An Extensive Security Analysis on Ethereum Smart Contracts

Mohammadreza Ashouri(✉)

Department of Cyber Security, Saint Pölten University of Applied Sciences, Saint Pölten, Austria
mohammadreza.ashouri@fhstp.ac.at

Abstract. Smart contracts have extensive applications in various emerging domains such as IoT, 5G networks, and finance. In this regard, the Ethereum platform has provided the capability of running smart contracts on its distributed infrastructure. Smart contracts are small programs that describe a set of rules for supervising associated funds, often written in a Turing-complete programming language called Solidity. Furthermore, Ethereum is currently one of the most extensive cryptocurrencies next to Bitcoin. This provides an extraordinary opportunity for attackers to exploit potential zero-day vulnerabilities in this ecosystem that are tightly twisted with financial gain. Consequently, this paper introduces a practical framework called "EthFuzz" to identify vulnerabilities and generate concrete exploits for the Ethereum ecosystem. Our system works through a graph-based method in combination with dynamic symbolic execution. Moreover, our proposed framework can tackle the path explosion problem in its symbolic execution engine. To prove our approach's usefulness, we could successfully identify and generate *26,015* exploits out of *207,412* exploitable paths, within *1,000,000* real-world smart contracts on the Ethereum live blockchain network.

Keywords: Smart contract · Security · Analysis · Ethereum · Exploit

1 Introduction

Blockchain is a distributed ledger technology that designates exchanges of value between individuals securely, permanently, and in a simply provable manner [19]. It is also the underlying technology for cryptocurrencies such as Bitcoin and Ethereum. Even though initially used for financial transactions, applications of blockchain extend beyond finance and can affect a wide variety of industries such as 5G and beyond networks [23]. For example, smart contracts can enable applications to communicate with Things in the IoT [16], in a way similar to how hardware drivers allow applications to cooperate with devices. Moreover, smart contracts can provide high security for the 5G networks involved in decentralized ledgers.

The programmability of the Ethereum platform is predicated on its ability to build and perform smart contracts [19]. The term "smart contract" was introduced by Nick Szabo in 1996 [29], when he described it as "a set of promises,

J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 398, pp. 144–163, 2021.
https://doi.org/10.1007/978-3-030-90019-9_8

specified in digital form, including protocols within which the parties perform on these promises". Smart contracts are agreements between transacting parties that are written using computer code and programmed to self-execute when specific conditions are met. These can be integrated into a blockchain platform like Ethereum to implement the verification and integrity required for such an automated system to work.

However, creating trustworthy and secure smart contracts can be remarkably complicated due to the complex semantics of the underlying domain-specific languages and their testability. There have been high-profile incidents suggesting that certain blockchain smart contracts could accommodate various code-security vulnerabilities which can potentially lead to financial harm [26]. This is especially challenging given the notion that smart contracts are supposed to be "immutable". In other words, once a contract code is deployed, it cannot be changed anymore, which makes patching identified vulnerabilities impossible.

In this paper, we introduce a practical and scalable framework for performing in-depth security analysis and automatic exploit generation for commercial off-the-shelf (COTS) smart contracts available on the blockchain network. We call our system "EthFuzz", and it can identify and exploit zero-day vulnerabilities, exploits and runtime attacks based on user specifications in Datalog [31]. Our approach works based on a backward slicing method to classify and control the safety of critical execution paths with a combination of static and dynamic analysis in lockstep with a symbolic execution engine.

We made the following contributions in this work:

1. **Generating concrete exploits.** EthFuzz automatically generates concrete exploits for detected vulnerabilities in smart contracts without access to the source code. Hence, we created a symbolic execution engine based on the Z3 SMT solver to trigger critical executable paths in given Ethereum smart contracts and create exploit inputs.
2. **The low cost of specifying new vulnerabilities.** While previous work [12, 15, 17, 30] has relied on literal hard-coded configurations, which produce a high maintenance cost and a high cost per vulnerability controlled, in EthFuzz's design, new vulnerability and attack patterns can be specified by the end users in Datalog files, provided by an auxiliary API in our framework. This allows users and developers to upgrade the framework for new attacking patterns without struggling with low-level structures and recompilation process.
3. **Controlling false positives.** In contrast to previous work, EthFuzz proactively separates exploitable from non-exploitable paths with the help of a dynamic execution module in order to prune useless paths from further analysis and symbolic execution operations. In consequence, the final result is more reliable and the exploit generation is faster and less error-prone.
4. **Real-world evaluation.** We have gathered and investigated COTS smart contracts derived from the Ethereum network to find current trends in security issues in the Ethereum ecosystem and regulate the effectiveness of EthFuzz for real-world applications.

2 Background

Smart contracts are only controlled by code that can handle transactions fully autonomously. Moreover, smart contract code is executed when a user submits a transaction along with a smart contract as the recipient. Users add payload data in transactions, which in turn is provided as input to the subject smart contract. More specifically, a contract is established as a collection of functions, which users can invoke. A contract can also trigger the execution of another contract through *CALL* instruction. This critical instruction transfers a message similar to Remote Procedure Call (RPC) in other programming paradigms [11].

In order to execute a smart contract, a sender has to send a transaction to the subject contract and pay a charge, which is called "GAS" (it will be acquired from the contract's computational cost.). The contracts themselves can also call other contracts present on the Ethereum blockchain [26]. Note that every contract is tied to an account and the contract code can be triggered by calls or transactions received from other contracts. However, accounts cannot launch new transactions on their own, which means they can only respond to other transactions they receive. Since smart contracts are generally designed to manipulate and hold funds designated in Ether, they are considered to be highly attractive targets for cybercriminals [27].

