is referred to the excellent book by Nielson et al. [43]. All discussions concerning specific tools are based on their respective publications [4, 23, 33, 36, 39, 41, 46, 62] and their source code [3, 22, 32, 35, 40, 45, 48, 61] at the time of writing.

During our review, we have found that all tools use some kind of overapproximation which may introduce false positives. As a result, we define **Validation** as an additional potential obstacle; i.e., are any overapproxmiations correctly validated afterwards? An overview of our analysis results is presented in Table 1. Note that MadMax is based on Vandal, thus it inherits its limitations and we only discuss Vandal in detail in the following.

Table 1: Feature comparison between existing tools and our approach.

Tool	Inter-Contract	Memory	Keccak	Validation
Manticore [39]	◐	◐	◐	○
Mythril [41]	◐	◐	◐	○
MAIAN [46]	○	◐	◐	●
Oyente [36]	○	◐	○	○
teEther [33]	○	◐	◐	●
Vandal [4]	○	◐	○	○
MadMax [23]	○	◐	○	○
Securify [62]	○	◐	◐	○
ETHBMC	●	●	●	●

● Correctly implemented ◐ Partially implemented
○ Incorrectly implemented or missing

3.3.1 The Keccak256 Function

Due to the prevalence of keccak computations, most tools we analyzed offer some kind of strategy to deal with them during analysis, but all of them in an imprecise way. All tools offer support for computing keccak values over constant execution memory regions with constant parameters (i.e., every value of memory is non-symbolic). This allows them to extract the corresponding memory regions and compute the actual hash value.

Securify considers during symbolic computations every memory location as a potential dependency, even those who are infeasible in practice. Mythril, on encountering a symbolic offset or a symbolic portion of memory, overapproximates the keccak value with fresh unconstrained symbolic one instead. When any memory value or argument is symbolic, Manticore uses a concolic strategy and fixes the values to constant ones. However, they keep a mapping of all previously computed hashes and try to match the current one to already seen ones. In a similar vein, teEther stores a placeholder object during symbolic execution and then applies a concolic strategy to resolve all seen placeholders. Vandal does not attempt any concrete or symbolic handling, but ignores the instruction and treats the outcome as a new symbolic variable. The outlier to the above schemes is Oyente, it only support concrete keccak computation, but makes no effort in computing the actual values. It rather extracts the string representation of the memory region, compresses and base64 encodes it, and uses this encoding as a mapping to match later hash computations [37].

Our Solution: When encountering a symbolic keccak value, we utilize a special encoding scheme presented in Section 4.6. The scheme is based on the idea that keccak is a *binding* function, i.e., when the same input is supplied to the function, it will produce the same output. We utilize this behaviour by adding constraints to the execution, encoding different keccak computations to be the same, when their input memory regions can be identical.

3.3.2 Memory Modelling

Our review revealed that none of the examined tools fully supports a precise memory model. Some revert to overapproximation or concolic strategies to circumvent complications regarding symbolic memcopy-style operation, while others simply choose not to support them. More specifically, MAIAN supports symbolic read operations, but drops any symbolic write or memcopy-style operation. Mythril supports standard read/write operations, but flounders when encountering copy instructions. It handles concrete ones correctly, yet, when for instance a symbolic offset is supplied to the memcopy operation, it either drops the path or fixes its size to a value of one. Similarly, Manticore and teEther fully support simple memory operations, but resort to concolic strategies otherwise. When encountering any symbolic memory write, Securify behaves conservatively and clears the entire memory, since it cannot reason about specifics anymore. Neither Oyente nor Vandal support any copy-based instruction.

Our Solution: In contrast to previous work, we employ a fully symbolic memory model. We represent the memory as a graph representation, connecting different memory regions when we copy from one to the other (see Section 4.4). When we need to assess the feasibility of a given path, we encode the memory graph as constraints, utilizing the well known theory of arrays [21], as well as the extension by Falke et al. [17], for addressing memcopy-like operations.

3.3.3 Inter-Contract Analysis

Mythril and Manticore are the only two tools supporting inter-contract analysis, however, both do so in an imprecise way. When a contract interacts with another contract, the input for the next execution stems from the execution memory of the callee. Both Mythril and Manticore support fully concrete contract calls, i.e., if the part of execution memory which is used as calldata completely corresponds to concrete values, execution continues as normal. Nonetheless, when any value in the concerning memory region is symbolic, both tools apply different strategies to tackle the problem. Mythril ignores the content of execution memory and overapproximates calldata by creating a new unconstraint array object. In contrast, Manticore utilizes a concolic approach, fixing any symbolic values to constant ones.

Our Solution: We utilize our memory model, which supports symbolic copy instructions, to correctly model the input memory to the call operation (see Section 4.5 for details).

3.3.4 Validation

All tools discussed in this section heavily rely on overapproximation. We want to stress that this is a common approach and is necessary to combat state explosion [9]. Nonetheless, these design choices can obviously lead to false positives. Recognizing these problems, both MAIAN and teEther use private chains to simulate their bug findings in a controlled environment. None of the other approaches makes any attempt at pruning potential false positives.

Our Solution: We follow previous work and simulate each potential bug as a concrete offline execution to weed out false positives. We will discuss details in Section 5.3.

4 Modelling Ethereum

In the following, we provide an overview of the theoretical model underpinning ETHBMC. We start with an overview of attack vectors and a general introduction, move on to our environmental modelling, cover our memcopy-supporting memory model extensively, and finally describe our handling of call and keccak instructions.

4.1 Attacker Model

ETHBMC provides a symbolic, multi-account capable representation of the Ethereum ecosystem which can be used to check arbitrary models. To demonstrate its capabilities, we model three specific attack vectors which we deem most critical: First, an attacker who wants to extract Ether from the analyzed contract. Second, an attacker who wants to redirect the control flow of the analyzed contract to her own account. Third, an attacker who wants to selfdestruct the analyzed contract. Note that we only require our attacker to be able to participate in the Ethereum protocol, giving her a live view of the network and the blockchain, including storage and bytecode level access to contracts (i.e., access to the world state).

4.2 High-level Overview

We want to reason about smart contracts as precisely as possible. This involves an accurate model of the EVM including multiple contracts interacting. However, as all static analyzers do, we have to make choices what to model precisely and what to overapproximate. We decided to neither model the consensus protocol, as well as gas usage. Invalid transactions are guaranteed to not be executed, thus do not influence smart contract state. Moreover, the code we want to analyze has to be executable in practice, i.e., it must have reasonable gas consumption. Note that a series of gas-related issues exists [23], we leave the extension to a gas framework for future work.

We model the EVM as an Abstract State Machine (ASM) Γ, giving us an execution context in which we can reason about a contract's execution. The ASM Γ takes the bytecode array Σ as input, i.e., the contract code, and starts execution at the first instruction. When finishing execution, the machine returns the set of all halting states σ_h, i.e., the set of all states, where

Γ reached a non-exceptional halting condition, as defined by the yellow paper [64].

