The Journal of Systems & Software 200 (2023) 111653

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode[☆]

Michele Pasqua [*], Andrea Benini, Filippo Contro, Marco Crosara, Mila Dalla Preda, Mariano Ceccato [*]

University of Verona, Strada le Grazie 15, Verona 37134, Verona, Italy

ARTICLE INFO

Article history:
Received 17 January 2022
Received in revised form 15 December 2022
Accepted 20 February 2023
Available online 2 March 2023

Keywords:
Reverse engineering
Static analysis
Smart-contract
Ethereum

ABSTRACT

The immutable nature of Ethereum transactions, and consequently Ethereum smart-contracts, has stimulated the proliferation of many approaches aiming at detecting defects and security issues before the deployment of smart-contracts on the blockchain. Indeed, the actions performed by smart-contracts instantiated on the blockchain, possibly involving substantial financial value, cannot be undone.

Unfortunately, smart-contracts source code is not always available, hence approaches based on static analysis have very often to face the problem of inspecting the compiled Ethereum Virtual Machine (EVM) bytecode, retrieved directly from the blockchain. However, due to the intrinsic complexity of EVM bytecode (especially in jumps address resolution), the state-of-the-art static analysis-based solutions have poor accuracy in the automated detection of Ethereum smart-contracts programming defects and vulnerabilities.

This paper presents a novel approach based on *symbolic execution* of the EVM operands stack that allows to resolve jumps address in the EVM bytecode and to construct a *precise* Control-Flow Graph (CFG) of compiled smart-contracts. Many static analysis techniques are based on a CFG-based representation of the smart-contract to validate, and would therefore benefit from our approach.

We have implemented the CFG reconstruction algorithm in a tool called *EtherSolve*. Then, we have validated the tool on a large dataset of real-world Ethereum smart-contracts, showing that EtherSolve extracts more precise CFGs, w.r.t. state-of-the-art available approaches. Finally, we have extended EtherSolve with two detectors for two of the most prominent Ethereum smart-contracts vulnerabilities (Reentrancy and Tx.origin). Experimental results show that exploiting the proposed CFG reconstruction static analysis, leads to more accurate vulnerabilities detection, w.r.t. state-of-the-art security tools.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Smart-contracts extend blockchain-based cryptocurrencies (e.g., Ethereum), allowing to store *programs* transparently on the blockchain, that are eventually executed by the distributed network of miners. In other words, a smart-contract is a self-executing program that runs on a blockchain.

The peculiarity of a smart-contract is that the program and all the actions it performs (e.g., money transactions) are immutable: once written on the blockchain, they cannot be modified nor deleted, even in case of programming defects identified after deployment. Programming defects and vulnerabilities might result in frauds or in financial values that are frozen indefinitely (Prisco, 2020). Moreover, smart-contracts security is becoming a crucial point due to the advent of the so called *Decentralized Financial* (De-Fi): a range of semi-familiar financial products re-skinned for the cryptocurrency age (Chambers, 2020). DeFi sees a great involvement of new smart-contracts, reaching 3.1 million contract calls in a single day (Coindesk, 2020) with $88.5 billion total value locked (Pulse, 2021), and these numbers are still growing.

For these reasons, code review, possibly supported by automated analysis tools, is crucial, in order to detect programming defects and vulnerabilities before smart-contracts deployment or before erroneous transactions committed (and permanently stored) in the blockchain. This is especially critical for closed-source smart-contracts, whose source code cannot be inspected

[☆] Editor: Christoph Treude.

[*] Corresponding authors.

E-mail addresses: michele.pasqua@univr.it (M. Pasqua), andrea.benini@studenti.univr.it (A. Benini), filippo.contro_01@studenti.univr.it (F. Contro), marco.crosara@studenti.univr.it (M. Crosara), mila.dallapreda@univr.it (M. Dalla Preda), mariano.ceccato@univr.it (M. Ceccato).

https://doi.org/10.1016/j.jss.2023.111653

by end-users, and the only available representation is the compiled EVM bytecode on the blockchain (Li et al., 2020).

The precision of the static analysis is one of the key points to promptly deploy and run correct smart-contracts. However, state-of-the-art tools for detecting programming defects and vulnerabilities in Ethereum smart-contracts experience substantial limitations: analysis results contain lots of *false positives* (false alarms) and *false negatives* (overlooked issues) (Ghaleb and Pattabiraman, 2020). A possible explanation for such poor performance could be the intrinsic difficulty of analyzing the EVM bytecode. In fact, despite the bytecode is easy to parse (fixed length opcodes) its semantics and control-flow graph (CFG) are difficult to reconstruct due to the following EVM design choices.

- Jumps destination is not an opcode parameter. 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: the "return" is implemented by pushing the return address on 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" placed at the smart-contract entry-point, that decides what address to jump to, depending on the call actual parameters.
- The smart-contract constructor is executed only when the contract is deployed on the blockchain, and then discarded. Thus, its bytecode is not available in the blockchain.

The most effective static analysis algorithms rely heavily on the particular representation of the code, that is usually based on the control-flow graph. Indeed, CFG precision is a key point of every analysis, and when research tools base their analysis on a partial or imprecise CFG, the final tool results are also imprecise. Since it is still quite difficult to precisely compute a precise CFG, often, tools use alternative program representations (e.g., trace tree Mueller, 2020 or three address code Brent et al., 2018), sacrificing analysis accuracy.