2.1 Smart Contract Vulnerabilities

There are multiple well-known security issues reported in the smart contract ecosystem that all have been comprehensively described in various references such as [7,26]. However, we briefly introduce some of the most prevalent vulnerability classes that we frequently mention throughout this paper. For the sake of saving space, we present a two-letter acronym for each vulnerability.

Integer Overflow (IO) and Underflow (IU). Integer overflow (and underflow) is a common error in numerous programming languages but in the context of Ethereum it can have serious outcomes. In Solidity "Integer" data types have no built-in security against integer overflow (IOF) and underflow (UOF) attacks [17,30]. For example, if a loop counter were to overflow, generating an infinite loop, the funds of a contract would become fully frozen. Thus, attackers can exploit this bug by increasing the number of iterations of a loop, for example, by introducing new users to a vulnerable contract [30].

Re-Entrancy (RE). This is a well-known attack that has taken Ethereum security communities by storm, particularly after the notorious DAO hack [28]. This vulnerability will be exploited when a contract attempts to send Ether before having updated its internal state. If the target address is a different contract, the contract code will be executed and can invoke the function to ask Ether again and again, which results in generating funds.

Unhandled Exceptions (UE). Some low-level operations in Solidity (e.g. *send*), which is used to transfer Ether, do not throw an exception on failure,

instead they report the status by returning a Boolean. If this returns value were to be unchecked, a contract would continue its execution even if the payment failed, which could lead to inconsistencies [22].

Transaction Order Dependency (TOD). In Ethereum, different transactions are carried in a single block, which means that the state of a contract can be updated many times in the same block. If the order of two transactions calling the same contract changes the final outcome, adversaries can exploit this property. For example, in the case of a smart contract that expects members to submit the resolution to a puzzle in exchange for a bonus, an adversary member could decrease the amount of the bonus when the transaction is submitted.

Locked Ether (LE). Ethereum smart contracts can also have a function labelled as payable that allows the contract to receive Ether and to increase its balance. The contract can also have a function which sends Ether. For example, a contract might have a payable function called *deposit*, which receives Ether, and a function called *withdraw*, which sends Ether. However, there are several reasons why the withdraw function may become unable to send funds any longer. One reason could be that the contract may depend on another contract which has been destructed using the *SELFDESTRUCT* instruction of the EVM—i.e. its code has been removed and its funds transferred. It is also possible that the withdraw function requires an external contract to send Ether. However, if the dependence contract has already been destructed, the *withdraw* function will not be able to actually send the Ether anymore and lock the funds of the contract. This case occurred in the Parity Wallet bug in November 2017, which resulted in a loss of millions of USD worth of Ether [25].

3 Security Analysis Method

Critical Operation. To have a better understanding of the security exploitation in the smart contract ecosystem, we studied all reports available on the National Vulnerability Database (NVD) in order to extract and specify the most critical EVM instructions that are commonly involved in cyber attacks. As a result, we concluded that there are a number of EVM instructions involved in most of the exploits that are essentially linked to value transformation operations. For example, creating transactions (*CALL*), transaction termination (*SELFDESTRUCT*), code injections (*CALL CODE*), and (*DELEGATECALL*) are some of the most repeated instructions that the public exploits databases. Listing 1.1 represents some of the critical instructions in the abstract level, and Table 1 shows the details of the instructions.

Listing 1.1. A sample generated exploit in Slick (for easier reading the sample contract is shown at the source level)

```
<address>.call(bytes memory) returns (bool, bytes memory)
<address>.delegatecall(bytes memory) returns (bool, bytes memory)
<address>.staticcall(bytes memory) returns (bool, bytes memory)
```

Accordingly, we found that smart contract attackers often steal Ether by exploiting these critical instructions or in some cases, they attempt to interrupt target contracts by triggering errors in the code logic. Consequently, to implement our security analysis mechanism, we are particularly interested in analyzing the runtime behavior of the EVM bytecode instructions associated with critical operations that can be potentially involved in suspicious activities during the code execution.

Table 1. The most critical instructions in the EVM bytecode

OPCODE	INSTRUCTION	DESCRIPTION
0x55	SSTORE	Save word to storage
0xe2	SSTOREBYTES	Only referenced in pyethereum
0xf1	CALL	Message-call into an account
0xf2	CALLCODE	Message-call into this account with alternative account's code
0xf3	RETURN	Halt execution returning output data
0xf4	DELEGATECALL	Message-call into this account with an alternative account's code
0xfa	STATICCALL	Similar to CALL, but does not modify state
0xff	SELFDESTRUCT	Halt execution and register account for later deletion

Accordingly, we aim to introduce an efficient analysis system that identifies not only zero-day vulnerabilities and unseen attacks but also generates reliable exploits the identified bugs without human intervention. Hence, we combine a hybrid approach based on "static call graph analysis", "dynamic execution", and symbolic execution in order to gain accurate results with maximum coverage.

Figure 1 represents the abstract architecture of our introduced approach, which is called "EthFuzz". To deploy our system without special firmware modifications or root privileges on different platforms, we implemented EthFuzz in an isolated and portable execution environment working on top of Ubuntu kernel 16.04 64 bit inside the QEMU emulator [10], which is a fast dynamic translator. Moreover, to evade accessing the contracts' source code for the analysis, EthFuzz functions based on the EVM bytecode instrumentation through leveraging "Parity Ethereum Client" [5]. This feature also enables us to perform bytecode instrumentation for real-world smart contracts.