Additionally, we define a state $\sigma = (\mu,\ \mathsf{pc},\ \Pi)$, where μ represents the stack, pc is the program counter pointing to the next instruction, and Π is the set of path constraints for the given path. We also define $\mu[0]$ to be first (topmost) argument of the stack, $\mu[1]$ the second, and so forth.

4.3 Modelling the Environment

We want to caputre a rich environment model (i.e., the world state) in our executor. Thus, we define an account state to be the tuple $\alpha = (\mathsf{balance},\ \mathsf{code},\ \mathsf{storage})$. Where balance is a symbolic value, representing the balance of an account. code is the (optional) code belonging to an account, and storage is a 256-bit to 256-bit key-value-store, holding the persistent account state (also optional).

Additionally, we define a *transaction (tx)* to be the tuple $(\mathsf{origin},\ \mathsf{recipient},\ \mathsf{callvalue},\ \mathsf{calldata},\ \mathsf{calldatasize})$. origin being the origin account address of the transaction, recipient being the recipient, callvalue being the value attached to the transaction, calldata being the (optional) calldata attached to the transaction, and, calldatasize being a symbolic variable representing the size of the calldata array (again optional).

We expand the definition of Γ by adding a mapping $\mathsf{accounts}: \ address \rightarrow \alpha$ mapping account addresses to their respective states, i.e., the world state. Additionally, we introduce the set transactions which represents all transactions issued in the system. When analyzing a specific contract, we set up an attacker account and (possibly multiple) victim account(s). We then simulate a chain of transactions $t_1, t_2, \ldots, t_k$. For each transaction, we execute an entire run of Γ yielding the halting states σ_h. Then, for each $\sigma_i \in \sigma_h$, we fork execution and proceed with the next t_j up to t_k.

4.4 Memory Model

Our memory model is based on the work of Sinz et al. [55]. It models memory as a series of updates called the *memory modification graph*. We extend this graph notation to accommodate for the EVM characteristics, such as multiple memory regions.

4.4.1 Memory Graph

The graph itself is used to keep track of all modifications to memory. It starts with an initial node and gets updated at every read/write to memory. More formally, we introduce the *memory graph* $\Delta = (V, E)$ which holds all memory nodes and add it to our state definition, i.e., $\sigma = (\mu,\ \mathsf{pc},\ \Pi,\ \Delta)$. We assign every node a unique index. Thus, we denote the node with index i as n_i. Additionally, we assign every node a label, either `init` or a memory altering operation (`write`, `copy`, or `set`), to keep track of which operation created the node.

For now, we only consider one memory region, e.g., storage. We start from an initial node s_0, updating the graph every time we encounter a write to memory (e.g., `SSTORE`) by creating a new node s_t connected to its parent's node s_u. This gives us a unique memory image at any state during execution (akin to static single assignment (SSA) form known from compiler theory). When translating the memory layout to constraints, we start from the newest node s_t, traversing the graph in a backwards fashion collecting all memory updates, and encoding them as logical formulas based on their respective label [17, 21]. This approach enables us to reason about symbolic memory operations.

When considering multiple memory regions, e.g., execution memory and calldata, we introduce an initial node for each region, i.e., the graph starts as a forest, and remains as such as long as memory operations only operate on single memory regions. It can get connected in two different ways. First, indirectly by loading from one region and storing to the other, linking two parts of the graph implicitly through a constraint. Second, directly by a memcopy style operation (e.g., `CALLDATACOPY`), linking two parts of the graph explicitly through a node and edges. Loading and storing introduce constraints in the system, linking the memory regions when they are translated to first-order-logic. Copying introduces a new node in the destination tree of the forest (e.g., the execution memory for `CALLDATACOPY`). This node gets connected to both source and destination regions of memory, explicitly linking the two parts of the graph.

4.4.2 General Memory Operations

During execution, for every account, we create a new storage memory node n_j and store the corresponding index in its account state $\alpha.storage \leftarrow n_j$. In the same vein, every transaction creates a new calldata and stores its identifier. Additionally, we assign our ASM Γ an execution memory $\Gamma.m \leftarrow n_k$. We define reading and writing to memory as follows:

- $\Delta.\mathsf{write}(n_i, p, v) \mapsto n_j$: Writes the value v to the address p with the memory node n_i as parent node, returning the new node n_j.
- $\Delta.\mathsf{read}(n_i, p) \mapsto v$: Reads the value v from memory n_i at position p.

This makes modelling the `SSTORE` instruction straightforward:

$$\alpha.storage \leftarrow \Delta.\mathsf{write}(\alpha.storage, \mu[0], \mu[1])$$

In this example, we write to the current storage of the account ($\alpha.storage$) the value $\mu[1]$ to the address $\mu[0]$. This is represented by creating a new memory node n_j in the memory store Δ. We then assign the index of this new memory node to be the current account storage $\alpha.storage$.

Modelling other memory operations is more difficult, since the word size of the EVM is 256-bit. However, both calldata, as well as execution memory, are byte-addressable memories.

As a result, we have to translate between 256- and multiple 8-bit chunks.

$$n_0 \leftarrow \Delta.\text{write}(\Gamma.m, \mu[0], \mu[1][31])$$
$$\dots$$
$$n_{31} \leftarrow \Delta.\text{write}(n_{30}, \mu[0]+31, \mu[1][0])$$
$$\Gamma.m \leftarrow n_{31}$$

We denote the lowest byte of $\mu[i]$ with $\mu[i][0]$ and the highest (leftmost) with $\mu[i][31]$. We model `MSTORE` as a sequence of 32 8-bit writes to execution memory, shifting the address and the extracted 8-bit sized chunks, accordingly. Reading from execution memory is done similarly, reading 8-bit chunks while shifting the read index, concatenating the result.

When modelling `CALLDATALOAD`, the EVM defines calldata as a theoretically unbounded array. Thus when a memory operation reads out of bound, i.e., a location greater than `calldatasize`, the EVM simply ”reads“ zeros. Thus for every read from calldata, we wrap it in an `ite` (IF-THEN-ELSE) operation, which constraints the load to evaluate to zero when the supplied address reads out of bounds.

4.4.3 Supporting Memcopy- and Memset-Style Instructions

The EVM offers multiple instructions which behave in a memcopy-like fashion. We define the following functions on Δ:

- $\Delta.\text{set}_\infty(n_i, p, v) \mapsto v$: Sets all values in memory n_i, starting at position p, to value v
- $\Delta.\text{copy}(n_i, p, n_j, q, s) \mapsto n_k$: Copies a size s chunk from node n_j, starting at position q until $q+s$, to node n_i, starting at position p until position $p+s$

These functions enable an implementation of memcopy-style operation and simplify memory initialization. Both storage and execution memory are assumed zero at the start of their lifetime. Utilizing the set_∞ function, we can initialize these regions. We utilize the Theory-of-Memcopy introduced by Falke et al. [18] to implement these operations efficiently. This theory extends the Theory-of-Arrays [21] to support C-style memcopy operations, making the translation to constraints possible.

4.5 Modelling Calls

As previously introduced, the EVM offers contracts to interact with one another. Consider Figure 1 assuming we simulate a user transaction targeting contract A.

