EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode

Filippo Contro
Computer Science department
University of Verona
filippo.contro_01@studenti.univr.it

Marco Crosara
Computer Science department
University of Verona
marco.crosara@studenti.univr.it

Mariano Ceccato
Computer Science department
University of Verona
mariano.ceccato@univr.it
ORCID: 0000-0001-7325-0316

Mila Dalla Preda
Computer Science department
University of Verona
mila.dallapreda@univr.it
ORCID: 0000-0003-2761-4347

***Abstract*—Motivated by the immutable nature of Ethereum smart contracts and of their transactions, quite many approaches have been proposed to detect defects and security problems before smart contracts become persistent in the blockchain and they are granted control on substantial financial value.**

Because smart contracts source code might not be available, static analysis approaches mostly face the challenge of analysing compiled Ethereum bytecode, that is available directly from the official blockchain. However, due to the intrinsic complexity of Ethereum bytecode (especially in jump resolution), static analysis encounters significant obstacles that reduce the accuracy of exiting automated tools.

This paper presents a novel static analysis algorithm based on the symbolic execution of the Ethereum operand stack that allows us to resolve jumps in Ethereum bytecode and to construct an accurate control-flow graph (CFG) of the compiled smart contracts. EtherSolve is a prototype implementation of our approach. Experimental results on a significant set of real world Ethereum smart contracts show that EtherSolve improves the accuracy of the execrated CFGs with respect to the state of the art available approaches.

Many static analysis techniques are based on the CFG representation of the code and would therefore benefit from the accurate extraction of the CFG. For example, we implemented a simple extension of EtherSolve that allows to detect instances of the re-entrancy vulnerability.

***Index Terms*—Reverse engineering, Static analysis, Smart contract, Ethereum**

I. Introduction

Smart contracts are a recent extension to cryptocurrencies (e.g., Ethereum) that allows programs to be stored in the blockchain and to be executed by the distributed network of miners. Thus, a smart contract is a self-executing program that runs on a blockchain [1].

The most peculiar feature of a smart contract is that the program and all its transactions are immutable, i.e., once written to the distributed blockchain, they cannot be updated, even in case of programming defects identified after deployment [2]. Programming defects and vulnerabilities might result in frauds or in financial values of cryptocurrencies that are frozen forever [3]. Hence, code review, possibly supported by automated analysis tool, is crucial to detect programming errors and vulnerabilities before defective smart contracts are used or erroneous transactions are committed and permanently stored in the blockchain. This is especially critical for closed source smart contracts, whose source code can not be inspected by a client, and the only available representation is the compiled bytecode [4].

The accuracy of static analysis is one of the key points to promptly deploy and run correct smart contracts. However, tools from the state of the art for detecting programming defects and vulnerabilities in Ethereum smart contracts experience substantial limitations, and their results contain worrying amount of false positives (false alarms) and false negatives (overlooked problems) [5]. A potential explanation for such poor performance could be the intrinsic difficulty of analysing Ethereum bytecode. In fact, despite the bytecode is easy to parse (fixed length opcodes) [6] its semantics and control-flow graph (CFG) are difficult to reconstruct due to specific language design choices:

- Jump destination is not an opcode parameter, but a jump opcode assumes the destination address to be available on the stack, dynamically computed by previous code;
- There is no opcode for returning from functions: *return* is implemented by pushing the return address to the stack, and then performing a jump;
- Functions are removed by the compiler. Intra-contract function calls are replaced by jumps. Inter-contract function calls are resolved by a *dispatcher* at the smart contract entry point, that decides what address to jump to, depending on the call's actual parameters;
- The smart contract constructor is executed only once, when the contract is initially deployed in the blockchain and then discarded. Thus, the constructor bytecode is not available in the blockchain [7].

The accurate extraction of the CFG from the bytecode is at the basis of the success of many static analysis algorithms. In this paper we focus on the extraction of the CFG from the Ethereum bytecode. To this end, we propose a static analysis approach, called *symbolic stack execution*, that resolves jump destinations based on the symbolic execution of the operand stack. After jump destinations are resolved, an accurate CFG can be built. This approach has been implemented in EtherSolve[1] [8], [9]: a fully automated tool that provides a beneficial starting point for any sophisticated static analysis meant

[1]The tool is available on github.com/SeUniVr/EtherSolve/tree/ICPC-2021

to identify programming defects or vulnerabilities. Indeed, starting from the results of EtherSolve, we implemented a component to detect occurrences of re-entrancy vulnerability in Ethereum smart contracts. This detector turns out to be comparable or superior to state-of-the-art security scanning tools.

The paper is structured as follows. After covering the background of smart contracts in Section II, the different phases of our analysis are described in Section III. Section IV presents our empirical validation and comparison with state of the art tools. Then, Section V discusses related work and Section VI closes the paper.

II. Background

Ethereum is a global, open-source platform for decentralised applications, based on a blockchain technology. On Ethereum network, it is possible to write simple programs, called *smart contracts* [1], [10], [11], that control the cryptocurrency called Ether (ETH) [12]. The actions that can be performed in Ethereum are basically transactions, i.e. movements of funds or data between different accounts. Every new transaction is irreversible and it is permanently added in a new *block* that updates the blockchain [1], [13].

A. Solidity Language

```
pragma solidity ^0.6.0;
contract SimpleBank {
    mapping(address => uint256) private balances;
    function deposit(uint256 amount) public payable{
        require(msg.value == amount);
        balances[msg.sender] += amount;
    }
    function deposit100() public payable{
        require(msg.value == 100);
        balances[msg.sender] += 100;
    }
    function withdraw(uint256 amount) public{
        require(amount <= balances[msg.sender]);
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
```

Listing 1: Solidity code example

Solidity is the most widely spread programming language for Ethereum: it is an object-oriented, high-level and Turing-complete language for implementing smart contracts [10], [14].

Solidity contracts are basically objects with functions and fields. The example in Listing 1 reports a smart contract written in Solidity to implement a bank. The field *balances* stores the internal state of the smart contract. It is a key-value map that associates every address to an integer value that represents the funds own by the address. The functions *deposit* and *deposit100* allow the user to deposit currency into its virtual account. The former allows to deposit an arbitrary amount, the latter is a special case which allows to transfer exactly 100 Wei (10^{-16} Ether). The *withdraw* function allows the user to get back a certain amount of Ether previously deposited. Solidity provides different primitives to interact with the blockchain environment: *transfer* sends Ether to a certain address, *revert* makes the transaction fail and roll back to the state prior to the transaction, *require* enforces a certain boolean condition and in case the condition is not met it performs a *revert*, and many others not shown in the example.

B. Compiling Solidity into Ethereum Bytecode

Before running a smart contract in the Ethereum blockchain, Solidity source code needs to be compiled to Ethereum bytecode to be executed by the Ethereum Virtual Machine (EVM).