As illustrated in the abstract of EthFuzz, our approach works based on the following three main stages:

1. Code Property Graphs Analysis
2. Dynamic Execution
3. Symbolic Testing

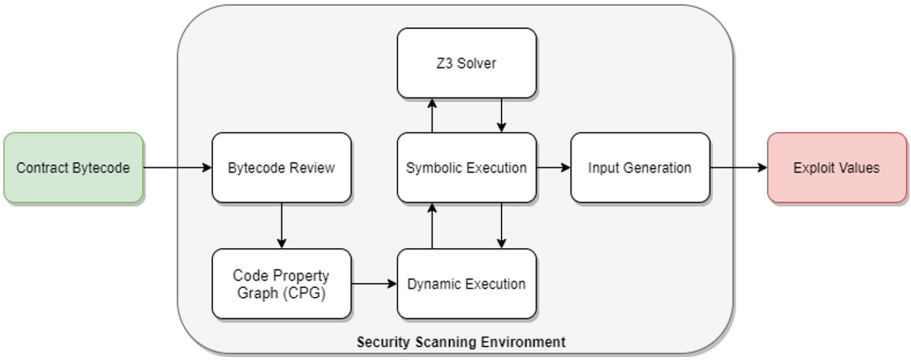

Fig. 1. The overview architecture of EthFuzz.

3.1 Step 1: Call Graph Analysis

In order to interpret and identify potentially exploitable execution paths within the bytecode of smart contracts, we generate a graph model of the target smart contract in a first step. Our graph model is based on *Code Property Graphs (CPGs)* [32], which is an extensible and language-agnostic representation of program code designed for incremental and distributed code analysis. The CPGs are constructed based on the EVM bytecode to help us to distinguish critical instructions and the relevant execution paths. Note that a critical instruction in a generated CPGs can be managed to find data dependency paths between the variables. After finding these paths, we later execute them symbolically to reproduce their corresponding exploits. In order to generate the call graph we used "Porosity" [2], which is an open-source tool. However, the available source code contained many bugs. For example, it stopped the call graph at *STOP* and *REVERT* instructions and did not treat *STATICALL* as call instructions. Porosity only recognized *JUMPI* as jump instruction and thus ignored *JUMP* instructions.

Backward Slicing. In order to generate concrete exploits, we also need to gain the correct entry point (we call it "source") so that a potential generated exploit can reach the critical instructions and perform an attack in the target contract successfully. To do so, EthFuzz performs a hybrid technique that comprises two steps, namely backward slicing" and non-exploitable path pruning (introduced in [6]), which is shown in Algorithm 1.

Algorithm 1: Performing backward-slicing to extract exploitable paths

```
   Input: Sensitive Instructions
   Result: Exploitable Paths
 1 initialization;
 2 SensitiveNodes ← FINSensitiveIntructionNode(SensitiveInstructions);
 3 foreach sn ∈ SensitiveNodes do
 4 |   ExploitableEVMPaths = ANALYZECRITICALNODE(sn);
 5 end
 6 returnExploitableEVMPaths;
 7 Function ANALYZECRITICALNODE(vertex):
 8 |   ExploitableEVMPaths ← [] ;
 9 |   paths = BackwardSLC(sn);
10 |   foreach path ∈ paths do
11 |   |   if pathhasasource then
12 |   |   |   ExploitableEVMPaths ← path;
13 |   |   else
14 |   |   |   callPaths = ANALYZECRITICALNODE(callVertex);
15 |   |   |   ExploitableEVMPaths ← path + callPaths;
16 |   |   end
17 |   end
18 |   return ExploitableEVMPaths;
19 Function BackwardSLC(vetex):
20 |   IntraPaths ← [] ;
21 |   while vertex is not a source vertex is not a func. argument do
22 |   |   Incvertices = GETINCOMINGDDVERTEX(vertex);
23 |   |   UnsanVertics = FILTERSANNVERTICS(Incvertices);
24 |   |   vertex ← unsanVertixs;
25 |   end
26 |   IntraPaths = GETPATHSTO(vertex);
27 |   return IntraPaths;
```

The backward-slicing algorithm starts by investigating the nodes (presenting EVM instructions) in the generated graph in order to draw critical instructions (line 2). For each node showing an instruction in the graph, EthFuzz explores its data dependency links in a backward way. *ANALYZECRITICALNODE* calls *BackSLC* in order to succeed all data dependency paths from an instruction node either to a source or a function argument. If the path drops at a function argument, *ANALYZECRITICALNODE* is called recursively over the points denoting the call-sites of that particular function. The function *BackSLC* then analyzes intra-procedural paths between sources and the critical nodes. It also controls safety functions (e.g., SafeMath in OpenZeppelin [3]) in the identified paths and prunes non-exploitable ones. Eventually, *GETPATHSTO* realizes all investigated paths in the graph leading to sources (entry points).

Pruning Non-exploitable Paths. To reduce the overhead caused by analyzing non-exploitable execution paths, we made an assumption. Suppose a detected path cannot reach a critical EVM instruction in the contract under analysis. In that case, we consider the path as a non-exploitable path, which must be excluded from further analysis because it may cause overhead and false-positive results. Algorithm 2 represents the details of the pruning method for non-exploitable paths.