We would first setup an ASM_A to simulate the execution of contract A, resulting in an execution tree for A. Now assume that—during the execution—we encounter a message call to contract B. We then set up ASM_B, run through the entire execution and then fork the execution tree for each state $\sigma_i \in \sigma_h$. This enables us to simulate each possible outcome for the message call. Note that this technique can be applied recursively

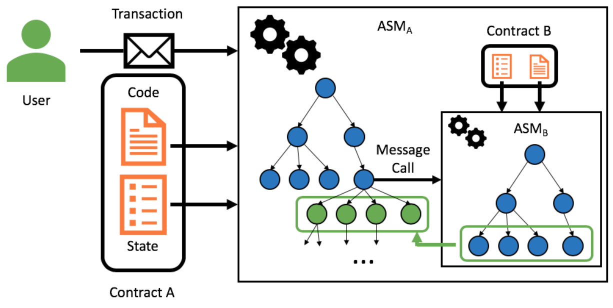

Figure 1: Message call into another account.

to simulate nested message calls. Similarly, when executing `DELEGATECALL` or `CALLCODE` instructions, we switch the account's code and proceed as outlined above. When calling into another account, the EVM uses part of the execution memory as input to the new execution. Continuing our running example, when executing the message call from ASM_A to ASM_B, we create a new `calldata` node in Δ and then utilize the `copy` function to copy over the input from the execution memory of ASM_A. When execution of ASM_B finishes, we copy over some of the execution memory from ASM_B to ASM_A, serving as return data [64].

4.6 Handling Keccak Instructions

The EVM offers a specific instruction for computing a keccak-256 hash over a portion of execution memory. However, these functions have been proven difficult for static analysis in the past [56]. One common technique is to use an Ackerman encoding, used for encoding non-interpretable functions [2]. It exploits the fact that cryptographic hash functions are binding functions [10], i.e., under the same input the function is guaranteed to produce the same output. We can leverage this property as follows:

$$\begin{aligned} x = y &\Rightarrow hash(x) = hash(y) \quad \wedge \\ x \neq y &\Rightarrow hash(x) \neq hash(y). \end{aligned} \tag{1}$$

However, since the EVM computes the keccak function over execution memory, we cannot directly utilize this encoding for our purpose. When encountering a keccak computation, we proceed as follows: If all dependent variables and memory regions are constant, we simply compute the constant hash value. Otherwise, we replace the outcome with a placeholder object, which stores a current image of the execution memory, as well as, the starting and end addresses of the keccak computation. When we want to assess the feasibility of a given execution path, instead of directly encoding Equation (1) on the inputs, we encode it for each memory address instead.

More formally, we define the tuple keccak with three fields: (i) keccak.addr, the starting memory address, (ii) keccak.len, the length of the memory range to be considered, and (iii) keccak.m, which is the index of the execution memory present

Algorithm 1: EncodeKeccak

```
   Input  : Two distinct keccak tuples $i$ and $j$, the execution constraint
            set $\Pi$, the memory graph $\Delta$
   Output : A modified constraint set $\Pi'$
1  begin
2      if isSymbolic($i.len$) or isSymbolic($j.len$) then
3          return
             $\Pi \bigcup \{(i.len = j.len \Rightarrow i = j) \wedge (i.len \neq j.len \Rightarrow i \neq j)\}$
4      if $i.len \neq j.len$ then $\Pi' \leftarrow \Pi \bigcup \{i \neq j\}$
5
6      else
7          $\gamma \leftarrow (i = j)$
8          for $k \in 0...i.len$ do
9              if $\Delta[i.m, i.addr + k] \neq \Delta[j.m, j.addr + k]$ then
10                 $\gamma = \emptyset$; break
11             $cond \leftarrow (\Delta[i.m, i.addr + k] = \Delta[j.m, j.addr + k])$
12             $\gamma \leftarrow$ ite($cond$, $\gamma$, $i \neq j$)
13
14         if $\gamma \neq \emptyset$ then $\Pi' \leftarrow \Pi \bigcup \{\gamma\}$
15         else $\Pi' \leftarrow \Pi \bigcup \{i \neq j\}$
16
17     return $\Pi'$
```

at the time of computation. We encode all possible pairs of keccak tuples using Algorithm 1. Assume two distinct tuples i and j which we want to translate to first-order logic and add to the path constraints Π. We first try to utilize more sophisticated encodings. However, in cases where we cannot argue over the len parameter (e.g., one parameter is an unconstrained symbolic variable), we resort to a fallback encoding (line 2-3).

Assuming both $i.len$ and $j.len$ to be constant, we can utilize a more sophisticated scheme. First, we can trivially disprove that two values compute to the same hash if $i.len \neq j.len$ (line 4) and thus simply add $i \neq j$ to Π. Second, when both values match (line 6-16), we construct a nested ITE (IF-THEN-ELSE) expression over the possible memory location (line 9-13) used for the hash computation. When constructing the encoding, at each level we check if we can trivially disprove two memory locations to be equal (line 9), otherwise we can instantly abort (line 10) and encode both keccak values to be unequal (line 15). Otherwise we keep iterating along the range of the len parameter (line 8). Traversing each memory location (line 9-12), we construct the condition $\Delta[i.m, i.addr + k] = \Delta[j.m, j.addr + k]$, encoding that the memory position for i must be same as j to compute to the same hash. At each iteration, we assign the true branch of the ITE expression to the encoding from the previous iteration of the loop, a special case being the first iteration of the loop, where we supply $i = j$. Thus, if our backend SMT solver traverses the nested encoding and it can prove all memory locations to be equal, it will eventually arrive at the final predicate $i = j$. However, if it disproves any condition, it arrives at the negated predicate $i \neq j$ which we assign at every iteration of the construction.

At first glance, requiring that both keccak tuples depend on constant length parameters might seem like a strong assumption. In practice however, this is often the case, e.g., keccak values computed over fixed size data structures always have a fixed length, the same is true for calculating memory offset for the *mapping* data type. As introduced in Section 3.1.1, it gets accessed by a keccak operation. Hence, we additionally extract the key part of the computation to later match read/writes.

Additionally, when we encounter an equality check with a constant variable, i.e., keccak $== c$, where c is constant, we can immediately conclude that the result must be non-equal. Otherwise, we would assume that we could calculate hash collisions. In other words, we would assume that we could compute the one specific input, which leads to the (constant) output of the keccak function. Note: in specific circumstances an attacker might know the correct input which generates c; we elaborate more on this in Section 7.

5 Design and Implementation

We now provide an overview of the architecture of ETHBMC, a graphical overview is provided in Figure 2. The tool consists of three main modules, the *symbolic executor*, a *detection module*, and a *validation module*. ETHBMC is implemented in around 13,000 lines of Rust code.