Constructor. The Solidity compiler, namely *solc*, generates the *creation code*. This is the constructor of the smart contract that performs the initial operations and deploys the *runtime code* on the blockchain; the constructor code is then discarded and not stored in the blockchain [7], [15].

Runtime Code. The runtime code can be divided into three main segments. The first segment contains the opcodes that the EVM executes; the second one is optional and contains static data (e.g., strings or constant arrays); the last segment contains the metadata, such as compiler version and different hashes of the code. The structure of metadata has been continuously changing through the different versions of the Solidity compiler.

Application Binary Interface. The Solidity compiler emits also the Application Binary Interface (ABI). This file contains the list of the functions in the smart contracts that can be called by a user, together with type and number of parameters. Functions are not identified by their name but by the hash of the signature. The ABI file is not deployed in the blockchain and it needs to be distributed separately as it contains the main information for interacting with a smart contract.

Stack. The stack is the main memory of a smart contract, it is a volatile LIFO queue with 1024 blocks of 32 bytes [10], [15]. The execution relies heavily on it, as arithmetic and logic operations follow the reverse polish notation, where the data are loaded into the stack before the operation [16]. For instance, the bytecode `6005600301` is parsed into `PUSH1 0x05 PUSH1 0x03 ADD`, and the EVM execution will (i) push a byte to the stack containing the value `0x05`; (ii) push the value `0x03` and then (iii) execute the addition operation, which consumes two elements from the stack and leaves their sum as result, leaving the final stack with only the value `0x08`.

Opcodes. The complete list of opcodes with their semantics is defined in Ethereum's yellow paper [6], and there can be little variations among different EVM versions.

```
Runtime Code:
608060405260043610610034576000 3560e01c8063140e9ac714
610039 ... 6000206000828254019250508190555050 56fe
Metadata:
a2646970667358221220e62b6e0d256ecbc0a1b39b99bf0a2b50
9ed60dd83c71541b2d00fed1bde5a9e464736f6c634300060b00
33
```

Listing 2: Bytecode example

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE
LT PUSH2 0x34 JUMPI PUSH1 0x0 CALLDATALOAD PUSH1
0xE0 SHR DUP1 PUSH4 0x140E9AC7 EQ PUSH2 0x39 JUMPI
...
```

```
PUSH1 0x0 KECCAK256 PUSH1 0x0 DUP3 DUP3 SLOAD ADD
SWAP3 POP POP DUP2 SWAP1 SSTORE POP POP JUMP INVALID
```

Listing 3: Opcodes example

Listing 2 shows a portion of the Solidity smart contract compiled into bytecode, while Listing 3 shows the translation of the bytes into EVM opcodes. Ethereum bytecode can be easily parsed into opcodes, which are the minimum instructions that the EVM can execute and are identified with a byte.

Every opcode pushes or pops a certain number of elements from/to the stack, and it can either access memory, get information about the execution environment or interact with other blockchain smart contracts. The only opcodes with a parameter are those in the `PUSH` family: the value that the EVM pushes into the stack is taken directly from the bytes following the opcode. There are different variants of `PUSH`, depending on the number of bytes that needs to be pushed to the stack, varying from `PUSH1` (1 byte is pushed) to `PUSH32` (32 bytes are pushed) [6], [15].

A portion of the code can be used as read-only data; in fact with the `CODECOPY` opcode the execution can copy a portion of the code to the memory and then treat it as data. [17]. Thus, parsing this segment of memory as code might generate spurious results, including invalid opcodes and wrong jump destinations.

Control flow opcodes. The control flow of the program is managed through the stack. In fact, in order to jump among different portions of the code both the `JUMP` and the `JUMPI` opcodes (unconditional and conditional jumps) read the jump destination from the stack. These destinations are not managed with labels, but with the offset of the next instruction in the code [17]. Unlike x86 assembly, in the EVM there is no concept of *function*: everything is managed through jumps, and there are neither opcodes for function call nor for return from function call. The only return available is for function calls coming from external smart contracts.

These design decisions make the EVM bytecode difficult to analyse statically. In particular, since jump destination are computed at run time, the CFG cannot be reconstructed without a sort of stack simulation whose precision directly affects the accuracy of extracted CFG.

Dispatcher. When a transaction starts the execution of a smart contract, it can send both funds and information as *call data*. In order to transfer the control to the code corresponding to the intended function, the compiler adds a *dispatcher* at the beginning of the contract code. When a caller is willing to execute a certain contract function, it sends a transaction that contains the hash of the function signature, so that the dispatcher can compare it with all the hashes of the smart contract functions and then take the execution to the begin of the corresponding function code. Instead, if no call data are supplied or none of the hashes matches, the dispatcher takes the execution to the beginning of the *fallback function*. This function has no parameter and returns no value [10].

Every function call in the Solidity contract is translated by the compiler into a sequence of `PUSH` opcodes, followed by a `JUMP` and a `JUMPDEST`. This sequence loads into the stack the return address of the calling context, the (optional) actual parameters and the address of the function to call. Then `JUMP` executes the function body, that eventually consumes the parameters from the stack, leaving the return address which, once executed, brings the execution to the `JUMPDEST`, resulting in an actual return statement. In the following sections we provide a simple example of this pattern.

C. Re-entrancy Vulnerability

Re-entrancy is one of the most prominent vulnerabilities in Solidity, because it was exploited by the infamous DAO incident [3] that caused serious consequences to the whole Ethereum network. This vulnerability consists in re-entering a paying function multiple times while the contract is in an inconsistent state, thus causing possible leak of funds [5]. This vulnerability might be present when a contract state update follows (instead of preceding) a fund send primitive (i.e., a `call`). An example of vulnerable contract is shown in Listing 4, where the `call` statement at line 7 precedes the update of variable `bank` at line 8.