In this paper we go in the opposite direction: we aim at extracting a *precise CFG* from the EVM bytecode. To this end, we propose a static analysis approach, called *symbolic stack execution*, that resolves jumps destination based on the symbolic execution of the operands stack. After the jumps destination is resolved, an accurate CFG can be built. The approach has been implemented and open sourced as EtherSolve,[1] a fully automated tool to compute a precise CFG starting from EVM bytecode. This result represents a beneficial starting point for any subsequent sophisticated static analysis meant to identify programming defects or vulnerabilities in Ethereum smart-contracts. To quantify the improved precision of this CFG, we have extended EtherSolve adding analyses aiming at detecting two of the most prominent Ethereum smart-contracts vulnerabilities (Dika and Nowostawski, 2018), namely *Reentrancy* and *Tx.origin*. These detectors turn out to perform better than state-of-the-art security EVM bytecode scanning tools, with results comparable to analysis tools that can also inspect smart-contracts source code.

The present paper is a deeply revised and extended version of the companion conference paper (Contro et al., 2021). In particular, we better explained the problem and improved the overall presentation of the proposed methodology. Furthermore, we extended EtherSolve with a mechanism to identify functions entry-point and a specific analysis detecting Tx.origin vulnerabilities. Finally, we extended the experimental validation, adding an assessment of the accuracy of EtherSolve in identifying functions entry-point and a comparison of vulnerabilities detection efficacy (Reentrancy and Tx.origin) with the state-of-the-art source code level analysis tools.

The paper is structured as follows. After covering the background of smart-contracts in Section 2, the static analysis reconstructing the CFG of Ethereum smart-contracts is described in Section 3. In Section 4, the static analysis is extended in order to detect Reentrancy and Tx.origin vulnerabilities. Section 5 presents our empirical validation and comparison with state-of-the-art tools. Then, Section 6 discusses the related work and Section 7 closes the paper.

2. Background

2.1. Ethereum

Ethereum is an open-source platform for decentralized applications, based on the *blockchain* technology. On the Ethereum network, it is possible to write simple programs, called *smart-contracts* (Dannen, 2017; Antonopoulos et al., 2018), that (semi-)automatically manage the underlying network cryptocurrency, called *Ether* (ETH). The actions that can be performed in Ethereum are transactions, i.e., transfer of funds or data between different ETH accounts. Every new transaction is irreversible and it is permanently added in a new *block* that updates the blockchain (Dannen, 2017; Antonopoulos et al., 2018).

In the Ethereum network each principle has an account identified by an address (a sequence of 20 bytes). There are two types of accounts: *Externally Owned Accounts* (EOA) and *contract accounts* (Buterin, 2015). The former is a simple address that does not point to any code: it can only emit and receive transactions (similarly to Bitcoin wallets Antonopoulos, 2017). The latter is the identifier of a *smart-contract* deployed in the network, which is run whenever a transaction is sent to its address (Antonopoulos et al., 2018).

Ethereum uses a *proof-of-work* (POW) system as a consensus mechanism (Zheng et al., 2020; Baliga, 2017). Participants of the network, called *miners*, use their time, computational power and crypto currency assets to compete. The miner that succeeds is allowed to add blocks to the Ethereum blockchain and gets a reward (Buterin, 2015). Rewards are paid by the users who invoke the execution of a smart-contract or simply want to transfer funds to other accounts. In fact, every operation in the network has a cost expressed in the unit of *Gas* and the price per unit is expressed in Wei, a fraction of an Ether. In the near future, a switch to a *proof-of-stake* (POS) consensus mechanism will be performed, yielding to Ethereum 2.0. In a POS setting, miners are replaced by *stackers*, that validate transactions based on the amount of cryptocurrency they have staked, instead of the computational power they are able to employ.

2.2. The solidity language

Smart-contracts for Ethereum can be written with different high-level programming languages, but *Solidity* (Ethereum, 2020a) is indubitably the most wide-spread (Etherscan.io, 2020). Solidity is a Turing-complete object-oriented language and smart-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 representing the funds owned by the address account. The functions `deposit` and `deposit100` allow the user to deposit currency into its virtual

[1] The open source tool is available at https://github.com/SeUniVr/EtherSolve.

```
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

account. The former allows to deposit an arbitrary amount, the latter is a special case which allows to transfer exactly 100 Wei. 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, for instance: transfer, that sends Ether to a certain address; revert, that makes the transaction fail and rolls-back to the state preceding the transaction; require, that enforces a certain boolean condition and in case the condition is not met a revert is performed.

2.3. Compiling solidity into EVM bytecode

In order to actually run a smart-contract on the Ethereum blockchain, the Solidity source code needs to be compiled into *EVM bytecode*, in order to be executed by the Ethereum Virtual Machine (Ethervm.io, 2020).

Given a smart-contract, the Solidity official compiler `solc` generates the *creation code* and the *runtime code*. The former 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 Ethereum, 2020b). The latter is the actual bytecode deployed on the blockchain and it is divided in three main segments. The first segment contains the opcodes that the EVM executes; the second segment is optional and contains static data (e.g., strings or constant arrays); the last segment contains the metadata. In particular, the metadata segment contains compilation information, such as the compiler version and the (hashed) sources used, in order to verify its source code (Ethereum, 2020a). The metadata segment is hashed and appended to the contract bytecode.

In addition, `solc` also produces the *Application Binary Interface* (ABI), a file containing the list of the functions in the smart-contracts that can be called by a user, together with the type and number of parameters. Functions are not identified by their name but by the hash of their signature. The ABI file is not deployed on the blockchain, it is distributed separately to all parties that aim to interact with the smart-contract.

Concerning the EVM execution, the main memory of a smart-contract consists in a *stack*, namely a volatile LIFO queue with 1024 blocks of 32 bytes (Ethereum, 2020a; Ethervm.io, 2020). 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 (Tabora, 2020). For instance, the (hexadecimal) bytecode string 6005600301 translates to opcodes list 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 (iii) execute the addition operation, which consumes two elements from the stack and produces their sum as result, leaving the final stack with the value 0x08 only.