Algorithm 2: Pruning non-exploitable paths

```
   Input: β: candidate path set, exposedCN: explosed critical node
   Result: α: set of paths after pruning
1  foreach p ∈ Path do
2      if isRelevant() == True then
3          α.push(P);
4      else
5  end
6  Function isExploitable(P):
7      if Const.solve() == ∅ then
8          return False;
9      else
10     if P.Succs ∩ exposedCN == ∅ then
11         return False;
12     else
13     return True;
```

3.2 Step 2: Dynamic Execution

In real-world smart contracts using off-the-shelf libraries for safety enhancement is quite common. For example, *Open Zeppelin SafeMath* is one of the popular libraries for protecting smart contract against Integer overflow/underflow attacks. In our approach, we also need to detect the presence of this type of protection in execution paths in order to reduce the potential of false-positive reports. To do so, we leverage dynamic execution to assess the exploitable target paths with actual runtime data. Thus, in the dynamic execution module, a special input or operation result, and a symbolic variable with a specific name (we call it "Taint Label") is added to the exploitable path. This symbolic variable is initially set by *0* and will be defined *TAINT-PC* (*PC* means program counter). We demonstrate this module in Fig. 2.

The taint label covers to succeeding branches along with the potential dangerous data, engaging in computations but without modifying the results. Our taint tracking method is described as follows:

1. Performing data flow analysis based on the propagation rules that describe which operations can propagate taint message or lead to new taint messages during the execution time (e.g. *ADD*, *SUB*, *MUL*.).
2. At some specific program points, security-critical parameters or state variables are supposed to be tainted or not on-demand, according to whether they carry taint labels.
3. If security-critical data of a risky point was tainted, the source entry can also be detected based on the tainted label.

3.3 Step 3: Symbolic Testing and Exploit Input Generation

After identifying exploitable paths, EthFuzz starts to generate concrete exploits for the paths. To avoid path explosion issues, we perform this stage with the help

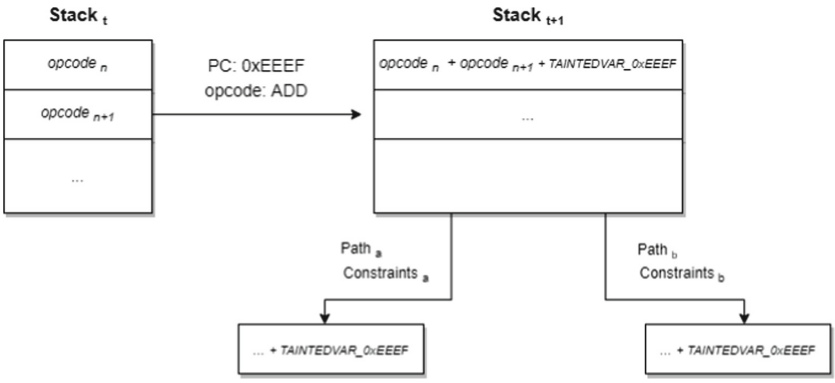

Fig. 2. Dynamic execution allows us to tracks all tainted inputs in the memory during the execution time.

of the collected actual and runtime data during the dynamic execution. Thus, our symbolic engine is not trapped in infinitive loops and infeasible paths. The exploit generation system operates based on a dynamic symbolic execution and Z3 SMT solver [14]. Consequently, we model the exploitable paths as a logical formula "F_{-Xpath}" so that their constraints are derived from the arguments of the extracted source "$F_{-Source}$" and critical instruction "F_{-Crins}" that represent which values after submitting to the target contract can lead to successful attacks.

As a result, the final formula is created as "$F_{-Xpath} \cap F_{-Source} \cap F_{-Crins}$" and will be sent to the Z3 SMT solver. The solver is responsible for executing the exploitable path symbolically and collects a set of path constraints to deliver the values (we call the values "payloads"). In the Z3 engine, we model the arguments of the call sites as "fixed-size" and "bit-vector" expressions. Moreover, we define the "variable-length" elements, such as the arguments by using the array expressions. The outcome of this modelling is actual practical exploits to trigger vulnerabilities inside target EVM bytecode.

Note that EthFuzz generates exploits on a single path first, before seeking more extensive path sequences. Due to the relatively small size of EVM smart contracts, EthFuzz explores path sequences up to length *10*, consisting of at most *8* state-changing paths and one last exploitable path. Listings 1.2 and 1.3 present a vulnerable contract and corresponding exploit generated by EthFuzz. As we stated before, we have implemented our symbolic execution engine based on the Z3 SMT solver.

4 Evaluation Result

We evaluated EthFuzz with *1,000,000* real-world smart contracts available on the Ethereum main blockchain. We collected these benchmarks from the beginning of October 2019 until the end of December 2019 with the help of

Listing 1.2. A generated exploit in EthFuzz in the source code level (for the ease of readers)

```
contract Overflow {
    uint private Balance=0;
    function add(uint value) returns (bool){
        Balance += value; // possible overflow vulnerability
    }
    function secure_add(uint value) returns (bool){
        require(value + Balance >= Balance);
        Balance += value;
    }}
```

Listing 1.3. An example of a EthFuzz's generated exploit for an overflow vulnerability

```
Location: from 22:27 to 22:46
2 numberTokens * COST_PER_TOKEN
3 --------------------
Transaction Sequence:
Tx #1:
Origin: 0xdeadbfefdeadbeefdeadbeefdeadbfefdeadbfee [ ATTACKER ]
Function: buy(uint256) [ d969094a ]
Data: 0xd969094a80100000000000000000000000000000000000000000
00000000000000000000
Value: 0x0
```

Etherscan [1]. The details of our collected corpus is shown in Table 2. Figures 3 and 4 also present the average Lines of Code (LoC) and number of functions, contracts and libraries in each categories respectively.