```
1  contract Bank {
2      // Mapping of money owned by an address
3      mapping (address => uint) bank;
4      // Function to withdraw money
5      function withdraw (uint amount) public {
6          require(amount <= bank[msg.sender]);
7          if (msg.sender.call.value(amount)(""))
8              bank[msg.sender] -= amount;
9      }
10 }
```

Listing 4: Re-entrancy example

III. Static Analysis

EtherSolve [8] aims at extracting the CFG from the bytecode of Ethereum smart contracts. The CFG is a directed graph representing the flow of execution: nodes are the program's basic blocks (sequence of opcodes with no jumps, other than in the last opcode) and edges connect potential successive basic blocks.

A. Approach Overview

The static analysis implemented in our approach is composed of the following steps:

- *Bytecode parsing:* The binary representation of the Ethereum bytecode is split between code and metadata; the code is then parsed to identify opcodes;
- *Basic blocks identification:* Opcodes are grouped in basic blocks and the easier jumps between basic blocks are resolved;
- *Symbolic stack execution:* The execution stack is subject to symbolic execution in order to resolve the more difficult jump destinations;
- *Static data separation:* The static data segment is separated from the actual executable code;
- *CFG decoration*: The so obtained CFG is decorated to highlight the dispatcher and to identify the entry point of the fallback function.

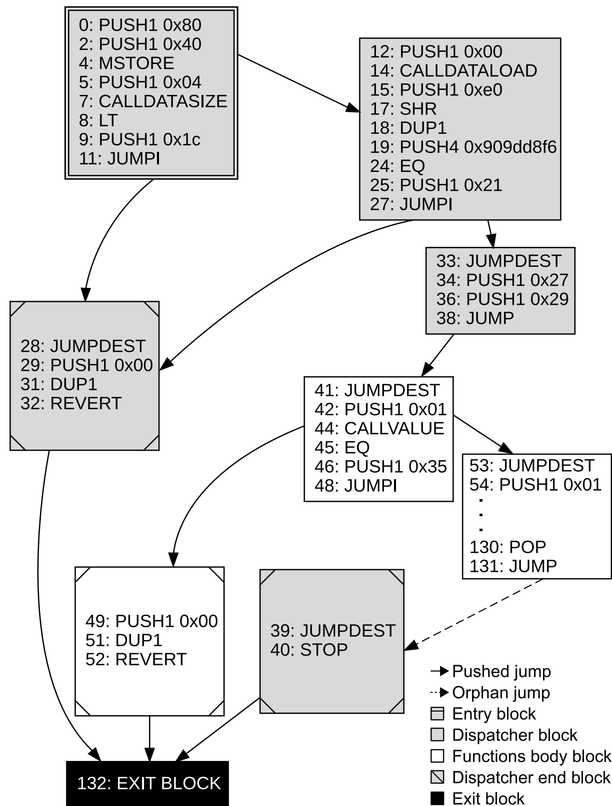

Fig. 1: Example of control-flow graph.

In the following, we describe these steps in details referring to the *deposit100* function of the *SimpleBank* smart contract in Listing 1.

B. Bytecode Parsing

The analysis starts with the raw bytecode. The metadata section is identified by finding the respective header reported in the official documentation [10]. In case of metadata with *experimental features*, the header is different and not documented. We inferred the header structure of non documented experimental cases by manually inspecting the bytecode of some contracts. The version of the Solidity compiler used to compile the bytecode is extracted from the metadata.

The metadata are then dropped and the remaining bytes are considered as actual code and parsed by the EtherSolve parser module. An example of how bytecode is parsed into opcodes is shown in Listing 2 and Listing 3, where every two characters of the bytecode are translated into the respective opcode (e.g. `0x6080` becomes `PUSH1 0x80` and so on). Each opcode is univocally identified by its offset address.

C. Basic Block Identification and Pushed Jumps

A basic block is a sequence of opcodes which are executed consecutively between a jump target and a jump instruction, without any other instruction that alters the flow of control. Thus, opcodes that alter the control flow of the program divide the code into basic blocks. Opcodes `JUMP`, `JUMPI`, `STOP`, `REVERT`, `RETURN`, `INVALID`, `SELFDESTRUCT` mark the end of a basic block, whereas `JUMPDEST` marks the beginning of a new basic block. Every basic block is uniquely identified by its offset, i.e., the position of its first opcode in the bytecode. In Fig. 1 we can see the basic blocks of the code in Listing 3 extracted following this procedure. Indeed, each basic block either starts with a `JUMPDEST` or ends with an opcode which alters the control flow.

Once the code is divided into basic blocks, we proceed with the computation of the edges. This operation is not always simple as the jump destination is not an opcode parameter but it is available on top of the stack at execution time. We identified two types of jumps: *pushed jumps* and *orphan jumps*. A *pushed jump* is a `JUMP` immediately preceded by a `PUSH` opcode, so that its target is easy to resolve, just by looking at the value in the preceding `PUSH` opcode. *Orphan jumps* instead are not preceded by a `PUSH` and their target is not immediate to compute. In the example CFG of Fig. 1 the block 53 ends with an *orphan jump* whereas the remaining jumps are *pushed jumps*.

We start by computing the edges of *pushed jumps*. To this end, each basic block is analysed according to its last opcode:

- `JUMP` preceded by a `PUSH`: the argument of the push is the destination offset of the jump and the corresponding edge is added to the CFG.
- `JUMPI` preceded by a `PUSH`: the false branch goes to the following block (in offset order), the true branch is the argument of the push interpreted as destination offset for the `JUMPI`. In this case the two corresponding edges are added to the CFG.
- `JUMP` not immediately preceded by a `PUSH`: the resolution of the jump is not trivial and it needs to be resolved through symbolic stack execution, described in III-D.
- `REVERT`, `SELFDESTRUCT`, `RETURN`, `INVALID`, `STOP`: there are no successors as the control flow is interrupted.
- In any other case the control flow proceeds with the basic block in the next position in the bytecode.

At the end of this phase we have extracted a partial CFG where the edges related to *orphan jumps* are still unresolved. For example, the extraction of the CFG of the code in Listing 1 at this point is depicted by the basic blocks and continuous edges of the CFG in Fig. 1, while the outgoing edge from the basic block 53 has not been resolved yet.

D. Symbolic Stack Execution and Orphan Jumps