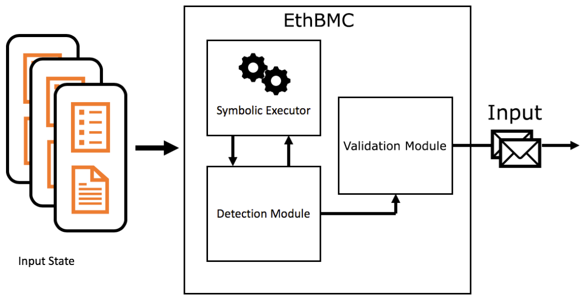

Figure 2: High-level overview of ETHBMC and its inner workings.

ETHBMC utilizes its symbolic execution engine to explore the available state space a program can reach (Section 5.1). During this exploration, we can, at any time, translate the necessary conditions (or constraints) needed to reach this state into first-order logic. When the exploration finishes, i.e., the execution terminates in a halting state, we encode the attacker's goal using additional constraints (Section 5.2). As an example, we encode a constraint that the balance of the attacker's account must be higher at the last state of execution than at the first state. We then utilize our backend SMT solver to solve the constraint system. As introduced in Section 2.3, an SMT solver performs proof by enumeration: it tries to find a satisfying (concrete) assignment for the constraint system, thus proving it can be solved. We model the full execution of smart contracts. Thus, a satisfying assignment that both reaches a valid halting state and fulfills the attacker model, proves a vulnerability in the contract. Additionally, the concrete assignment, found by the SMT solver, is a valid input (i.e., transaction) to the smart contract, which triggers the exploit. Finally, we verify that the exploit is a true positive by running a concrete offline execution (see Section 5.3).

5.1 Symbolic Executor

The executor explores the contract in a breadth-first search. Whenever the executor needs to assert satisfiability of a given code path, we query our backend SMT solver. We evaluated different solvers and found that Yices2 [15] outperforms other approaches such as Boolector [44], and Z3 [12] in this problem domain (see Section 6.5). We explore all code paths until either they reach a halting state, or the solver times out or disproves the path. If we encounter a loop during execution, we use loop-unrolling, i.e., we execute the loop n number of times, after which we drop out of the loop. We use the same strategy in limiting call depth, since in an environment with multiple accounts, contracts could keep calling each other in infinite loops. Additionally, we employ several standard symbolic execution optimization techniques: constant folding, arithmetic rewriting, and constraint set caching [7]. When the executor comes to a hold, all end states are passed to the detection module for further analysis.

5.2 Detection Module

We encode the attacker's goal using additional path constraints, e.g., we push an additional constraint specifying that *after* the current transaction executed, the balance of the attacker account must be higher than at the start of the entire analysis. When encountering a `DELEGATECALL` or `CALLCODE` instruction, we create an additional state *hijack*, where we try to hijack the control flow of the contract. We add a constraint to *hijack*, constraining the target address of the `CALLCODE`/`DELEGATECALL` to be the attacker's account address. If this constraint is satisfiable, we can redirect the control flow. In a similar vein, we flag states which execute a `SELFDESTRUCT` instruction, to detect contracts that can be destroyed by an outside attacker. Note that if the `SELFDESTRUCT` instruction can be used to steal money from the account, ETHBMC detects both cases. If we detect any type of vulnerability, we pass the corresponding state to the validation module.

If we cannot detect any attack, we compute the set of *state altering* states, i.e., the subset of σ_h which experienced changes to their environment. Only these states can provoke new paths in the executor, other states would result in the same initial states as explored in the previous round. Thus, we only explore these states further.

Table 2: Results of evaluating different analyzers on toy examples.

Tool	Keccak	Mapping	Memcopy	Inter-Contract	Parity
teEther	○	●	●	○	○
Manticore	●	○	○	●	○
Mythril	○	○	○	n/a	○
Vandal	○	○	○	○	○
MadMax	○	○	○	○	○
Securify	●	○	○	○	○
ETHBMC	●	●	●	●	●

● Correct ○ Incorrect or not supported

5.3 Validation Module

In the last step, we try to generate valid transactions for every state which has a feasible attack path. We utilize our the SMT solver to generate the transaction data needed to trigger the vulnerability. After successfully generating attack data, we leverage the go-ethereum [20] tool suite, especially the EVM utility, to simulate the attack in an offline fashion. This allows us to simulate all the generated transactions and check if they indeed match their required attack vector.

6 Evaluation

We evaluated ETHBMC in several different experiments and focus on the main results in the following.

6.1 Empirical Analysis of Current Techniques

We start with comparing ETHBMC against the static analysis tools examined in Section 3.3. We use the toy examples presented in Section 3.1 as a set of trials. We embedded a `SELFDESTRUCT` instruction in each contract, since all tools offer a detection module for this. Additionally, we recreated the Parity account hack examined in Section 3.2 to simulate a complex, real-world scenario. A general overview of our findings is presented in Table 2.

Analysis Setup Unfortunately, we could not get MAIAN to work properly; multiple libraries required by the analyzer are by now incompatible. The authors neither specified which version they used in the original publication, nor responded to multiple GitHub issues regarding these problems [47]. Again, we only discuss Vandal since MadMax inherits its capabilities.

We evaluated against the latest version of the tools at the time of writing. This corresponds to teEther at github commit d7b7fd1 [32], Manticore in version 0.2.4 [48], Mythril in 0.20.0 [40], Vandal at github commit f7bfee7 [3], securify at github commit 8fd230 [61] and Oyente at github commit 6c9d382 [40]. While Oyente offers a mode to detect exposed `SELFDESTRUCT` instructions, we discovered during testing that the mode seems to be inherently broken. As a sanity check we tested a simple contract with a simply one line function which selfdestructs the contract (i.e., Listing 1 without the surrounding if clause). Oyente flags the contracts as non-vulnerable. Thus, we exclude it from the experiment.

For the evaluation, we compiled all contracts to bytecode and used this as input to the different analyzers. This guarantees that the comparison is fair among all tools and no one can get an advantage by leveraging source code information.

Keccak256 Function We start with the simple contract testing the analyzer's abilities to model hash functions, i.e. Listing 1. The contract compares the hashed input to a randomly-chosen constant value. If the attacker wanted to pass the check (line 2), they would have to supply a preimage. Since keccak is a cryptographically secure hash function, this is infeasible in practice and the contract is not vulnerable.

Manticore, Securify, and EthBMC correctly identify the contract as secure, all other tools report a vulnerability. However, according to our source code review, teEther should pass the experiment. In a first pass over the contract, teEther uses binary slicing to find paths resulting in potentially vulnerable states. In a second path, it executes these paths symbolically to find an input which can potentially reach this state. However, for this experiment, teEther reports that it cannot find a potential path containing a `SELFDESTRUCT` instruction. According to our understanding, it should only discard the possibility of an exploitable contract in the second pass. Thus we list teEther as incorrect for this experiment.

Due to the prevalence of the `mapping` data type, we continue our analysis with the contract listed in Listing 2, an attacker could exploit the contract by first calling `createUser`, supplying her own account address as input, then calling `destruct` with her assigned `id`. Only teEther and EthBMC find the vulnerable state.

Memcopy-Style Operations The next experiment is meant to test the executors' handling of memcopy-style operations. We use the contract depicted in Listing 3. Since the input is defined as `string`, the calldata gets copied to memory, using a memcopy-like instruction.