Table 2. Various contract categories in our corpus.

Category	SLOC	LLOC	CLOC
High ETH Moving	450	300	100
High Occurrence	195	150	50
High Interaction	390	225	40
High Origin	500	350	100
High Value	450	275	45

SLOC indicates Source lines of Code, LLOC represents Logical lines of Code, and CLOC represents Comments Line of Code. Our pre-analysis indicates that the average size of the EVM smart contracts on the blockchain is smaller than *400* LLOC. Furthermore, we represented the number of functions, contracts and

libraries in the corpus, based one *5* categories including "Higher Moving", "High Occurrence", "High Interaction", "High Origin", and "High Value".

Interestingly, the top *100* contracts (i.e. *0.1*% of the corpus) hold 98.86% of the total Ether value. Hence, this category of the contracts would be an attractive target for hackers (we label them *"High Value Targets"*). Likewise, we define another category as *"High Origin"* that comprises the top *100* contracts that impacted approximately *800k* other contracts in terms of bytecode similarity.

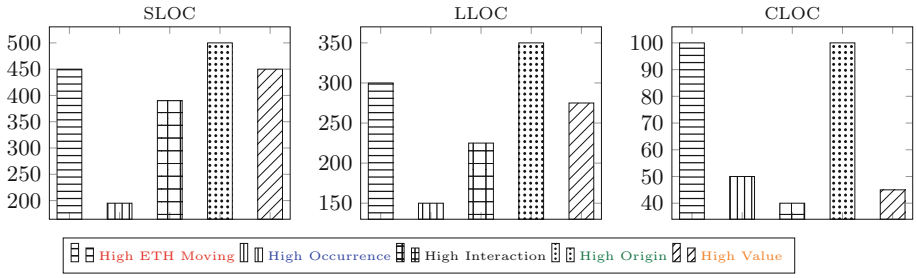

Fig. 3. Average number of Lines of Code (LoC)

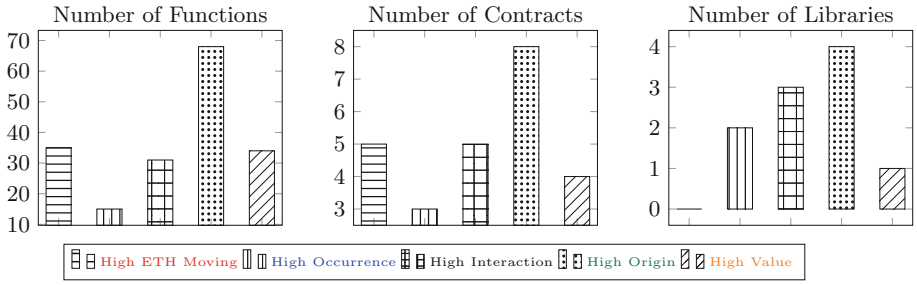

Fig. 4. Presenting the details of our benchmark suite based on the number of functions and libraries in the contracts

Moreover, we presented the top *10* most duplicated contracts in Table 3, and surprisingly one of the contracts, which is called *"User Wallet"*, has been deployed over *651K* times, which reveals the impact that the top *10* contracts have on the whole EVM smart contract blockchain network. Furthermore, only *2* contracts out of these top *10* most duplicated contracts actually have available and verified source code. This is highly surprising, since it raises the question of how a close-sourced contract can be replicated so many times? It seems these *2* contracts have been extensively used by a few certain companies active in the EVM blockchain.

Table 3. Top 10 most duplicated contracts in the corpus.

Address	Source code available	Frequency
2bf69ddcf80f6b24f2e6a8bf1454f662	✓	651930
fa00c5b8d83dbf920aec56d52c1df224	✗	158186
55f0329f9e5dbac461e933c66e0e29b5	✗	115132
dfcc91bcdc37abae7e8e9c82d57fbf6d	✓	99548
702edb219bba3238d55b2b38c759798b	✗	90489
923d7eaf6e90eb272493d3ca5c5859d5	✗	78018
7b63bae3ec81aa70d809a091240dccaa	✗	42868
62dbffb5cce3d14500568320ab6dcd75	✗	40456
1ae99eb3c89152c83cf788a5e7df4532	✗	37534
125fb7c1ad488e0d0b9b034cfd12a977	✗	28255

Experimental Setup. We presumed that the *storage* of each smart contract is initially empty, and that we can therefore handle duplicated contracts at the same time. We eventually made the experiment on an *8 Core Xeon W 3.2* GHz machine with *32* GB RAM running Ubuntu *16.04* LTS. In order to avoid any potential deadlock in analyzing a bytecode excessively, we dedicated *5* min as the maximum analysis time. We yielded this time constrain after multiple configurations of EthFuzz on *1000* arbitrary contracts to reach the maximum path coverage.

4.1 Analysis Results

Performing our in-depth security analysis on the collected corpus has taken approximately *60* days (from the beginning of January 2020 until the end of February 2020). Consequently, EthFuzz could successfully identify and generate *26,015* exploits within one million collected contracts. On the other hand, EthFuzz could not find even a single exploitable execution path inside *681,005* contracts. In other words, we could not detect any critical instructions (e.g., *CALL*, *SSTORE* and *DELEGATECALL*) in the bytecode of these contracts, so we would label them as "secure contracts".