```
Runtime Code:
6080604052600436106100345760003560e01c8063140
e9ac714610039 ... 600020600082825401925050819
055505056fe
Metadata:
a2646970667358221220e62b6e0d256ecbc0a1b39b99b
f0a2b509ed60dd83c71541b2d00fed1bde5a9e464736f
6c634300060b0033
```

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.

The EVM bytecode is composed by *EVM opcodes* that can be grouped in categories, including arithmetic and logic operations, control flow operations, stack operations, environmental and block information, memory and storage operations and system operations. The complete list of opcodes with their semantics is defined in the Ethereum yellow paper (Wood, 2014), and there can be little variations among different EVM versions. Listing 2 shows a portion of a Solidity smart-contract compiled into EVM bytecode, while Listing 3 shows the translation of the bytes into EVM opcodes. Bytecode can be easily parsed into opcodes, which are the minimum instructions that the EVM can execute and are identified with bytes.

Every opcode pushes or pops a certain number of elements from/to the stack, and it can 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) (Wood, 2014; Ethervm.io, 2020).

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 (Brent et al., 2018). Thus, parsing this segment of memory as code might generate spurious results, including invalid opcodes and wrong jumps destination.

The control-flow of the smart-contract is also managed by means of 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, respectively) have to read the jump destination from the stack. Jumps destination is not indicated with a label, but with the offset w.r.t. the next instruction in the code (Brent et al., 2018). Unlike x86 Assembly, in the EVM there is not the concept of *function*: everything is managed through jumps (there are no opcodes for function calls nor for call returns). The only return available is for function calls coming from external smart-contracts.

These design choices make the EVM bytecode quite difficult to analyze statically. In particular, since jumps destination is computed at run-time, the CFG cannot be reconstructed without a sort of stack simulation, whose accuracy directly affects the precision of extracted CFG.

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 move 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 moves the execution to the beginning of the *fallback function*. This function has not parameters nor return values (Ethereum, 2020a), and it is called automatically when a money transfer is performed (on the destination contract).

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, moves the execution to JUMPDEST, resulting in an actual return statement.

3. Control-flow graph reconstruction

As already introduced, the main goal of our approach is to reconstruct a precise Control-Flow Graph (CFG) of Solidity smart-contracts, starting from the EVM bytecode only (no ABI nor source code is needed). The CFG is a directed graph representing the smart-contract flow of execution: nodes are the contract basic blocks (sequence of opcodes with no jumps), while edges connect potential successive basic blocks. In this section we will describe in details how we retrieve a CFG from EVM bytecode.

3.1. Approach overview

The CFG reconstruction algorithm is composed of the following incremental steps.

Bytecode Parsing The binary representation of the bytecode is split in the actual smart-contract code and in the metadata. Then, the code part is further parsed to identify opcodes.

Basic Blocks Identification Opcodes are grouped in basic blocks and the explicit jump destinations between basic blocks are computed.

Symbolic Stack Execution Symbolic execution is applied to the execution stack, in order to resolve non-trivial jump destinations.

Static Data Separation The static data segment is separated from the actual executable code.

CFG Decoration The obtained CFG is decorated to highlight the dispatcher and to identify the entry-point of the fallback function.

Entry-points Detection Functions entry-point is detected inspecting the dispatcher blocks.

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

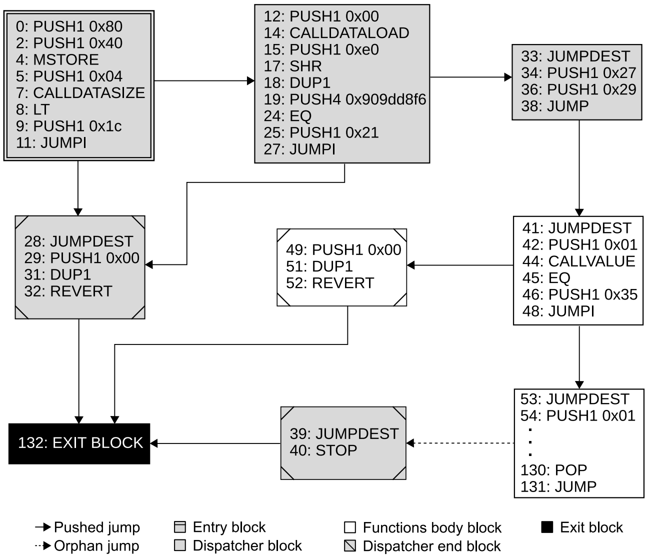


Fig. 1. Example of control-flow graph.

3.2. Bytecode parsing

The analysis starts with the binary representation of the EVM bytecode. The metadata section is identified by finding the corresponding header reported in the official Solidity documentation (Ethereum, 2020a). In case of metadata with *experimental features*, the header is different and not documented. Indeed, as stated in the Solidity documentation (Ethereum, 2020a), some features (such as the pragmas ABIEncoderV2 and SMTChecker) are not enabled by default. Still, they can be enabled by using the experimental pragma modifier. 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 generate the bytecode is extracted from the metadata.

The metadata are then dropped and the remaining bytes are considered as the actual smart-contract code to be further parsed. An example of how the bytecode is parsed into opcodes is shown in Listing 2 and Listing 3, where every two characters of the bytecode are translated into the corresponding opcodes (e.g., 0x6080 becomes PUSH1 0x80). Each opcode is unequivocally identified by its offset address, i.e., the position of the opcode in the bytecode.