On first glance, Securify seemed to pass the experiment, reporting a vulnerable state. However, this is in direct conflict with our source code review in Section 3.3 as we discovered that it simply ignores memcopy-esque instructions. We thus perform a second validating experiment as follows:

```
1 function alias(string input, uint x, uint y) public {
2     require(x == y);
3     if(input[x] != input[y]) {
4         selfdestruct(msg.sender);
5     }
6 }
```

Running the experiment two times, one as is, and one where the condition on line 2 is negated, resulted in Securify flagging both instances as vulnerable. This confirms our suspicion that Securify does not correctly reason about this program, since the instance presented above is clearly non-vulnerable. We repeat this experiment for all tools with no change in outcome. All tools except teEther and EthBMC fail to find a vulnerable state.

Inter-Contract Analysis Analyzing inter-contract analysis proved tricky for Mythril; the tool supports inter-contract analysis, but the contract has to be already deployed on a blockchain. Thus we exclude them from this test, only leaving Manticore for evaluation since none of the other tools support inter-contract analysis. The experiment is simulated using two contracts `Library` and `Target`, mirroring the toy example presented in Listing 4. We assume `Target` to be the contract which gets analyzed. Both Manticore and EthBMC find correct inputs for this example.

Parity Finally, we recreate the Parity account hack examined in Section 3.2 to simulate a complex, real-world scenario. We run an archive Ethereum node which stores all past information of the network. This allows us to retrieve state and environment information for any past block. We use this information to analyze one of the exploited accounts, 10 blocks before the hack took place.

Mythril offers an on-chain analysis mode, where it downloads all necessary live information from the blockchain. Unfortunately, it only supports analysis at the currently newest block. We extend the tool to work with past blocks and are currently in the process of submitting this patch to the upstream repository. However, when analyzing the parity contract, Mythril does not report any vulnerabilities.

EthBMC does support a mode similar to Mythril: we extract the storage information at the specific block and preconfigure the environment with them. When reaching any call-based instruction, we extract any constant arguments and load the corresponding receiver contract. EthBMC finishes analysis and correctly reports two ways to exploit the contract. In the actual parity code, the constructor and initialization code are split across two functions. Thus, an attacker can either call the exposed constructor or the initialization method directly. EthBMC generates valid attack code for both vulnerabilities.

Manticore does not support any kind of online analysis. Therefore, we extract the storage parameters at the corresponding blocks and set up a test environment with both accounts by utilizing their API. After processing the first transaction, Manticore reports that it has not detected any state which can be explored further and finishes the analysis without reporting any issues.

6.2 Large-Scale Analysis

To further evaluate EthBMC, we conducted a large-scale scan of all 2,194,650 accounts listed on Google BigQuery [11] as of 24. December 2018. We split the scan into three stages, enabling us to directly compare it against two previous large-scale experiments performed: the first by Krupp and

Table 3: Large-scale analysis results displaying the amount of contracts found (with the amount of unique exploits generated in brackets)

Analyzer	Steal Ether		Hijack		Suicidal		Total	
ETHBMC	**1,681**	(1,893)	**51**	(54)	1,431	(1,474)	2,856	(3,367)
teEther	1,509	(1,541)	8		-		1,509	(1,541)
ETHBMC	1,693	(1,964)	51	(54)	**1,439**	(1,482)	2,921	(3,448)
MAIAN	-		-		1,423		1,423	
ETHBMC	2,708	(3,916)	97	(123)	1,924	(1,989)	4,301	(5,905)

Rossow [33] and a second one by Nikolic et al. [46]. Krupp and Rossow presented teEther which uses binary slicing in conjunction with symbolic execution. The tool focuses on extracting Ether, as well as as redirecting control flow. Nikolic et al. developed MAIAN, a concolic executor, to study suicidal accounts, i.e., accounts which could be destroyed by anyone. An overview of our findings is presented in Table 3 and discussed in detail below. Note that, as in the Parity example, ETHBMC often found multiple ways to exploit the same vulnerability, thus we list the number of unique exploits found during analysis in brackets.

Experiment Design Since we run an archive node, we can freely recreate account environments at any given block height. We utilize this capability to first recreate the environment at which Krupp and Rossow conducted their scan, analyzing all accounts listed by their dataset. Subsequently, we extracted all contract addresses present at the time of Nikolic's scan. Since we want to avoid unnecessary rescanning of contracts, we continue with only scanning the difference between the teEther and MAIAN account set. We calculated this difference by collecting all newly created accounts, as well as all accounts whose account state changed between the two blocks, thus "updating" our view of the blockchain to the newer block. Finally, we used the same method to calculate the difference between the MAIAN scan and all the accounts listed on Google BigQuery as of December 2018, giving us a complete picture of the current Ethereum vulnerability landscape. Note we chose both the teEther and MAIAN scans since both tools provide false positive pruning, enabling a fair comparison.

Due to the scale of our analysis, we have to impose some restrictions on ETHBMC. The analysis is configured to use a 30 minute timeout. Moreover, we bound loop execution to one iteration, use a two minute timeout for our backend SMT solver, as well as only loading up to 10,000 storage variables. When an accounts has zero balance on chain, we assume a substitute of 10 Ether so the model checker can reason about extracted Ether. Additionally, we limit transaction depth to three transactions and introduce an additional constraint to our execution to limit memcopy operations to size 256, mimicking teEther's behaviour.

We used a cluster of machines for our experiments: 20 virtual machines in our university's internal cloud running 6 $\times$ 2.5 Ghz virtualized cores with 12 GB of memory assigned each. Additionally, we ran 12 ETHBMC instances on two servers, each equipped with an Intel Xeon E5-2667 and 96GB of memory. Scanning 2,193,697 unique accounts took the entire cluster around 3.5 months in total, which equals to roughly 39 CPU years.

teEther We contacted the authors of teEther [33] and got access to their experimental data and performed an analysis of all 784,344 accounts listed by their dataset on the same date (Nov 30, 2017). Note that Krupp and Rossow first assumed an empty storage during their analysis. This, in conjunction with only single contract analysis, allowed them to skip analyzing duplicate contract codes resulting in a reduced initial analysis set of 38,757 contracts. They first analyzed this reduced contract set for vulnerabilities. When their tool flagged an account as vulnerable, they searched the bigger set for all accounts which share this contract code. Subsequently, they than reran their analysis for these accounts while also extracting the corresponding environment (e.g., the storage variables of these accounts). However, note that this shortcut might miss vulnerable contracts since they may behave differently based on initialized storage variables and accounts they interact with. To avoid this, we scan all 784,344 accounts separately, extracting initial storage variables, as well as called accounts discovered during the analysis. We want to stress that both scans target the same set of contracts, we only differ in the approach.