Nevertheless, to check the accuracy of the results, we randomly picked *1,000* of these secure contracts, and we manually controlled their opcode with the aid of the Etherscan dissembler. This tool allowed us to convert the bytecode to the EVM assembly code. Thus, we found only *17* contracts (i.e. *1.7*%) actually contain *CALL* instructions, which are reported secure by EthFuzz mistakenly. This trivial false-negative issue occurred due to the time constraint we had set for the analysis (i.e., *5* min). Hence, by increasing the analysis time to *10* minutes, the issue could be resolved. We consider a *1.7*% false-negative rate for EthFuzz's precision for the strict time constraints. Table 4 presents the vulnerable smart contract with the most popularity on the blockchain network.

Table 4. Results of some vulnerable but popular smart contracts in our analysis.

Contract name	Vulnerability	Address
BeautyChain(BEC)	Integer Overflow	0xC5d105E63711398aF9bbff092d4B6769C82F793D
BlackJack	Bad Randomness	0xA65D59708838581520511d98fB8b5d1F76A96cad
CityMayor	Reentrancy	0x4bdDe1E9fbaeF2579dD63E2AbbF0BE445ab93F10
CNYToken	Integer Overflow	0x041b3eb05560ba2670def3cc5eec2aeef8e5d14b
CNYTokenPlus	Integer Overflow	0xfbb7b2295ab9f987a9f7bd5ba6c9de8ee762deb8
CryptoRoulette	Unitialized Storage Pointers	0x8685631276cFCf17a973d92f6DC11645E5158c0c
DAO	Reentrancy	0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413
EtherLotto	Bad Randomness	0xA11E4ed59dC94e69612f3111942626Ed513cB172
EtherPot	Unchecked External Call	0x539f2912831125c9B86451420Bc0D37b219587f9
Ethraffle v4b	Bad Randomness	0xcC88937F325d1C6B97da0AFDbb4cA542EFA70870
EthStick	Bad Randomness	0xbA6284cA128d72B25f1353FadD06Aa145D9095Af
FirePonzi	Typographical error	0x062524205cA7eCf27F4A851eDeC93C7aD72f427b
G-GAME	Unitialized Storage Pointers	0x3CAF97B4D97276d75185aaF1DCf3A2A8755AFe27
GGToken	Integer Overflow	0xf20b76ed9d5467fdcdc1444455e303257d2827c7
GoodFellas	Function Default Visibility	0x5E84C1A6E8b7cD42041004De5cD911d537C5C007
ICO	Transaction Order Dependance	0xd80cc3550Da18313aF09fbd35571084913Cd5246
KingofTheEtherThrone	Unchecked External Call	0xb336a86e2feb1e87a328fcb7dd4d04de3df254d0
LastIsMe	Transaction Order Dependance	0x5D9B8FA00C16BCafaE47Deed872E919C8F6535BF
Lottery	Bad Randomness	0x80ddae5251047d6CeB29765f38FED1C0013004b7
LuckyDoubler	Bad Randomness	0xF767fCA8e65d03fE16D4e38810f5E5376c3372A8
MESH	Integer Overflow	0x3AC6cb00f5a44712022a51fbace4C7497F56eE31
MTC	Integer Overflow	0x8febf7551eea6ce499f96537ae0e2075c5a7301a
OpenAddressLottery	Unitialized Storage Pointers	0x741F1923974464eFd0Aa70e77800BA5d9ed18902
Rubixi	Function Default Visibility	0xe82719202e5965Cf5D9B6673B7503a3b92DE20be
SMART	Integer Overflow	0x60be37dacb94748a12208a7ff298f6112365e31f

True and False Positives. In our evaluation, we classified and represented true and false positive results based on the attack type. Hence, we believe this helps developers and security experts to take these measurements into perspective in order to design secure test cases before releasing their contracts on the blockchain network. Figure 5 gives the precision of true and false positives in our analysis.

According to descriptions, classical vulnerabilities, e.g., *Integer Overflow* seem to be easier to approach by EVM programmers, particularly by using secure alternative Solidity libraries (e.g. *SafeMath*). However, the more complex operations exist in a code, the higher the chance of receiving intricate security issues in the contract. For example, understanding of some vulnerabilities such as *Re-Entrancy* might require a better comprehension of the Ethereum architecture, which is naturally less likely among junior developers. Consequently, this issue is one of the most common safety flaws in commercial smart contracts.

5 Exploit Generation Precision

In Table 5, we present the breakdown of exploit generation results. While we identified the majority of exploitable paths comprising *CALL* and *SELFDESTRUCT* critical instructions, only a small number of our generated exploits are based on *CALLCODE* and *DELEGATECALL* opcodes.

Exploit Verification. Considering that every contract account has its own storage that can alter the execution, we checked each exploit upon each concerned account separately. Accordingly, we built a new test Ethereum network including three contracts: a contract under analysis, a normal contract to represent an attacker, and a third contract to play the rule of a proxy for running specific exploits associated with *CALLCODE* and *DELEGATECALL* instructions.

Fig. 5. Distribution of true positives and false positives in each vulnerability class.

Table 5. Showing the break down of the critical paths in the generated exploits (left) and the precision of the generated exploits (right).

#Path type	#Contracts
Exploitable paths	207,412
Non-exploitable paths	701,924
Generated exploits	26,015

	#Exploits	Precision
TP Exploits	25,703	99%
FP. Exploits	312	1%
Total	26,015	

Note *CALL* is involved in a value transfer, *SELFDESTRUCT* is involved in contract termination. Moreover, *DELEGATECALL* and *CALLCODE* can allow for code injection

We also provided *100* Ether for the attacker account and *10* Ether for each target contract account. Then, we ran the test network in our simulation on top of the Ethereum Parity client [8], which enabled us to submit the generated exploit transactions to the test network. So, when an exploit succeeded, we submitted

its transaction to our test network. In order to avoid transaction reordering, we paused our testing for the miner to process each transaction before yielding the next one.