3.3. Basic blocks 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 CFG edges. This operation is not always trivial as the jump destination is not an opcode parameter, but rather it is available on top of the stack at execution time. We identified two types of jumps: *pushed jumps* and *orphan jumps*. A

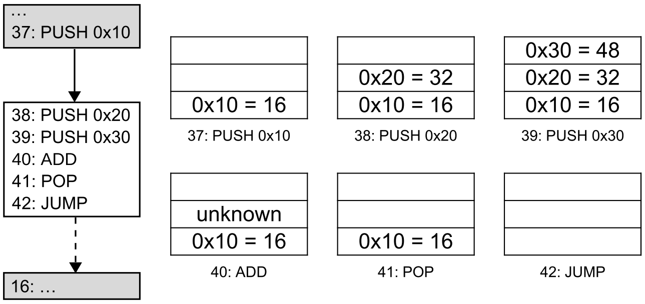


Fig. 2. Symbolic stack execution.

pushed jump is 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. Instead, orphan jumps are not preceded by a PUSH and their target is not immediate to compute. In the example CFG of Fig. 1 the block (starting at offset) 53 ends with an orphan jump, whereas the remaining jumps are pushed jumps.

We start by computing the edges resulting from pushed jumps, then the edges resulting from orphan jumps will be computed in a subsequent phase. To this end, each basic block is analyzed according to its last opcode as follows.

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 next block (in offset order), while the true branch is the argument of the push interpreted as destination offset for the JUMPI. The two corresponding edges are then 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 Section 3.4.

Others Opcodes like STOP, REVERT, RETURN, INVALID and SELFDESTRUCT have no successors, as the control flow is interrupted.

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.

3.4. Symbolic stack execution and orphan jumps

The most challenging step in the CFG reconstruction is the resolution of the destination addresses for *orphan jumps*. These jumps are very common: for instance, the Solidity compiler uses them to return from function calls. 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 or 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 jump addresses, updating the state of the stack accordingly, in such a way that 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 opcode the symbolic stack pops and pushes unknown elements, as they do not deal with the jump addresses.

Table 1
Look-up table for the *executeOpcode* function of Algorithm 1.

Hex	Opcode	Popped	Pushed	Effect on the symbolic stack
0 × 16	AND	2	1	$S.pop()$ 2 times, $S.push($unknown$)$ 1 time
0 × 50	POP	1	0	$S' = S.pop()$
0 × 60	PUSH1			
⋮	PUSHn	0	1	$S' = S.push(opcode.\text{argument})$, $\lvert opcode.\text{argument}\rvert = n$ bytes
0 × 7f	PUSH32			
0 × 80	DUP1			
⋮	DUPn	n	$n+1$	$S' = S.push(S[n-1])$
0 × 0f	DUP16			
0 × 90	SWAP1			
⋮	SWAPn	$n+1$	$n+1$	$S'[0] = S[n], \ \ S'[n] = S[0]$
0 × 9f	SWAP16			
0x*	Other	k	l	$S.pop()$ k times, $S.push($unknown$)$ l times

In the example depicted 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 modeled in full details (we do not need to): it simply consumes two elements of the stack and then generates a single unknown value to be pushed. The jump address is loaded before arithmetic operations, but it persists until the actual JUMP, so it can be resolved.

The symbolic stack execution handles the opcodes according to the rules represented in the look-up Table 1, where S denotes a stack that can contain numeric or the unknown values. We indicate the top of the stack with the position 0. The third and fourth columns of the table represent, respectively, the number of pop and push operations executed on the stack, according to the language documentation (Wood, 2014). The last column describes the effect of an opcode on the symbolic stack, where S' is the symbolic stack after the modification. The last row of the table considers a generic opcode not involving jump addresses, that performs an arbitrary number of push and pop operations on the stack. Note that, l can be only 0 or 1 since, at the moment, there are no operations that push more than one element in the stack.

The algorithm walks through the CFG using a Depth-First Search (DFS) keeping a snapshot of the stack state for each basic block. The following constraints have been introduced in order to avoid infinite loops: an edge cannot be analyzed 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 to note is the fact that a function can be called from different points of the code, resulting in different symbolic stacks and different paths of the CFG. Indeed, blocks can be traversed during the symbolic execution by multiple paths. A path is said to be *infeasible* when a real execution, given a particular state, would never take it. An example is a path that contains branches that are guarded by contrasting conditions, e.g., $x > 0$ in the first branch and $x \leq 0$ in the second branch. Even if the two branches are not dead code, they cannot be taken by the same execution when x is an input. In order to avoid infeasible paths, when the DFS visit encounters a block ending with JUMP, only its destination block (obtained from the symbolic stack) is added to the DFS queue.

The detailed algorithm for resolving orphan jumps is shown in Algorithm 1. It starts at Line 2 by initializing the variable V,

Algorithm 1: Resolve Orphan Jumps.