Our analysis finished successfully for the majority of contracts (91.21%), with only a small number of timeouts (2.41%). In comparison, teEther successfully analyzed 85.65% of the contracts. Due to the large-scale nature of our analysis, we did encounter multiple errors during analysis (6.38%). Some are the result of a bug in EVM, the framework used for validation. Some are related to us not being able to load the account from the blockchain which is an issue we are currently still investigating. However, in any case we are conservative and flag the corresponding account as an error, excluding it from analysis.

After both stages of their analysis, Krupp and Rossow report 1,532 vulnerable accounts. During our analysis, we discovered 2,856 vulnerable contracts, 1,681 contract from which we could extract Ether, 51 whose control flow we could redirect, and 1,431 which we could kill at any time (i.e., suicidal contracts). Note that an account can be flagged in multiple categories, e.g., 255 accounts are both flagged as suicidal and able to extract Ether. During their evaluation, the teEther authors list accounts which are vulnerable to hijacked control flow, both in a separate category, as well as in the steal ether category. The reasoning being, that once an attacker can redirect the control flow, they can easily extract all funds from the account [33]. We follow their lead to enable better comparability.

We examined how our results directly compare to the accounts flagged as vulnerable by teEther. During our analysis, we flagged 1,493 out of the 1,532 accounts as vulnerable. The remaining 39 are either timeouts (16) or reported benign by ETHBMC (23). We discovered that teEther does not correctly model the environment, i.e., during analysis they treat all environmental information (e.g., the block hash or block number) as fully symbolic. When their framework flags an account as potentially vulnerable, they try to correct these overapproximations by simulating the environment with a private development chain. However, they start the private chain with the default initial parameters, beginning the chain at block number one. In contrast, we simulate the execution at the corresponding real-world blocks and supply the environment we discovered during live analysis. The authors stated that this also caused problems while generating exploits in the original publications [33] and, after contacting them, they confirmed our suspicion about such false positives, leaving 1,509 vulnerable accounts with 1,541 valid exploits. In summary, ETHBMC is able to find 10.3 % more vulnerable accounts and 22.8 % more exploits than teEther.

MAIAN Nikolic et al. conducted their own analysis by scanning 970,898 contracts on a later date than teEther [46]. Unfortunately, their data set is not available to us and we could not recreate their experiments due to the problems described in Section 6.1. We scanned up to the same blocknumber and found a total of 1,439 (+1.1%) accounts to be suicidal, MAIAN found 1,423. As we do not have access to the experimental data, we speculate that the concolic execution used by MAIAN underapproximates several contracts. Our analysis successfully finished for 92.46% of all accounts, a slight improvement compared to the teEther results.

Current Vulnerability Landscape Finally, our last scan revealed a total of 4,301 vulnerable, active contracts on the Ethereum blockchain as of December 2018. These are split between 2,708 contracts from which we could extract Ether, 97 accounts whose control flow can be redirected, and 1,924 contracts which we could selfdestruct at will. Our technique still finished successfully for around 92.49% of all contracts.

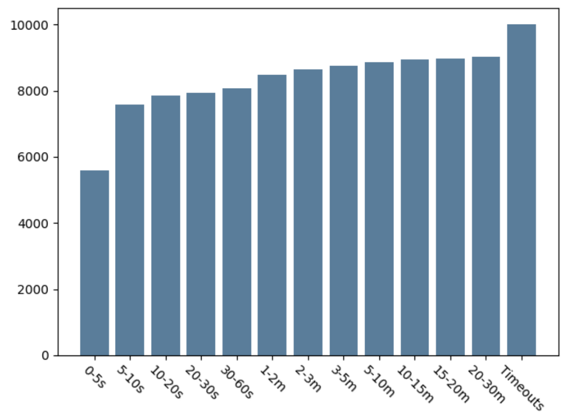

Figure 3: Cumulative overview of analysis time of 10,000 randomly sampled contracts. Note that the x-axis is not linearly scaled.

6.3 Performance Analysis

In Section 6.2, we demonstrated ETHBMC's ability to scale to large datasets. However, we are also interested in its performance when analyzing individual contracts. We randomly sampled 10,000 contracts from our dataset and conduct a study of our analysis time. Note that if the contract interacts with other contracts, we still load them from the blockchain. The results are presented in Figure 3.

From the 10,000 contracts, we successfully analyzed 5,577 in the first 5 seconds and an additional 2,006 in 5 to 10 seconds(i.e., a total of 7,583 in 10 seconds). Afterwards, the number of solved contracts gradually increases, with 8,471 of 10,000 contracts being solved in the first 2 minutes. After 30 minutes, we have successfully analyzed 9,031 out of 10,000 accounts, i.e., around 90%, which mirrors our performance during the large-scale analysis. Note that we plotted errors and timeouts together in Figure 3 for a better presentation.

6.4 Ablation Study

We perform an ablation study to gain a better insight into ETHBMC's inner workings and how the enhancements presented in Section 4 affect the model checker's ability to detect vulnerabilities. We re-scan all vulnerable accounts found in the first phase of our experiments, i.e., our evaluation comparing against teEther, while successively disabling different features. This gives us a clear picture which feature contributes to finding additional bugs. Note that we chose the teEther contracts to gain a frame of reference with a different approach, i.e., concolic execution. Since ETHBMC is a multi-threaded system, we raise the timeout limit to one hour ensuring the difference it not by chance. We disabled the memcopy feature, leaving us with a memory model similar to other memory models discussed in Section 3.3. When disabling inter-contract calls, we still simulate a full environment with transaction (and thus Ether transfer), i.e., we still simulate an

Table 4: Ablation Study of ETHBMC

Features	Steal Ether		Hijack		Suicidal		Total	
teEther	1,509		8		-		-	
Baseline ETHBMC	1,543		50		1,403		2,709	
+ Memory	1,557	(+0.91%)	51	(+2%)	1,409	(+0.43%)	2,725	(+0.6%)
+ Keccak	1,628	(+4.56%)	51		1,425	(+1.13%)	2,803	(+2.86%)
+ Calls	1,681	(+3.36%)	51		1,431	(+0.42%)	2,856	(+1.89%)

attacker account executing the victim account. However, the analyzed contract cannot call (or `DELEGATECALL`) into other accounts. Lastly, we disable the keccak handling presented in Section 4.6, overapproximating every keccak computation with a fresh symbolic variable.

The results are presented in Table 6.4. Note the Baseline ETHBMC row refers to ETHBMC with all three features turned off. The percentages are calculated relative to the previous row, read top to bottom, i.e., additionally enabling the keccak handling resulted in a 4.56% increase compared to only enabling a full memory model. The study clearly shows that all three features play a crucial role in discovering additional bugs when compared to previous approaches. While the memory feature might not seem too important, note that the memory model is so precise to enable inter-contract analysis. As presented in Section 4.5, when executing an inter-contract call, the calldata of the new call is *copied* from the old execution memory. In the same vein, the returndata of the call gets *copied* back to execution memory. Thus, one might also interpret these features as one, which puts them to an about equal contribution to the keccak handling.