The most challenging step in the CFG construction is the resolution of the destinations of *orphan jumps*. These jumps are very common: for instance the Solidity compiler uses them on return from function call. Indeed, between the function entry point and the function exit point (i.e., the return), the stack is heavily used by the function body to implement all the desired features (arithmetic operations, calls to other functions, transfer of funds).

The analysis consists in executing the stack symbolically: the algorithm walks the partially built CFG executing only the opcodes that interact with the jump addresses, updating the

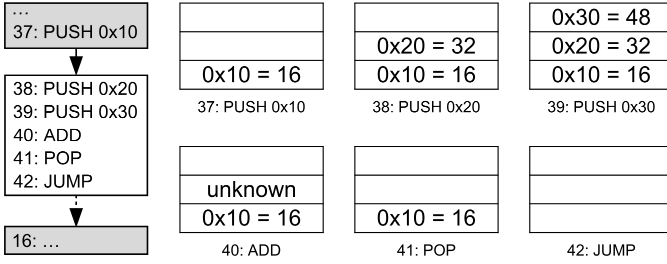

Fig. 2: Symbolic stack execution

state of the stack accordingly, in such a way that the *orphan jump* destinations can be found on the symbolic stack. Indeed, the symbolic stack execution considers only the opcodes in the `PUSH`, `DUP` and `SWAP` families, together with the `AND` and `POP` opcodes. For every other opcodes the symbolic stack pops and pushes "unknown" elements, as they do not deal with the jump addresses.

In the example shown in Fig. 2 there is a simple piece of code that has been executed symbolically to highlight the procedure. In particular the `ADD` is not modelled in full detail, but it simply consumes two elements and then generates a single "unknown" value. The jump address instead is loaded before the arithmetic operations, but it persists until the actual `JUMP` opcode, so it can be resolved.

The symbolic stack execution handles the opcodes according to the rules represented in the look-up Table I, where the following notation is used:

- S: stack that can contain numeric values or "unknown" to represent unknown values. We indicate the top of the stack as position 0.
- $\alpha : Opcodes \rightarrow \mathbb{N}$: function that associates to each opcode the number of elements added to the stack.
- $\delta : Opcodes \rightarrow \mathbb{N}$: function that associates to each opcode the number of elements consumed from the stack.

The algorithm walks through the CFG using a DFS (Depth-First Search) keeping a state of the stack for each basic block. The following constraints have been enforced in order to avoid infinite loops: an edge cannot be analysed more than once with the same symbolic stack state, and there is a limit on the number of elements to compare when checking for stack equivalence.

Another important aspect is the fact that a function can be called in different points in the code, resulting in different symbolic stacks and different paths on the CFG. In order to avoid infeasible paths (paths that real executions would never traverse) when the DFS visit encounters a basic block ending with a `JUMP` only its destination block, obtained by the symbolic stack, is added to the DFS queue.

The detailed algorithm for solving orphan jumps is shown in Algorithm 1. It starts at Line 2 by initialising variable V that stores the edges that have already been analysed using stack equivalence as described before (lines 20 and 27); an edge is also labelled with the symbolic stack that has been used for its symbolic execution (Cf. lines 19 and 26). Then, the queue Q

TABLE I: Look-up table for *executeOpcode* in Algorithm 1

Value	Name	δ	α	Stack
0x16	AND	2	1	$S[1] = S[1] \wedge S[0]; S.pop()$
0x50	POP	1	0	$S.pop()$
0x60 ⋮ 0x7f	PUSH1 PUSHn PUSH32	0	1	$S.push(opcode.$argument$)$ $\lvert opcode.$argument$\rvert = n$ bytes
0x80 ⋮ 0x0f	DUP1 DUPn DUP16	n	$n+1$	$S.push(S[n-1])$
0x90 ⋮ 0x9f	SWAP1 SWAPn SWAP16	$n+1$	$n+1$	$S[0], S[n] = S[n], S[0]$
0x*	O	$\delta(O)$	$\alpha(O)$	$S.pop()$ $\delta(O)$ times $S.push($unknown$)$ $\alpha(O)$ times

used for the DFS is initialised at Line 5: it contains pairs with a block and a symbolic stack. The first pair $\langle CB, S\rangle$ contains the first block and an empty stack. Then, the algorithm proceeds with the symbolic stack execution by iteratively repeating the following steps until Q is empty:

- Lines 9-11: symbolically execute the opcodes of the basic block and update the state of the symbolic stack according to the look-up Table I.
- Lines 12-16: resolution of the *orphan jump* destination with the newly updated symbolic stack. The target block is added as a successor of the basic block under analysis.
- Lines 17-31: handle the update of the queue Q. If the edge from the analysed basic block to the target one has not been already analysed with the same stack then the successor blocks are added to Q. If the last opcode is a `JUMP` then only its target block is added to Q.

An example of the symbolic stack execution for the resolution of *orphan jumps* is shown in Fig. 3 and refers to a portion of the program shown in Listing 3, whose CFG is depicted in Fig. 1. The symbolic execution starts at the offset 36, which loads into the stack the value `0x29` after the value `0x27`. Then, our approach symbolically executes the `JUMP` opcode that, according to Table I, consumes a value. Next, the symbolic execution of `JUMPDEST` leaves the stack unchanged and then value `0x01` is loaded. Then, the execution proceeds until the opcode at 129 following the look-up table, which leaves an unknown value on the stack that is removed by the `POP`. Finally, the opcode at 131 contains the *orphan jump*, which can be resolved with the value pushed into the stack back on offset 34. At this point the symbolic stack execution can detect that the successor basic block is the number 39.

Eventually, we have resolved the target of all branches in the CFG, so the dashed edge in Fig. 1 is added at the end of this phase.

Algorithm 1 Resolve Orphan Jumps