```
 1: function RESOLVEORPHANJUMPS($basicBlocks$)
 2:     $V \leftarrow set()$                                  ▷ visited
 3:     $CB \leftarrow basicBlocks.first$                     ▷ current block
 4:     $S \leftarrow symbolicExecutionStack()$               ▷ stack
 5:     $Q \leftarrow stack()$                                ▷ DFS queue
 6:     $Q.push(\langle CB, S \rangle)$                       ▷ DFS first element
 7:     while $Q \neq \emptyset$ do
 8:         $\langle CB, S \rangle \leftarrow Q.pop()$
 9:         for $op \in CB.opcodes$ do
10:             $S.executeOpcode(op)$                         ▷ look-up Table 1
11:         end for
12:         if $CB.opcodes.last =$ JUMP then
13:             $NO \leftarrow S.peek$                        ▷ next offset from stack
14:             $NB \leftarrow basicBlocks[NO]$               ▷ next block
15:             $CB.addSuccessor(NB)$
16:         end if
17:         if $CB.opcodes.last \neq$ JUMP then
18:             for $suc \in CB.successors$ do
19:                 $edge \leftarrow \langle CB.offset, suc.offset, S \rangle$
20:                 if $edge \notin V$ then
21:                     $V.add(edge)$
22:                     $Q.push(\langle suc, S \rangle)$
23:                 end if
24:             end for
25:         else if $CB.opcodes.last =$ JUMP then
26:             $edge \leftarrow \langle CB.offset, NO, S \rangle$
27:             if $edge \notin V$ then
28:                 $V.add(edge)$
29:                 $Q.push(\langle NB, S \rangle)$
30:             end if
31:         end if
32:     end while
33: end function
```

that stores the edges that have already been analyzed using stack equivalence as described before (lines 20 and 27). An edge is also labeled with the symbolic stack that has been used for its symbolic execution (lines 19 and 26). Then, the queue Q used for the DFS is initialized 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 1.

Lines 12-16 resolve orphan jumps 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 analyzed basic block to the target one has not been already analyzed using 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 contract shown in Listing 3 (the corresponding 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 1, consumes a value. Next, the symbolic execution of JUMPDEST leaves the stack unchanged and then the value 0x01 is loaded. The execution proceeds until the opcode at offset 129 is reached, which leaves an unknown value on the stack, that is then removed by the POP. Finally, the opcode at offset 131 contains the orphan jump, which can be resolved with the value pushed into the stack back at offset 34. At this point, the symbolic stack execution can detect that the successor basic block is the one at offset 39.

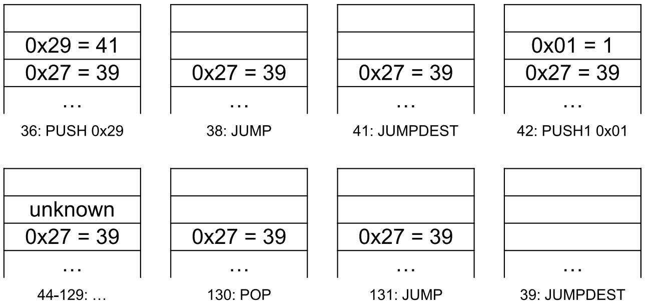


Fig. 3. Orphan jump resolution.

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.

3.5. Static Data Separation

The proposed approach proceeds with the removal of static data (if present). The first basic block containing the instruction 0xFE, which is the designated opcode for an invalid instruction, is identified. 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. All the subsequent opcodes 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 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 indicates that this assumption is reasonable.

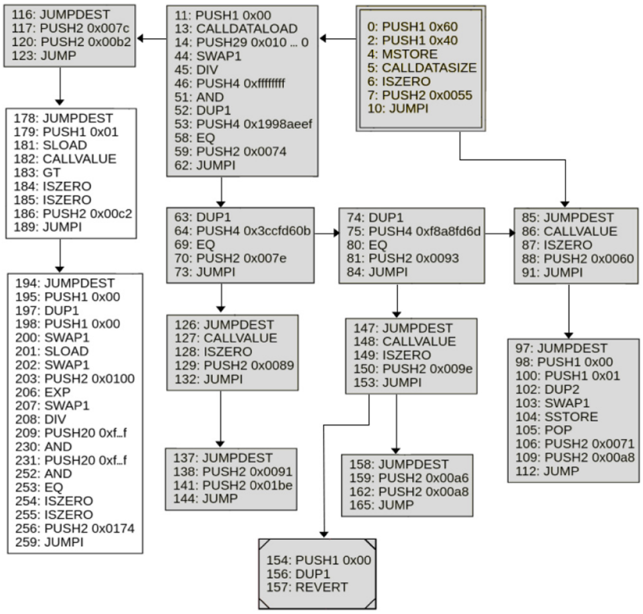
3.6. Control-flow graph decoration

In order to provide to the CFG additional information potentially useful to an analyst or for a static analysis (e.g., for vulnerabilities detection), our approach tries to highlight some relevant code portions, such as the dispatcher, the fallback function and the last basic block of the contract.

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 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. These opcodes cannot be present in other locations as they would manage return values outside the dispatcher. So, the algorithm considers as dispatcher every block with an address lower than the address of these opcodes. In the example of Fig. 1, the dispatcher blocks are highlighted in gray.

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

The detection of the fallback function entry-point is more difficult, because the dispatcher structure has been changing continuously across different versions of the Solidity compiler. In Fig. 4 we report a portion of the CFG retrieved from a simple contract with declared fallback function. In the picture, white