6.5 SMT Solver

All executors evaluated in Section 3.3 use Z3 as their backend solver [33, 36, 39, 41, 46]. However, during our research we empirically discovered that using other SMT solvers resulted in a drastic performance gain. We compare three participants of the 2018 SMT competition [26] in the category QF_ABV (quantifier-free theory of arrays and bitvectors), Boolector [44], Z3 [12], and Yices2 [15]. From the account addresses computed in Section 6.2, we randomly sampled 1,000 addresses to evaluate our backend SMT solver. All experiments were run on a server with an Intel Xeon X5650 CPU and 48GB Memory. We run ETHBMC on the 1,000 addresses and recorded all queries sent to the SMT solver resulting in 1,161,498 unique queries. From these queries, we randomly sampled 10,000 queries and ran them on each solver 5 times, with a two minute timeout, averaging the results.

The results are plotted in Figure 4. We omit some smaller formulas since all solver handle them almost instantly. The best performing solver in our experiments is by a wide margin Yices2, followed by Boolector and Z3 being the worst. From anecdotal evidence, we can report that switching our backend solver to Yices2 cut our analysis time down by a third. Thus, we highly encourage other projects to evaluate this change as well and test different SMT solvers.

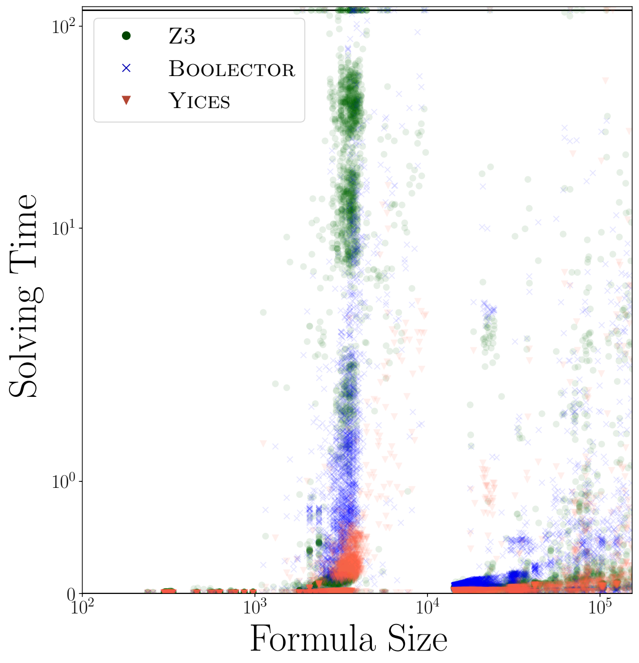

Figure 4: Solving time for a sample of formulas produced by ETHBMC across various common solvers.

7 Discussion

In the following, we discuss the underlying assumptions and limitations of ETHBMC.

Environment Model While our environment model is precise, we still have to impose some limitations on it. When executing an instruction which interacts with other accounts in the environment, e.g., the instruction `BALANCE` or `CALL`, we only consider accounts in the currently loaded environment as valid targets. Otherwise, we would have to consider every single account in the Ethereum ecosystem as a valid target. While we could simply model the execution fully symbolically, this would also introduce the drawback that such an account constellation might never even be possible. Thus, we decided to only consider accounts supplied to the environment or discovered during live analysis. Also, we do not model account creation. At the time of writing and to the best of our knowledge, no one has evaluated account creation as an attack vector.

Restrictions During our evaluation, we had to impose some restrictions on our framework, such as bounding loops and setting a time limit. While some of these restrictions cannot be lifted completely, e.g., we always have to impose an upper limit on loops, raising the timeout limit or loop count may lead to discovering bugs hidden deeper in programs. The same applies for contract invocations, i.e., ETHBMC cannot find bugs, which require more than three transactions. Also, we only model one attacker account at the moment. However, since smart contracts are used to model complex systems, actually including additional attacker or user accounts might lead to discovering interactions which may only be triggered when multiple parties are using the contract. Note that since ETHBMC already supports a full environment, it has the capabilities to be used in this fashion.

Extending to Other Vulnerabilities Our model checking approach can detect new attack vectors by modeling new vulnerabilities as constraints. Additionally, EthBMC can be utilized to provide formal guarantees over contracts. An analyst would model the correct behavior of the contract as a constraint system. In a standard model checking procedure, EthBMC would then be used to check if there exists a state which is both reachable, as well as satisfying the negation of the constraint system. These properties prove a violation of the correct behavior. The reachability assesses that the state is feasible in practice. The constraint system of the correct behavior is a subset of all feasible program states. When we find a state outside of this subset (i.e., the negation), which is also feasible in practice, we found a violation of this behavior.

Comparison to Other Analysis Techniques The difference between analysis techniques is typically characterized by a trade-off between flagging more bugs, but at the same time introducing more false positives. For example cryptographic schemes [16] are a common occurrence on Ethereum. Assuming our example in Listing 1, where an attacker has to supply a correct pre-image for a keccak value. If an attacker knows the particular value, e.g., it is a publicly known value, they could bypass the check and destroy the contract. We assume in the general case that an attacker is oblivious to this value. However, approaches which overapproximate keccak computations, e.g., by simply assuming it could result in any value (see for example Securify or Vandal), flag the contract as vulnerable accordingly. Thus, these approaches might detect bugs "hidden" behind these code constructs. Yet, at the same time they burden an analyst with more false positives, resulting in wasting valuable audit time.

Scalability Similar arguments can be made for scalability: Again, assuming the keccak example. We encode these computations with our strict encoding scheme, which results in higher analysis time due to the added complexity. If we would simply assume that the computation could have any results, i.e., overapproximate it, this makes reasoning straight forward. This is demonstrated when examining the analysis time of tools like MadMax: While we solve about 80 % of all contracts in the first minute, these tools analyze about 90 % in the first 20 seconds. Similar performance is reported by a comparison conducted by Brent et al. [4] for Vandal, Mythril, and Oyente. However, this faster analysis comes at the cost of more false positives to evaluate. During a normal development cycle of a smart contract, where the developer quickly iterates over many versions of the contract, they could utilize "faster" tools. Finally, before deploying to the blockchain, a final precise analysis could be conducted using ETHBMC.

Impact Giving a fair assessment of the practical impact EthBMC could have is quite hard. Since the Ethereum system is fully transparent to the outside world, an attacker could monitor the blockchain and extract funds from the accounts when they contain an attractive amount of Ether. Thus, we performed an analysis of the highest value recorded for each vulnerable account we identified, giving us an upper bound on the potential impact. This yielded a maximum impact of around 155,000 Ether at risk. However, EthBMC can recreate the Parity hack. If the tool had been around at the time, we could have extracted more than 370,000 Ether. These equal about 40 Million USD and 89 Million USD, respectively, at the rate in the end of February 2020.