```
 1: function RESOLVEORPHANJUMPS(basicBlocks)
 2:     V ← Set()                                   ▷ Visited
 3:     CB ← basicBlocks.first                      ▷ Current block
 4:     S ← SymbolicExecutionStack()                ▷ Stack
 5:     Q ← Stack()                                 ▷ DFS queue
 6:     Q.push(⟨CB, S⟩)                             ▷ DFS first element
 7:     while Q ≠ ∅ do
 8:         CB, S ← Q.pop()
 9:         for op ∈ CB.opcodes do
10:             S.executeOpcode(op)                 ▷ with look-up table
11:         end for
12:         if CB.opcodes.last == JUMP then
13:             NO ← S.peek                         ▷ get next offset from stack
14:             NB = basicBlocks[NO]                ▷ Next block
15:             CB.addSuccessor(NB)
16:         end if
17:         if CB.opcodes.last ≠ JUMP then
18:             for suc ∈ CB.successors do
19:                 edge ← ⟨CB.offset, suc.offset, S⟩
20:                 if edge ∉ V then
21:                     V.add(edge)
22:                     Q.push⟨suc, S⟩
23:                 end if
24:             end for
25:         else if CB.opcodes.last == JUMP then
26:             edge ← ⟨CB.offset, NO, S⟩
27:             if edge ∉ V then
28:                 V.add(edge)
29:                 Q.push(⟨NB, S⟩)
30:             end if
31:         end if
32:     end while
33: end function
```

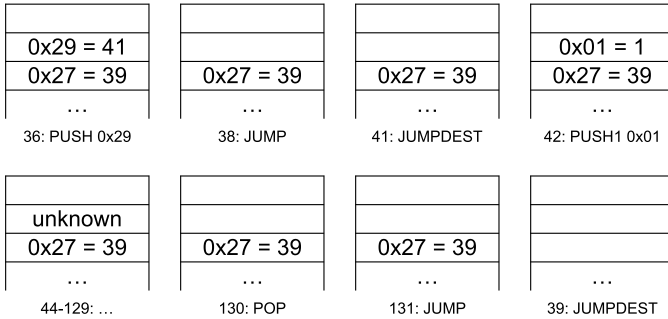

Fig. 3: Symbolic stack execution for orphan jump resolution

E. Static Data Separation

The proposed approach proceeds with the removal of static data (if present). EtherSolve searches for the first basic block containing the instruction `0xFE`, which is the designated opcode for an invalid instruction. In fact, the Solidity compiler uses this opcode to mark the end of the executable code section and the beginning of the static data section. Thus, all the opcodes from here on are removed from the graph, and considered as static data and not as code. Then, the algorithm proceeds by removing from the graph any basic block that is not connected to the main graph, if any.

The static data detected are usually strings contained in the source code or child contracts which are instantiated by the main one through the opcodes `CREATE` and `CALL`. Even if in principle we cannot be sure that the removed data is actually data and not code, our experimental validation shows that our approximation is very accurate.

F. CFG Decoration

In order to attach more information to the CFG that could be relevant to an analyst or for a subsequent static analysis (e.g., for vulnerability detection), EtherSolve tries to highlight some relevant portions, such as the dispatcher, the fallback function and the last basic block.

The dispatcher is the entry point of the smart contract, so it is at the beginning of the bytecode. The dispatcher directs the execution to the intended Solidity function and it manages parameters and return values. The fact that the dispatcher manages the return values is the key used for its detection. In fact, the only basic blocks that contain instructions such as `RETURN` and `STOP` are part of the dispatcher. This opcodes cannot be present in other locations as they would manage return values outside the dispatcher. Hence, the algorithm considers as dispatcher every basic block with an address lower then the address of these opcodes. In the example of Fig. 1, the dispatcher blocks are highlighted in grey.

This approach is effective to identify both the *linear* dispatcher, used in the older versions of the Solidity compiler, and the *tree* dispatcher, introduced in the latest versions of Solidity in order to improve the dispatcher performance.

The detection of the fallback function entry point is more difficult, because its structure has been changing continuously across different versions of the Solidity compiler. The first check of the dispatcher is the presence of call data and, if missing, it moves the execution to the Fallback function. Hence, the currently implemented technique starts from the entry block searching for the highest offset successor; then its successor with the highest offset is considered fallback only if it does not end with a `REVERT`, otherwise that would mean that the fallback function has not been declared or has been declared with only the `revert()` statement. This approach however does not work with some versions of *solc* due to different compilation patterns.

The last step of the CFG decoration is the addition of an artificial unique exit point, for all the basic blocks with no successor. This could be useful for any static analysis that applies afterwards. This particular basic block in the example in Fig. 1 is the number 132.

G. Re-entrancy Detection

Ethersolve has been pipelined with a subsequent static analysis meant to detect cases of the re-entrancy vulnerability. The proposed approach is a scan of the CFG in order to detect potential flows of execution where a `SSTORE` opcode (which updates the contract state) is executed after a `CALL` opcode. This pattern is considered unsafe if the contract address where funds are transferred to by the `CALL` cannot be statically determined by the symbolic stack execution. Indeed, in this

case the funds destination could be controlled by an attacker who mounts an attack to exploit a re-entrancy vulnerability.

IV. Experimental Validation

In this section we present the results of our empirical validation of the accuracy of the CFG computed by EtherSolve. Three research questions guide the definition of our experimental validation:

- **RQ$_1$**: What is the success rate of EtherSolve when analysing real-world smart contracts compiled with different versions of the Solidity compiler?
- **RQ$_2$**: How does EtherSolve compare with existing approaches?
- **RQ$_3$**: How precise is the re-entrancy vulnerability detection built on top of EtherSolve?

The first research question investigates the extent to which EtherSolve can process instances of real smart contracts with no errors. We are interested in verifying this on a wide range of smart contracts directly taken from the blockchain. The second one compares our approach with the state of the art. The third research question compares the results of the re-entrancy vulnerability detection based on EtherSolve with other existing vulnerability detection tools.

A. Dataset

The empirical validation has been conducted using a dataset of smart contracts obtained from the list of verified contracts from Etherscan[2]. They are publicly available open source smart contracts with information about compilation, deployment and transactions. From this list we have randomly extracted 1000 contracts[3]. Using Etherscan APIs, both bytecode and relevant information have been downloaded, obtaining for each smart contract its name, address, hash, deployment date, bytecode length, compiler version and other information that are not relevant for us. We enforced that smart contract bytecode hashes are unique, so that in the dataset there are no duplicates. Indeed, it is common practice to reuse existing smart contracts, especially libraries and interfaces, and deploy them multiple times in the blockchain at different accounts.

The average length of these smart contracts is 7 351 bytes, the average transaction number is 337, the average balance is 9.6×10^{17} Wei, which corresponds to 1158$ (exchange rate on 2021-01-17). As shown in Fig. 4.a, the compiler versions varies on a wide range, focusing especially on the old versions. This datum is crucial in order to assess that EtherSolve does not assume a specific Solidity version (the compiler often underwent dramatic changes) but our patterns are general and work well across many different language versions.

B. Comparison Tools

Among the existing tools for smart contracts analysis, we select for the comparison with EtherSolve the ones that: (i) perform a static analysis at the bytecode level (no information from the source code or ABI file), (ii) emit the CFG as output, possibly using a (documented) specific running configuration (e.g., a special command line flag). These criteria led us to consider the following approaches:

- *Oyente-EthIR*: EthIR extends the Oyente framework and performs a high-level analysis of Ethereum bytecode. Oyente builds a CFG of Ethereum bytecode to detect different kinds of vulnerabilities [19], [20].