6 Comparison with Related Work

Security analysis for the smart contract ecosystem is continuously gaining attention of cybersecurity researchers [12,15,18]. In this respect, Oyente [4] is one of the pioneers in performing vulnerability detection that leverages symbolic execution testing for identifying bugs in the Ethereum smart contracts. Similarly, ZEUS [17] works based on a formal verification method to build and verify the correctness of security policies in the contracts.

S-gram [21] and Regaurd [20] both take smart contracts in solidity source code and report potential issues based on static predefined patterns. Securify (v1 and v2) [30] is another static analyzer that takes both source code and the EVM bytecode for performing analysis. Securify v2.0 has also provided Datalog interface to specify new vulnerabilities for the end users. However, both versions are unable to generate exploits.

Although our work was initially inspired by the tools mentioned above, our approach has multiple advantages over previous tools. In this regard, we conducted a comparison measure based on detection efficiency for real-world security analysis, the outcomes of the measurement are shown in Tables 6 and 7.

Table 6. A summary of 100k smart contract analyzed with different tools presented in related work. DSE means "dynamic symbolic execution", the full explanation of various attacks can be found in [26].

Tool	Attack classes	Analysis technique	Source/Bytecode
EthFuzz	20	Hybrid Analysis + DSE	Source + Bytecode
Remix-IDE	7	Static Analysis	Source
SmartCheck	14	Static Analysis	Source
Slither	15	Static Analysis	Source
Oyente	5	Symbolic Execution	Source +Bytecode
Securify	8	Symbolic Execution	Source +Bytecode
Mythril	10	Symbolic Execution	Source +Bytecode

Attack Specification. Because the previous tools detect security problems based on static collection patterns, they are often ineffective in identifying new vulnerabilities and zero-day attacks. EthFuzz, on the other hand, not only supports a wide range of attack patterns but also enables users to easily specify different patterns in the Datalog format.

AEG and Path Explosion. The previous tools do not support automatic exploit generation (AEG). This feature allows developers and contract owners to investigate the safety level of target contracts faster, and therefore, get better prepared to deal with future attacks (e.g., withdrawing their crypto assets). EthFuzz also identifies and prunes non-exploitable paths in the call graphs, thereby reducing analysis overhead and false-positive reports as well as minimizing the risk of a path/state explosion problem occuring during symbolic execution [9].

Table 7. Security issues checked by available tools

	Blockchain				EVM		Solidity											
Tools	TOD	Random number	Timestamp	Unpredictable state	Callstack depth	Lost Ether	Reentrancy	Unchecked call	x.origin	Blockhash	Send	Selfdestruct	Visibility	Unchecked math	Costly pattern	Bad coding pattern	Deprecated	Other
EthFuzz	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MAIAN	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Mythril	✓	✓	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗	✓	✗	✗	✓	✓	✓
Osiris	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Oyente	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Porosity	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Remix-IDE	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Securify	✓	✗	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
SmartCheck	✗	✗	✓	✓	✗	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓
Solgraph	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓
Vandal	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓

Compared to AEGs. Since EthFuzz is also an AEG tool for smart contracts, we selected MAIAN [24] and Teether [18] because they are the only available AEGs for smart contracts. Hence, we chose these tools as our baseline and compared the result of EthFuzz with them.

We applied MAIAN and Teether to *100,000* of the most popular contracts in our corpus with a timeout of 5 min for each contract. However, Teether and MAIAN could not analyze 11071 and 632 contracts, respectively. This is because of program crashes or timeout. As Table 1 shows, they generated 803 and 497 valid smart contract exploits, respectively. EthFuzz covers 1198 more exploits

than Teether and MAIAN in total. It seems that Teether and MAIAN cannot generate valid exploits for contracts.

Moreover, MAIAN is not designed for Code Injection attack; therefore, it missed that type of exploits. Furthermore, in MAIAN's attack model, attackers are not allowed to submit funds into the contracts when trying to find Balance Increment, which causes a loss of coverage. Teether also generates 81 false positives. When Teether tries to solve hash checks, it generates unmatched hash input and output, making the exploits invalid for 81 contracts. Second, different from our definition of Balance In-Crement exploitation, Teether reports exploits once a currency transfer is triggered. However, another 81 false-positive contracts set explicit checks to ensure the in-going fund is larger than the out-going funds. Although the attackers can trigger an outgoing currency transfer, their cost is more than the profit, which is not successful exploitation. Additionally, Teether crashes many times when proceeding contracts, which damage the overall performance as well. Even though EthFuzz produced only 29 false-positives cases in 100,000 contracts, in future work, we plan to extend the attack model to address the false positives. As for time consumption, EthFuzz generates about 25 test cases per second. For generated exploits in this experiment, EthFuzz spends 52 test cases on average, taking several seconds. However, Teether and MAIAN take several minutes to identify an exploit on average.

7 Challenges and Future Work

Since smart contract code is expected to be immutable after deployment and contract owners are anonymous, responsible disclosure is usually infeasible. Hence, dealing with vulnerable contracts seems to be tricky, and there appears to be no way of addressing the errors detected in already deployed contracts [26]. Therefore, the contract owners can only deprecate the vulnerable contract, move all funds out of the contract, use a new contract, and move the funds to the new contract, which is cumbersome since other contracts might reference the address of the vulnerable contract.