8 Related Work

Beyond the static analyzers discussed in Section 3.3, we now review other works closely related to ours. ZEUS [28] analyzes Solidity source code using abstract interpretation and deploys its own policy language, which can be used to specify violations to check against. In the same vein, VerX [51] is a recently proposed framework for verifying temporal properties. They utilize symbolic execution as well as abstract interpretation based predicate abstraction in conjunction with their own policy language to check these properties. However, since the source code of neither ZEUS nor Verx is available, we exclude them from our survey. Two other approaches for detecting vulnerabilities are Osiris [60] and EthRacer [31]. Osiris utilizes symbolic execution and taint tracking to discover integer overflow bugs. Osiris is built on top of Oyente, first analyzing contracts symbolically and afterwards utilizing taint tracking to check a source-sink pattern for integer overflows. EthRacer [31] is another approach to analyzing multi-transaction relationships. They focus on event ordering bugs, i.e., events which exhibit different behaviours when executed in different order. They utilize symbolic analysis to first extract happens-before relations [34]. Based on these findings, they perform fuzz testing to generate long chains of transactions searching for different outputs, thus, detecting event ordering bugs.

A different approach is taken by formal verification. Instead of checking a contract against a predefined set of bugs, the contract is validated against a handwritten formal specification. The K-Framework [27] provides full semantics for the EVM. These allow users to specify properties in reachability logic, which in turn gets checked against the formal semantics. Grishchenko et. al. [25] formalize the EVM semantics in in the F* proof assistance, also finding multiple flaws in existing verification tools for Ethereum smart contracts. Furthermore they define multiple security properties, which can be utilized while verifying one's contract.

Zhou et al. [65] introduce ERAYS, a reverse engineering tool for the EVM. They additionally conduct an analysis on function reuse in the Solidity ecosystem, finding that some functions reappear in over 10,000 contracts. Rodler et al. [53] utilize taint tracking to discover reentrancy attacks while executing smart contracts. In their setting, miners run an extended Ethereum node which protects against attacks at runtime.

9 Conclusion

In this paper, we first presented a survey of recent static analysis tools for smart contracts. We demonstrated that all of these tools employ imprecise reasoning in at least one category. Recognizing these flaws, we presented ETHBMC, a symbolic executor able to capture inter-contract relations, cryptographic hash functions, and memcopy-style operations. We demonstrated its effectiveness by evaluating the implementation against several previous works and showed that ETHBMC's accuracy significantly outperforms them. Additionally, we presented a vulnerability analysis of the current contract landscape, as well as multiple studies into the inner workings of ETHBMC.

Acknowledgements We would like to thank our shepherd Jelena Mirkovic, our colleagues Moritz Contag, Andre Pawlowski, Emre Güler, Ali Abbasi, Tim Blazytko, Moritz Schlögel, Thorsten Eisenhofer, Lukas Bernhard, and our anonymous reviewers for their valuable feedback. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972, and the German Federal Ministry of Education and Research (BMBF, project iBlockchain – 16KIS0901K).

References

[1] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *ACM Conference on Advances in Financial Technologies (AFT)*, 2019.

[2] Aaron R Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, 2007.

[3] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Static program analysis framework for ethereum smart contract bytecode. github.com/vandal.

[4] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[5] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. github.com/ethereum/whitepaper, 2014.

[6] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2008.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[8] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC)*, 2008.

[9] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM (CACM)*, 2013.

[10] Ran Canetti and Marc Fischlin. Universally composable commitments. In *Annual International Cryptology Conference*, 2001.

[11] Allen Day and Evgeny Medvedev. Ethereum in bigquery: a public dataset for smart contract analytics. cloud.google.com/ethereum-bigquery.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[13] Solidity Documentation. Solidity. solidity.readthedocs.io/overview, 2017.

[14] Solidity Documentation. Solidity in depth. solidity.readthedocs.io/indepth, 2017.

[15] Bruno Dutertre. Yices 2.2. In *International Confernce on Computer-Aided Verification (CAV)*, 2014.

[16] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[17] Stephan Falke, Florian Merz, and Carsten Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, 2013.

[18] Stephan Falke, Carsten Sinz, and Florian Merz. A theory of arrays with set and copy operations. In *SMT@ IJCAR*, 2012.

[19] David Floyd. The top 5 ethereum dapps by daily active users. coindesk.com/top-applications, 2018.

[20] Ethereum Foundation. Go-ethereum. github.com/go-ethereum, 2015.

[21] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *International Confernce on Computer-Aided Verification (CAV)*, 2007.

[22] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. github.com/MadMax.

[23] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2018.

[24] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[25] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *International Conference on Principles of Security and Trust*, 2018.

[26] Matthias Heizmann, Aina Niemetz, Giles Reger, and Tjark Weber. Smt-comp 2018. smt-comp.sourceforge.net/2018, 2018.

[27] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. In *IEEE Computer Security Foundations Symposium (CSF)*, 2017.

[28] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[29] Lucianna Kiffer, Dave Levin, and Alan Mislove. Analyzing ethereum's contract topology. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2018.

[30] James C King. Symbolic execution and program testing. *Communications of the ACM (CACM)*, 1976.

[31] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[32] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. github.com/teether.

[33] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security Symposium*, 2018.

[34] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 1978.

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. An analysis tool for smart contracts. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[36] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[37] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, Aquinas Hobor, and Melonport Security. Oyente sha3 computation. github.com/oyente, 2018.

[38] Microsoft. Microsoft and Bank of America Merrill Lynch collaborate to transform trade finance transacting with Azure Blockchain as a Service. *Microsoft News Center*, 2016.

[39] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[40] Bernhard Mueller. Mythril - security analysis tool for ethereum smart contracts. github.com/mythril, 2018.

[41] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. github.com/smashing-smart-contracts, 2018.

[42] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. www.bitcoin.org, 2008.

[43] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2015.

[44] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 2014.

[45] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Maian: Automatic tool for finding trace vulnerabilities in ethereum smart contracts. github.com/MAIAN.

[46] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[47] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Maian: Automatic tool for finding trace vulnerabilities in ethereum smart contracts. github.com/MAIAN/issues, 2018.

[48] Trail of Bits. Maticore symbolic execution tool. github.com/manticore, 2017.

[49] Trail of Bits. Not so-smart-contracts. github.com/trailofbits/not-so-smart-contracts, 2018.

[50] Santiago Palladino. The parity wallet hack explained. zeppelin.solutions/parity-wallet-hack, 2017.

[51] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[52] Nathaniel Popper and Steve Lohr. Blockchain: A better way to track pork chops, bonds, bad peanut butter? *New York Times*, 2017.

[53] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[54] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[55] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *International Conference on Systems Software Verification*, 2010.

[56] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[57] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.

[58] The Raiden Team. Raiden network. raiden.network.

[59] Parity Tech. A postmortem on the parity multi-sig library self-destruct. paritytech.io/postmortem, 2017.

[60] Christof Ferreira Torres, Julian Schütte, et al. Osiris: Hunting for integer bugs in ethereum smart contracts. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[61] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. github.com/securify.

[62] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[63] Oscar Williams-Grut. Goldman Sachs: 5 Practical Uses for Blockchain — from Airbnb to Stock Markets. *Business Insider*, 2016.

[64] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, eip-150 revision (commit 759dccd - 2017-08-07). github.com/ethereum/yellowpaper, 2014.

[65] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse engineering ethereum's opaque smart contracts. In *USENIX Security Symposium*, 2018.