- *Octopus*: analysis framework for Ethereum bytecode. It starts from the bytecode and produces a CFG to support reverse engineering and understand the internal behaviour of smart contracts [21].
- *Mythril*: security analysis tool for Ethereum bytecode that detects security problems in smart contracts. It does not build a CFG, but a trace tree given by symbolic execution and SMT (Satisfiability Modulo Theories) solving [22], [23], [24].
- *Vandal*: static analysis framework for Ethereum smart contracts that decompiles bytecode to an intermediate representation that includes the control flow of the code [17], [25], [26].
- *Gigahorse*: a decompiler that transforms smart contracts from EVM bytecode into a highlevel 3-address code representation. The tool does not require the Solidity source code [27], [28], [29].

We discarded other tools such as *Securify2*, *Ethersplay*, *Manticore* and *Slither* because they analyse Solidity source code instead of Ethereum bytecode; *Evm_cfg_builder* because we did not find an easy way to make it emit the CFG; *Jeb* and *MythX* because they are paid tools and *Porosity* because it requires the ABI as input and also because the project seems discontinued. *Panoramix* [30] was discarded because it decompiles the code without building a CFG and it is a discontinued project.

C. RQ$_1$: Success Rate

EtherSolve manages to analyse all the smart contracts in the dataset except three, obtaining a success rate of 99.7%. The success rate is defined as the ratio between the smart contracts analysed without critical errors and the size of the dataset. The reason for these failures is that these three smart contracts do not match the most common patterns generated by the Solidity compiler. Indeed, one of them was written in Vyper rather than in Solidity, another one was empty with length of zero bytes and the last one has a unusual begin section that the tool failed to parse. The compiler version of Solidity is correctly identified by EtherSolve for all the solved contracts.

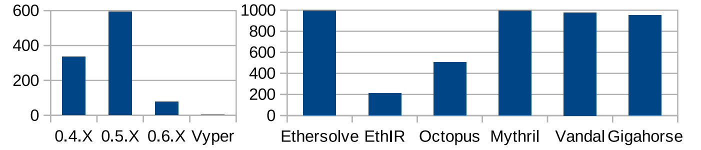

Fig. 4: Compiler versions of dataset contracts (a) and success rate for the different tools (b)

[2]Contracts have been download on 2020-06-10 [18]

[3]Using the standard random library in Python 3.7

We measured the time spent by EtherSolve to parse the bytecode, generate the basic blocks, solve *orphan jumps* and decorate the CFG. The average time is about 3 seconds per smart contract. For 930 smart contracts the analysis took less than a second each, while there are 7 smart contracts that needed more than a minute of computation, with a maximum of 10 minutes, due to their huge dimension and the large number of edges.

D. RQ_2: Comparison with Existing Approaches

First of all we compared the approaches in terms of success rate. We ran the tools on our dataset and we counted the successful executions (no crash) and the non-empty CFGs given in output. We also insert a reasonable timeout of 10 minutes per contract.

As shown in Fig. 4.b and in Table II, Mythril was able to analyse almost all the smart contracts (997 out of 1000), immediately followed by Vandal with 978 smart contracts and by Gigahorse with 951 smart contracts. Octopus and EthIR instead reached an error state on many smart contracts, computing a CFG only for 504 and 212 smart contracts respectively. Next we focused on the CFGs emitted by the tools, starting with an automatic numerical comparison of nodes and edges. Then, we proceeded with a manual inspection of the anomalous cases.

For each smart contract in the dataset we adopt a common representation for the outputs of the different tools. The chosen representation is a JSON file that contains a list of nodes (which represent the basic blocks), identified by their offset and a list of edges, identified by a pair of offsets. For each smart contract we count the number of nodes, the number of edges and the differences in the numbers of nodes and edges among the CFGs generated by different tools. To understand if there are portions of the bytecode which are actually data (static data, child contracts or compiler metadata) but that the candidate tool wrongly interprets as code, we calculate the number of nodes in the CFG generated by a candidate tool that have a higher offset than the highest offset node obtained by EtherSolve. The candidate tools have been executed in Docker [31] containers for compatibility reasons.

In the manual analysis we focused on the smart contracts for which the automatic analysis reported uncommon or anomalous results. These are relatively few contracts with sensitive differences in the number of nodes or edges between the CFG extracted by EtherSolve and by the compared tool. To support the manual analysis we implemented a script that generates a *diff graph*, where the two CFGs are combined and the nodes that are present only in the first or in the second graph are highlighted in a different colour.

The results of both automatic and manual analysis are discussed in the following. Table II contains a summary of the automatic analysis, with the smart contracts successfully analysed, the average number of both nodes and edges and the average number of basic blocks in the static data segment. Table III contains a subset of interesting/anomalous contracts that have been subject to manual analysis. Since contract names are not unique, contracts are identified by their addresses.

TABLE II: Comparison with state of the art.

	Success Rate (/1000)	**Average Nodes**	**Avg. Nodes Static data**	**Average Edges**
EthIR	212	139.4	51.6	150.3
Octopus	504	241.4	11.8	220.7
Mythril	997	4.0	8.5	3.1
Vandal	978	302.6	6.7	493.6
Gigahorse	951	245.4	1.2	288.7
EtherSolve	**997**	**301.6**	**0**	**361.8**

TABLE III: Subset of smart contracts used in manual analysis

Octopus	
a. SafeMath	0x7de33b2672efb11fde366dae96bd63b985bce186
b. ZipmexTokenP.	0xaa602de53347579f86b996d2add74bb6f79462b2
c. ltdFDTfactory	0xf155152d838b7a023317ad8c1e8c02aab7e8f2a2
d. Dividend	0xdb6f50cf0c521a98b6852839aa5cbea4e2430052
Mythril	
e. DharmaKeyR.	0x0000000000bda2152794ac8c76b2dc86cba57cad
f. DharmaT.R.	0x8b028e2fad2dc99999fb784ca9d7267981c90b4d
g. AltesF.G.	0x8313675d1405f3f7aee3da9d63e0bf5c30c75832
h. SMRT16Ext	0xdabb0c3f9a190b6fe4df6cb412ba66c3dd3e2ad1
Vandal	
i. FlashFloss	0xfd4085e56a96787fb7acd9b49510f874c3d4afcb
j. FoxInvSplit	0xd69015163e250a70ee4a607812afda5372132cc4
k. BCN20	0x1964f2f3ce45ac518b18ef4aa4265f8aadcef4ae
Gigahorse	
l. OneSplit	0x3a2d9db352580eb50018fc86eae32e19070a9982
m. EthexLoto	0x0e26b2dc8ef577baf50891eac94f0def59b5da16
n. ManagedAccount	0x0f4f45f2edba03d4590bd27cf4fd62e91a2a2d6a
o. ERC20Salary	0xcc2ba2eac448d60e0f943ebe378f409cb7d1b58a

1) Oyente-EthIR: In most cases, when EthIR finds more nodes than EtherSolve, they have very high offset (Cf. Table II), so they are static data or metadata interpreted as code. Instead, when nodes match, edges match too. Because of its very low success rate (21%), we deemed not so interesting to continue with a manual analysis of this tool results.

2) Octopus: Like EthIR, Octopus finds more blocks than EtherSolve with a higher offset, so they are probably data and not code. In the big majority of the cases, however, Octopus finds very few edges, sometimes even zero edges (Cf. Table II). During manual analysis, we discovered that Octopus misses some patterns for the metadata separation, so metadata are parsed as if it were code (Table III.a). Moreover, in many cases, Octopus does not detect edges that should be present according to source code (Table III.b). In some other contracts, Octopus evaluates the bytecode as creation code, thus it analyses only the second part starting with `60806040` (the most common begin sequence of a contract); however, this part of the code is a child contract and not the main one, so the computed graphs are completely different (Table III.c and Table III.d).

3) Mythril: This tool is a particular case, as it does not extract a CFG, but a trace tree from dynamic/symbolic analysis in order to detect vulnerabilities. The output of this execution

trace is not directly comparable with the CFG computed by EtherSolve. Nonetheless, indirect comparison can be performed by checking if the EtherSolve CFG misses nodes or edges that are found by the Mythril dynamic analysis. This case would correspond to incompletenesses in the CFG elaborated by EtherSolve.