```
0: PUSH1 0x60
2: PUSH1 0x40
4: MSTORE
5: CALLDATASIZE
6: ISZERO
7: PUSH2 0x0055
10: JUMPI
```

```
11: PUSH1 0x00
13: CALLDATALOAD
14: PUSH29 0x010 ... 0
44: SWAP1
45: DIV
46: PUSH4 0xffffffff
51: AND
52: DUP1
53: PUSH4 0x1998aeef
58: EQ
59: PUSH2 0x0074
62: JUMPI
```

```
116: JUMPDEST
117: PUSH2 0x007c
120: PUSH2 0x00b2
123: JUMP
```

```
178: JUMPDEST
179: PUSH1 0x01
181: SLOAD
182: CALLVALUE
183: GT
184: ISZERO
185: ISZERO
186: PUSH2 0x00c2
189: JUMPI
```

```
194: JUMPDEST
195: PUSH1 0x00
197: DUP1
198: PUSH1 0x00
200: SWAP1
201: SLOAD
202: SWAP1
203: PUSH2 0x0100
206: EXP
207: SWAP1
208: DIV
209: PUSH20 0xf...f
230: AND
231: PUSH20 0xf...f
252: AND
253: EQ
254: ISZERO
255: ISZERO
256: PUSH2 0x0174
259: JUMPI
```

```
63: DUP1
64: PUSH4 0x3ccfd60b
69: EQ
70: PUSH2 0x007e
73: JUMPI
```

```
74: DUP1
75: PUSH4 0xf8a8fd6d
80: EQ
81: PUSH2 0x0093
84: JUMPI
```

```
85: JUMPDEST
86: CALLVALUE
87: ISZERO
88: PUSH2 0x0060
91: JUMPI
```

```
126: JUMPDEST
127: CALLVALUE
128: ISZERO
129: PUSH2 0x0089
132: JUMPI
```

```
147: JUMPDEST
148: CALLVALUE
149: ISZERO
150: PUSH2 0x009e
153: JUMPI
```

```
97: JUMPDEST
98: PUSH1 0x00
100: PUSH1 0x01
102: DUP2
103: SWAP1
104: SSTORE
105: POP
106: PUSH2 0x0071
109: PUSH2 0x00a8
112: JUMP
```

```
137: JUMPDEST
138: PUSH2 0x0091
141: PUSH2 0x01be
144: JUMP
```

```
158: JUMPDEST
159: PUSH2 0x00a6
162: PUSH2 0x00a8
165: JUMP
```

```
154: PUSH1 0x00
156: DUP1
157: REVERT
```

Fig. 4. Example of CFG with complex dispatcher (outgoing edges to not reported nodes omitted).

```
1  pragma solidity ^0.5.0;
2  contract ReentrantContract {
3    mapping (address => uint) private balances;
4    ...
5    function withdraw (uint amount) public {
6      require(amount <= balances[msg.sender]);
7      if (msg.sender.call.value(amount)())
8        balances[msg.sender] -= amount;
9    }
10 }
```

Listing 4. A Solidity smart-contract vulnerable to Reentrancy.

```
pragma solidity ^0.5.0;
contract MaliciousContract {
  ReentrantContract reentrantContract;
  ...
  function attack () public {
    reentrantContract.withdraw(100);
  }
  function () payable {
    reentrantContract.withdraw(100);
  }
}
```

Listing 5. A Solidity smart-contract exploiting the Reentrancy vulnerability of Listing 4.

nodes are code nodes, while gray nodes are dispatcher nodes (we reported only some code and dispatcher nodes for simplicity, outgoing edges to not reported nodes have been omitted). As you can see from the picture, the dispatcher is quite complex and it is not trivial to identify automatically that the node at offset 0x97 is the first block of the fallback function.

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 successor offset. The successor with the highest offset is considered as fallback only if it does not end with a REVERT. Indeed, that would mean that the fallback function has not been declared or has been declared with only the REVERT statement. However, this approach 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 many static analyses techniques. This particular basic block in the example in Fig. 1 is the number 132.

3.7. Functions entry-point identification

In this final step we try to extract the hashes of the functions in the dispatcher blocks. As mentioned before, in order to pass the control to a desired function, the dispatcher matches the call data with the hash of the functions signatures. Hence, the algorithm navigates the dispatcher blocks looking for the opcodes dealing with such hash comparisons, obtaining a set of likely function hashes. The fallback function has no associated hash, and it is called in case no hash matches the call data.

In the example in Fig. 1, the function with hash 909dd8f6 is identified by the block starting at offset 12 (the fallback function is not present).

4. Smart-contract vulnerabilities

To demonstrate the usefulness of a precise CFG representation of Ethereum smart-contracts, we defined two static analyses for detecting smart-contracts vulnerabilities on top of EtherSolve, and compared their efficacy w.r.t. state-of-the-art detection tools. In particular, EtherSolve has been pipelined with two subsequent static analyses meant to detect cases of Reentrancy and Tx.origin vulnerabilities. Note that, we selected two security-related vulnerabilities, but any generic CFG-based analysis would benefit from our approach (e.g., data-flow analyses or under/over-flow detection mechanisms).

In this section, we first describe Reentrancy and Tx.origin vulnerabilities and then we explain the approach used to check such bugs in the EVM bytecode, based on the CFG extracted by EtherSolve. In the next section we will validate the efficacy of the proposed detectors on a dataset of real-world smart-contracts.

4.1. Reentrancy

One of the most prominent and dangerous vulnerabilities in Solidity (and, hence, in Ethereum smart-contracts) consists in the mishandling of possibly reentrant code. It has been made famous due to the catastrophic DAO incident (Prisco, 2020), that caused the loss of a large amount of money and serious consequences to the whole Ethereum network. This vulnerability consists in reentering a paying function multiple times while the contract is in an inconsistent state, thus causing possible leak of funds (Ghaleb and Pattabiraman, 2020). In the general case, the "reentrance" exploits the fact that the vulnerable contract calls primitives, such as money transfer, that the malicious contract can redefine, in such a way to reenter the vulnerable contract. If a money transfer occurs at each iteration of this loop, the process can be repeated and used to drain all resources from the attacked contract. Indeed, Reentrancy is consequence of an abuse of dynamicity in Solidity: the semantics of money transfer is dynamic and can be redefined.

A simpler, yet quite common, programming error that may lead to Reentrancy attacks consists in updating the contract state after (instead of before) executing a fund send primitive (i.e., a call). An example of a Reentrancy vulnerable contract following this pattern is shown in Listing 4, where the call statement at line 7 precedes the update of the variable balances at line 8. In Listing 5 we have a (malicious) contract that exploits the reentrancy vulnerability of the ReentrantContract. In particular,

```
1 function sendTo (address payable dest, uint amount)
      public {
2   require(tx.origin == owner);
3   dest.transfer(amount);
4 }
```

Listing 6. A Solidity smart-contract method vulnerable to Tx.origin.