However, it seems that designing a runtime shield module in EthFuzz can help not protect in protecting contracts against various runtime attacks and vulnerabilities but also providing a layer of protection for the vulnerable contracts until the contract owners can preserve the funds and the contract developers find a way to fix the issues in the future forks.

8 Conclusion

In this paper, we introduced EthFuzz as a practical and portable security analysis and automatic exploit generation framework for the smart contracts ecosystem. Our approach takes into consideration the complexities of analyzing real-world commercial smart contracts and effectively addresses them. The novelty of the paper is to use an efficient analysis system that identifies zero-day vulnerabilities, unseen attacks, and generates reliable exploits the identified bugs without

human intervention by combining a hybrid approach based on "static call graph analysis", "dynamic execution", and symbolic execution in order to gain accurate results with maximum coverage.

Appendix A

Re-Entrancy Attack

Ethereum Virtual Machine (EVM) establishes a machine language called EVM bytecode, which includes approximately *150* opcodes [13]. Unlike memory, storage perseveres beyond the execution history of a contract. Indeed it is stored as a part of the global blockchain state. The EVM also states specific instructions to access transactions' fields, modify the contract's private storage, examine the current blockchain state, and even create additional transactions. It should be highlighted that the original Ethereum paper [28] differentiates between transactions, which are signed by regular accounts, and messages, which are not. Note that the EVM only implements integer arithmetic and cannot handle floating-point values.

In addition to the persistent storage and *256-bit* word stack, the EVM also executes a byte-addressable memory, which serves as an input and output buffer to different instructions. For instance, the *SHA3* instruction, which calculates a *Keccak-256* hash over variable-length data, reads its input from memory, where two stack arguments present both the memory location and length of the input. The memory content is not endured between contract executions, and it is invariably set to "zero" at the opening of each execution.

References

1. Bytecode to opcode disassembler – etherscan. https://etherscan.io/opcode-tool. Accessed 2 Feb 2020
2. Github - comaeio/porosity: *unmaintained* decompiler and security analysis tool for blockchain-based ethereum smart-contracts. https://github.com/comaeio/porosity. Accessed 7 May 2020
3. Openzeppelin/openzeppelin-contracts: Openzeppelin contracts is a library for secure smart contract development. https://github.com/OpenZeppelin/openzeppelin-contracts. Accessed 29 Jan 2021
4. Oyente. https://github.com/melonproject/oyente. Accessed 11 Aug 2019
5. Paritytech/parity-ethereum: The fast, light, and robust EVM and WASM client. https://github.com/paritytech/parity-ethereum. Accessed 2 July 2019
6. Ashouri, M.: Kaizen: a scalable concolic fuzzing tool for scala. In: Proceedings of the 11th ACM SIGPLAN International Symposium on Scala, pp. 25–32 (2020)
7. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
8. Parity Authors. Ethereum rust client (2017)

9. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. (CSUR) **51**(3), 1–39 (2018)
10. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, vol. 41, p. 46 (2005)
11. Birrell, A.D., Nelson, B.J.: Implementing remote procedure calls. In: Proceedings of the Ninth ACM Symposium on Operating Systems Principles, p. 3 (1983)
12. Brent, L.: Vandal: a scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981 (2018)
13. Dannen, C.: Introducing Ethereum and Solidity. Apress, Berkeley (2017). https://doi.org/10.1007/978-1-4842-2535-6
14. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
15. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 116 (2018)
16. Huh, S., Cho, S., Kim, S.: Managing IoT devices using blockchain platform. In: 2017 19th International Conference on Advanced Communication Technology (ICACT), pp. 464–467. IEEE (2017)
17. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: NDSS, pp. 1–12 (2018)
18. Krupp, J., Rossow, C.: Teether: gnawing at ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium (USENIX Security 2018), pp. 1317–1333 (2018)
19. Law, A.: Smart contracts and their application in supply chain management. Ph.D. thesis, Massachusetts Institute of Technology (2017)
20. Liu, C., et al.: ReGuard: finding reentrancy bugs in smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, pp. 65–68. ACM (2018)
21. Liu, H., Liu, C., Zhao, W., Jiang, Y., Sun, J.: S-gram: towards semantic-aware security auditing for ethereum smart contracts. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 814–819. ACM (2018)
22. Vivar, A.L., Castedo, A.T., Orozco, A.L.S., Villalba, L.J.G.: Smart contracts: a review of security threats alongside an analysis of existing solutions. Entropy **22**(2), 203 (2020)
23. Nguyen, D.C., Pathirana, P.N., Ding, M., Seneviratne, A.: Blockchain for 5G and beyond networks: a state of the art survey. arXiv preprint arXiv:1912.05062 (2019)
24. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 653–663 (2018)
25. Palladino, S.: The parity wallet hack explained, July 2017. https://blog.zeppelin.solutions
26. Perez, D., Livshits, B.: Smart contract vulnerabilities: does anyone care? arXiv preprint arXiv:1902.06710 (2019)
27. Qureshi, H.: A hacker stole 31 m of ether–how it happened, and what it means for ethereum. Freecodecamp.org, 20 July 2017
28. Sirer, E.G.: Thoughts on the DAO hack. Hacking 17 July 2016
29. Szabo, N.: Smart contracts: building blocks for digital markets. EXTROPY J. Transhumanist Thought **16**, 18:2 (1996)

30. Tsankov, P., et al.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82. ACM (2018)
31. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using datalog with binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005). https://doi.org/10.1007/11575467_8
32. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604. IEEE (2014)