In the 4% of the contracts, Mythril finds a bunch of edges which are not detected by EtherSolve. In some cases, Mythril adds some artificial basic blocks containing `"0: STOP"` which are placeholders that do not come from the analysed code, but they indicate the end of the execution trace (Table III.e). Sometimes, basic blocks are not split when there is a `JUMPDEST` in the middle, so there is a little discrepancy in the edges, but the CFG is definitely compatible (Table III.f and Table III.g). In some other cases, Mythril finds new basic blocks and edges that are not part of the main contract, but that are opcodes of a child contract created by the main one (Table III.h). The child contract code is computed at runtime, so the EtherSolve static analysis simply treats those bytes as part of the static data segment.

4) Vandal: Vandal CFG has always one node more than EtherSolve, which is the ending node with the `INVALID` opcode. In the 36% of the solved contracts the edges match, but in the remaining cases there are differences that we analysed manually. In some contracts the basic blocks do not match because Vandal does not support two opcodes that have been added in the most recent versions of the EVM [32]. In fact, the `SELFBALANCE` opcode is treated as invalid by Vandal, obtaining a basic block break with no outgoing edges (Table III.i and Table III.j). In many other cases, Vandal detects a huge amount of edges. Probably, when Vandal is not able to correctly compute the destination of a jump, it conservatively assumes all the basic blocks as possible successors (Table III.k). This hypothesis is supported by the diff graph, which shows too many outgoing edges for basic blocks that occur at the end of functions, probably because the return address could not be computed accurately. However, the number of call sites (that should correspond to the number of return edges) is much smaller according to the Solidity source code of these contracts. All in all, as shown in Table II, the average number of edges found by Vandal is significantly higher than EtherSolve, whereas the basic blocks are reconstructed in a similar way.

5) Gigahorse: Gigahorse records a good success rate, it is able to correctly analyse 95.1% of the samples. It tries to identify the private functions inside the code, and the computed CFG reflects this objective. Because of this strategy, the conversion into the intermediate representation is tricky; often its CFGs contain artificial blocks with the special `CALLPRIVATE` statement, introduced by Gigahorse to mark private function calls. Nevertheless, automatic comparison records high similarity with EtherSolve, so we proceeded with the deeper manual inspection.

For the contract shown in Table III.l, EtherSolve computes a set of basic blocks that are unreachable from the contract entry point, that might represent dead code. However, these unreachable blocks are not present in the Gigahorse CFG. Similar cases are Table III.m and Table III.n, where EtherSolve finds more nodes and edges than Gigahorse; these elements correspond to a portion in the middle of the bytecode which has not been reported in Gigahorse's CFGs. An opposite case happens on the contract in Table III.o, where EtherSolve identifies a set of blocks that are not reachable from the contract entry point whereas, according to the Gigahorse CFG, these blocks are reachable. Our speculation is that such basic blocks (which have a higher offset) belong to a child contract or to an internal library which is not a proper part of the main contract (e.g. called via `STATICCALL`), and thus they are skipped by EtherSolve (that only analyses intra-contract calls).

E. RQ_3: Re-entrancy Vulnerability Detection

In order to validate the effectiveness of the vulnerability detector built on top of EtherSolve, we compared it with the benchmark shared by Ghaleb and Pattabiraman [5]. The benchmark consists in 50 Ethereum smart contracts whose source code has been injected with re-entrancy vulnerabilities coming from 42 code snippets. This benchmark has been used by Ghaleb and Pattabiraman to compare the most prominent vulnerability detection tools. While their comparison was performed at source-code level, EtherSolve targets the bytecode level, so these injected contracts have been compiled before applying our analysis. While the original dataset consisted of 50 files, each source file could contain more than one contract and injected vulnerabilities could multiply in the compiled contracts, because of the use of inheritance that caused vulnerable code to be cloned from abstract contracts to concrete ones. Additionally, abstract contracts do not produce bytecode as they are not executable. Hence, to simplify the analysis, we opted to scan only one compiled contract per source file, i.e. the largest compiled contract, assuming that the remaining contracts were only supporting libraries or abstract contracts, whose CFGs were disconnected from the main one.

Another problematic aspect, acknowledged by Ghaleb and Pattabiraman, is that the dataset already contained vulnerabilities before SolidiFI injection, but they were not documented. To gather comparable results, we considered only vulnerabilities added by SolidiFI injection, by running EtherSolve before and after injection, and by keeping only those new vulnerabilities that are detected by the second scan and not by the first scan. Table IV shows the results of the comparison, measuring the average number of detected vulnerabilities for each sample by each tool and then computing the difference with the average number of bugs injected by SolidiFI (~26.86). The table highlights that EtherSolve is the second-best analysis tool, immediately after Slither. After a deeper comparison with the results given by these two top tools, we discovered that EtherSolve misses some cases (due to the choice of analysing only a single bytecode) whereas Slither often finds false positives. Indeed, the samples where EtherSolve found false negatives are smart contract obtained by multi-contract source code. Moreover, EtherSolve is the only tool that analyses

TABLE IV: Summary of re-entrancy analysis comparison for the different tools [5] and for EtherSolve

Tool	Avg. detection per sample	Diff. with SolidiFI injection
Manticore	2.14	-24.72
Oyente	8.58	-18.28
Mythril	15.12	-11.74
Slither	27.26	+0.40
Securify	28.84	+1.98
EtherSolve	**26.00**	**-0.86**

bytecode, thus having less information, so its results are even more valuable: EtherSolve can successfully analyse even closed-source smart contracts with high precision.

To confirm the precision of EtherSolve, we scanned all the 42 code snippets that SolidiFI uses for the injection. The EtherSolve analysis exactly detected all the 42 vulnerabilities, with a precision of 100%.

F. Discussion

The results obtained in this experimental validation suggest that EtherSolve is very effective in computing an accurate CFG: it is able to work on a wide range of Solidity versions and in almost all cases it computes an exhaustive graph.

The key point of our approach is the simplicity of the symbolic stack execution, which is limited to only a tiny set of opcodes, but which is capable of resolving the destinations of *orphan jumps*. However, there are particular cases of very complex or big smart contracts with peculiar structures for which EtherSolve is not able to identify certain edges.

Among the compared tools, only Gigahorse showed an accuracy similar to EtherSolve. However, they seems to be complementary, because each one could solve cases that the other one could not.

The results of the vulnerability detector suggest that EtherSolve is a powerful tool, and that can be easily extended to support accurate subsequent static analyses based on a precise CFG. Indeed, the performance of a simple reachability analysis compares to those of state-of-the-art security scanning tools.

V. Related Work

Over the last five years many tools have been developed to analyse Ethereum smart contracts, with different approaches and different objectives. A recent survey of them has been written by Praitheeshan et al. [33].

Some tools analyse directly the bytecode, often trying to build a CFG. Their approaches are very similar, as they try to execute symbolically the code to create logic predicates which, once resolved with a solver such as *Z3 theorem prover* [34], can determine the destinations of the *orphan jumps*. Among these tools there are *Oyente* [35], *EthIR* [19] and *Octopus* [21]. However, these tools aim at detecting vulnerabilities, and the extracted CFG is only an intermediate output.

A slightly different approach is the one used by *Vandal* [17], [25], which translates the bytecode into registry based operations, identifies the basic blocks and then tries to resolve the jump address through a fixed point analysis. Even in this case the CFG is only an intermediate output, as the target of Vandal is the vulnerability analysis with the *Souffle* suite [36]. Our tool instead focuses on the CFG building, keeping the symbolic stack execution as simple as possible, in order to resolve the highest number of *orphan jumps*.

A related tool which extract a CFG is *Jeb* [37], which is a professional decompiler with the ability to analyse Ethereum smart contracts. However, it is closed source with a subscription fee. Another decompiler is *Porosity*, one of the first tool to analyse Ethereum bytecode, but it needed the contract ABI too. Moreover it is discontinued since January 2018 [38]. A relevant decompiler is Gigahorse [27]–[29], a recent tool which builds a CFG and tries to find internal function with heuristics, obtaining an approximation of the original Solidity source code. Another decompiler is Panoramix [30] which, however, does not emit a CFG.

The wide majority of the Ethereum tools that performs vulnerability analysis do not expose a CFG, or even they do not extract it. Other tools instead do the analysis on the Solidity source code, or use the bytecode together with additional information that are not always available for closed source contracts. A completely different approach is the one implemented by *Mytril*, which uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities [23]. It does not build a CFG, but a trace tree, i.e. a representation of all the execution paths encountered during the analysis. Its objective is to detect as many vulnerabilities as possible. *Crytic* [39] is an application that collects many tools for smart contract analysis, such as *Manticore*, *Ethersplay*, *Echidna*, *Slither* and more, but they do not use a CFG or they do not analyse only the bytecode. In fact their objective is the vulnerabilities detection inside the Solidity source code.

Finally, there are other tools such as *Securify2* [40], which analyse Solidity source code, *Maian* [41], which performs dynamic analysis on a private blockchain, and *Gasper* [42], which analyse the gas cost of the contracts.

VI. Conclusion

Despite it would be very important to automatically analyse smart contracts and detect potential defects and vulnerabilities on their code, most of the existing analysis tools for Ethereum bytecode come with some shortcomings and limitations. For example, the accurate extraction of the CFG from the Ethereum bytecode is very challenging due to engineering decision on its infrastructure. We propose EtherSolve, a fast, reliable and precise approach to compute an accurate CFG from Ethereum bytecode. We believe that this CFG could be the starting point for new static analysis tools that aim at detecting defects and vulnerabilities in Ethereum smart contracts, built on top of an accurate CFG, such as the EtherSolve re-entrancy detector.

Acknowledgment

This paper has been partially supported by project MIUR 2018-2022 “Dipartimenti di Eccellenza”.

REFERENCES

[1] C. Dannen, "Introducing ethereum and solidity," 2017.

[2] A. R. Sai, C. Holmes, J. Buckley, and A. L. Gear, "Inheritance software metrics on smart contracts," 2020.

[3] G. Prisco. The dao raises more than $117 million in world's largest crowdfunding to dat. [Accessed: 2020-07-28]. [Online]. Available: https://bitcoinmagazine.com/articles/the-dao-raises-more-than-million-in-world-s-largest-crowdfunding-to-date-1463422191

[4] X. Li, T. Chen, X. Luo, T. Zhang, L. Yu, and Z. Xu, "Stan: Towards describing bytecodes of smart contract," 2020.

[5] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 415–427, 2020.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger."

[7] Ethereum. Solidity documentation - creating contracts. [accessed: 2020-12-29]. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/contracts.html#creating-contracts

[8] EtherSolve: a tool for CFG extraction from Etherum bytecode. DOI: 10.5281/zenodo.4607305. [Online]. Available: https://github.com/SeUniVr/EtherSolve/tree/ICPC-2021

[9] EtherSolve: ICPC-2021 replication package. DOI: 10.5281/zenodo.4607307. [Online]. Available: https://github.com/SeUniVr/EtherSolve_ICPC2021_ReplicationPackage

[10] Ethereum. Solidity documentation. [accessed: 2020-07-02]. [Online]. Available: https://solidity.readthedocs.io/

[11] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, p. 475–491, Apr 2020. [Online]. Available: http://dx.doi.org/10.1016/j.future.2019.12.019

[12] Ethereum.org. Ethereum official page. [accessed: 2020-07-02]. [Online]. Available: https://ethereum.org/en/

[13] A. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018.

[14] Etherscan.io. Etherscan. [accessed: 2020-07-02]. [Online]. Available: https://etherscan.io/

[15] Ethervm.io. Ethervm. [accessed: 2020-07-02]. [Online]. Available: https://ethervm.io/

[16] V. Tabora. (2019) The ethereum virtual machine (evm) runtime environment. [Accessed: 2020-07-15]. [Online]. Available: https://medium.com/0xcode/the-ethereum-virtual-machine-evm-runtime-environment-d7969544d3dd

[17] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *CoRR*, vol. abs/1809.03981, 2018. [Online]. Available: http://arxiv.org/abs/1809.03981

[18] Etherscan. List of verified contract addresses with an opensource license. [Accessed: 2020-07-27]. [Online]. Available: https://etherscan.io/exportData?type=open-source-contract-codes

[19] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," *CoRR*, vol. abs/1805.07208, 2018. [Online]. Available: http://arxiv.org/abs/1805.07208

[20] P. Gordillo. Github - ethir. [accessed: 2020-07-07]. [Online]. Available: https://github.com/costa-group/EthIR#ethir

[21] P. Ventuzelo. Github - octopus. [accessed: 2020-07-07]. [Online]. Available: https://github.com/pventuzelo/octopus

[22] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITBSecConf*, 2018.

[23] ConsenSys. Github - mythril. [accessed: 2020-07-07]. [Online]. Available: https://github.com/ConsenSys/mythril

[24] M. Balunovic, P. Bielik, and M. Vechev, "Learning to solve smt formulas," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 10 337–10 348.

[25] M. Kong, "A scalable method to analyze gas costs, loops and related security vulnerabilities on the ethereum virtual machine," https://github.com/usyd-blockchain/vandal/wiki/pubs/MKong17.pdf, The University of Sydney, NSW 2006 Australia, 11 2017.

[26] Usyd-blockchain. Github - vandal. [accessed: 2020-07-07]. [Online]. Available: https://github.com/usyd-blockchain/vandal

[27] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1176–1186. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00120

[28] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276486

[29] ——, "Madmax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, p. 87–95, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3416262

[30] eveem.org. panoramix. [accessed: 2020-12-08]. [Online]. Available: https://github.com/eveem-org/panoramix

[31] Docker. [Accessed: 2020-07-28]. [Online]. Available: https://www.docker.com/

[32] M. H. Swende. Eip 1884: Repricing for trie-size-dependent opcodes. [Accessed: 2020-07-28]. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1884

[33] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," 2019.

[34] Z3Prover. Github - z3. [Accessed: 2020-07-07]. [Online]. Available: https://github.com/Z3Prover/z3

[35] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[36] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 422–430.

[37] Jeb - ethereum contract decompiler. [Accessed: 2020-07-27]. [Online]. Available: https://www.pnfsoftware.com/jeb/evm

[38] Comaeio. Github - porosity. [Accessed: 2020-07-20]. [Online]. Available: https://github.com/comaeio/porosity

[39] Crytic: continuous assurance for smart contracts. [Accessed: 2020-07-20]. [Online]. Available: https://crytic.io/

[40] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," *CoRR*, vol. abs/1806.01143, 2018. [Online]. Available: http://arxiv.org/abs/1806.01143

[41] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," 2018.

[42] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 442–446.