```
1 pragma solidity ^0.5.0;
2 contract MaliciousContract {
3  VulnerableContract vContract;
4  address attackerAddr;
5  ...
6  function () payable {
7   vContract.sendTo(attackerAddr, msg.sender.balance);
8  }
9 }
```

Listing 7. A Solidity smart-contract exploiting the Tx.origin vulnerability of Listing 6.

the attack is launched calling the withdraw method on the vulnerable contract (line 6). When the call primitive is executed in the withdraw method, performing the money transfer, the fallback function of the contract MaliciousContract is fired. The latter contains a recursive call to the vulnerable contract, again on the withdraw method (this is the reentrant code). Since the balances variable in the vulnerable contract is updated after the money transfer, the second withdraw can be legitimately called, and a second money transfer is performed. The process is repeated until all money is drained from the vulnerable contract.

Reentrancy detector. Reentrancy vulnerabilities be exploited when a contract state update is performed after calling a, possibly unsafe, paying function. The proposed approach consists in traversing 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 can mount an attack to exploit a Reentrancy vulnerability (e.g., with an particularly crafted fallback function). Note that, Solidity statements to send funds are translated with the CALL opcode, that has the callee address hard-coded into the bytecode.

In particular, the detector traverses the contract CFG in order to detect blocks that refer to a CALL opcode, namely blocks that may call a paying function. For each such block, the detector checks if the block is *unsafe*. To do so, a symbolic stack execution on the block is fired, retrieving the pushed element on the stack when the CALL is executed. If the element is an unknown address the block is unsafe. Finally, for each unsafe blocks the detector checks if a SSTORE opcode can be reached. In case a SSTORE is reachable the detector signals a reentrancy vulnerability.

4.2. Tx.origin

Another quite dangerous vulnerability is related to the misuse of the tx.origin command, that returns the address of the (first) EOA in the callers path ended up in the current contract instance. Indeed, during a contract execution, a chain of calls may happen: an EOA calls a contract method that, in turn calls another contract, and so on. In this case, tx.origin returns the address of the EOA performing the first transaction. Instead, the command msg.sender returns the address of the closest parent caller, that could be either a EOA or a contract.

Among the first instructions of a contract method, very often we can find an authorization check: the contract verifies that the caller is authorized to perform the subsequent operations. Indeed, a contract can execute operations on behalf of the caller, comprising funds transfer. The latter check should be performed using the msg.sender instead of the tx.origin, since the actual contract caller may be different from the initial EOA starting the first transaction.

We have a Tx.origin vulnerability in a smart-contract when into a method of the contract the command tx.origin is used incorrectly (i.e., in place of the command msg.sender), as we can see in the example described in Listing 6. In the example, we can see that tx.origin is used in the method sendTo to perform the caller authentication (line 2). To exploit the vulnerability of the method sendTo, another contract may use the code described in Listing 7. In this case, the fallback function (line 6) of the (malicious) contract MaliciousContract is able to call the vulnerable method sendTo (line 7) and to perform operations on behalf of the user that have called the contract. In particular, when the victim calls the malicious contract, the latter transfers all money of the victim to a specific address (of the attacker that have crafted the malicious contract). The execution of this contract calls chain is performed without errors on the blockchain.

The vulnerability arises because when an EOA calls a contract, all the recursive calls performed by that contract have the address of the EOA as tx.origin, potentially transferring the EOA credentials, as we have seen the example.

Tx.origin detector. Concerning Tx.origin vulnerabilities, they can be exploited when we perform an authorization check on the contract caller using tx.origin instead of msg.sender, namely when we have a line like require(tx.origin==msg.sender) in the vulnerable contract. The proposed approach consists in traversing the CFG in order to detect the blocks resulting from the translation of that kind of line of code in the corresponding EVM bytecode. In particular, the block is identified by the opcode ORIGN, namely the translation of tx.origin, and it ends with a JUMPI, namely a conditional jump based on the equality concluding the block. Considering that the authorization check is usually performed at the beginning of the function, we can assume that the block is completely demanded to the translation of the require line. To identify this kind of blocks, EtherSolve scans the contract CFG in order to find the following common patterns:

```
ORIGIN PUSH20 AND EQ PUSH2 JUMPI
ORIGIN PUSH20 AND EQ ISZERO ISZERO PUSH2 JUMPI
```

Unfortunately, patterns may miss potential vulnerability instances. Indeed, some particular compiler versions may translate the require line to slightly different bytecode. To mitigate the problem, EtherSolve performs, in addition to the pattern matching check, a simple *taint analysis* on the block containing the opcode ORIGIN. In particular, EtherSolve performs a symbolic execution in order to propagate in the stack the value inserted by the ORIGIN and to verify whether the value inserted influences the JUMPI at the end of the block or not.

5. Experimental validation

In this section we present the results of our empirical validation of the CFGs computed by EtherSolve. The following five research questions guide the definition of our experimental validation.

RQ$_1$ What is the success rate of EtherSolve in analyzing real-world smart-contracts compiled with different versions of the Solidity compiler?

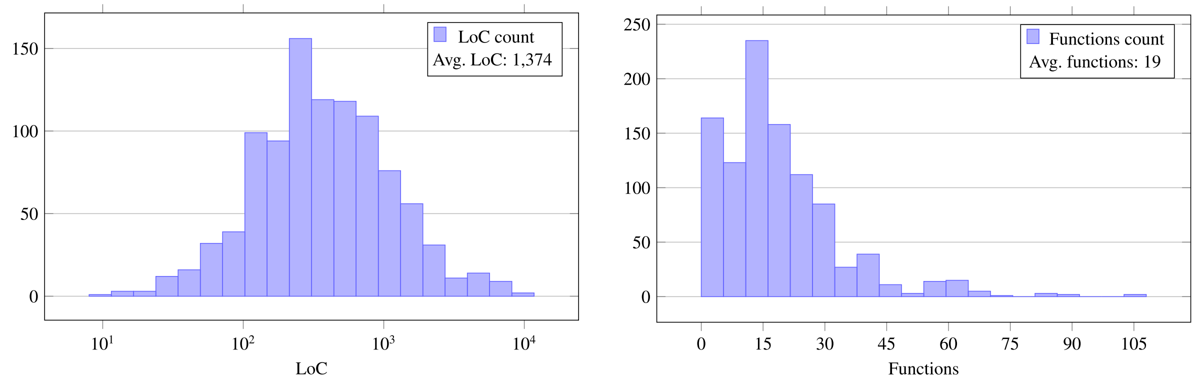


Fig. 5. Demographics of the Etherscan dataset smart-contracts: lines of code (left) and number of functions (right).

RQ$_2$ What is the accuracy of EtherSolve in identifying functions entry-point?

RQ$_3$ How precise is the smart-contracts CFG reconstruction performed by EtherSolve? How does it compare to state-of-the-art tools?

RQ$_4$ How does the Reentrancy detector built on top of EtherSolve compare to state-of-the-art source code level analysis tools?

RQ$_5$ What is the efficacy of the Tx.origin detector built on top of EtherSolve? How does it compare to state-of-the-art source code level analysis tools?

The first research question investigates the extent to which EtherSolve can process instances of existing smart-contracts with no errors. We are interested in verifying this on a wide range of smart-contracts, directly taken from the Ethereum blockchain. In the second research question we assess the accuracy of EtherSolve in finding functions entry-point, by comparing entry-points found with EtherSolve with the actual entry-points present in smart-contracts source code. The third research question compares our approach with the state-of-the-art EVM bytecode static analysis tools, w.r.t. success rate and CFG reconstruction precision. The fourth research question compares the results of the Reentrancy vulnerability detection based on EtherSolve with state-of-the-art vulnerability detection tools. Finally, the fifth research question compares the results of the Tx.origin vulnerability detection based on EtherSolve with state-of-the-art vulnerability detection tools. Note that, the last two research questions are quite challenging since EtherSolve works at the bytecode level while the considered tools work at source code level. Indeed, having the possibility to inspect the source code enhances considerably the analysis performance. We considered tools working at source code level since, to the best of our knowledge, there are no tools that detect vulnerabilities based on bytecode only.[2]

5.1. The smart-contracts datasets

Our empirical validation has been conducted using two datasets of smart-contracts. The first one, dubbed *Etherscan dataset*, is used to answer *RQ*$_1$, *RQ*$_2$ and *RQ*$_3$, and it is obtained from the list of verified contracts published by *Etherscan*[3] (Etherscan, 2020). They are publicly available open-source smart-contracts with information about compilation, deployment and transactions. From this list we have randomly extracted 1000 contracts, by using the standard random function of Python 3 (that samples from a uniform distribution) Using the APIs provided by Etherscan, both the EVM bytecode and the relevant information have been downloaded, obtaining for each smart-contract its name, address, hash, deployment date, bytecode length and compiler version. Note that, we do not use the Etherscan information during the analysis, we use it only to validate the information retrieved by EtherSolve. For instance, the compiler version is automatically retrieved by EtherSolve from the contract metadata, so the compiler version obtained from Etherscan is only used to check whether the version extracted from the metadata is correct or not. We enforced the uniqueness of smart-contracts bytecode, in order to avoid duplicates in the dataset. 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. For this reason, we computed the hash of the bytecode of each smart-contract, which has been used to delete the duplicates.

Fig. 5 shows some demographics of the dataset, computed on the source code available on Etherscan. It reports the histograms of the length of contracts solidity source code (left-hand side) and the number of public functions in their ABI (right-hand side). On average, contracts contain 1374 lines of code, with the largest one with 11,743 lines. They, on average, contain 19 named functions (i.e., excluding the fallback function, when defined), with the largest contract having 108 functions.

The average bytecode length of the smart contracts in the dataset is 7351 bytes, with a maximum length of 24,570 bytes (65 contracts exceeded the length of 20 KB). The average number of transactions of the smart contracts in the dataset is 337, with 27 contracts having 0 transactions. Note that, the Etherscan APIs allow to download at most 1000 transactions, hence for some contracts the count has been truncated (13 smart contracts in the dataset have more than 1000 transactions). The most diffused EVM version in the dataset is *Petersburg* (340 contracts), followed by *Istanbul* (210 contracts), *Byzantium* (114 contracts) and *Costantinople* (4 contracts). For some contracts compiled with old versions of `solc` (v0.4.26 and before) EtherScan does not provide the precise target EVM version, it just signals that the *Default* version is used. Unfortunately, such `solc` versions are not documented and, hence, it is not possible to deduce which is the corresponding default target EVM version. This happens for 331 contracts in the dataset. The average balance of the smart contracts in the dataset is 9.6×10^{17} Wei. As shown in Fig. 6(left), the compiler version is quite variable (with a majority of 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).

[2] Some tools work at bytecode level but use the information contained in the contract ABI. We do not consider such tools as bytecode-only.

[3] Contracts have been download on June 10, 2020.

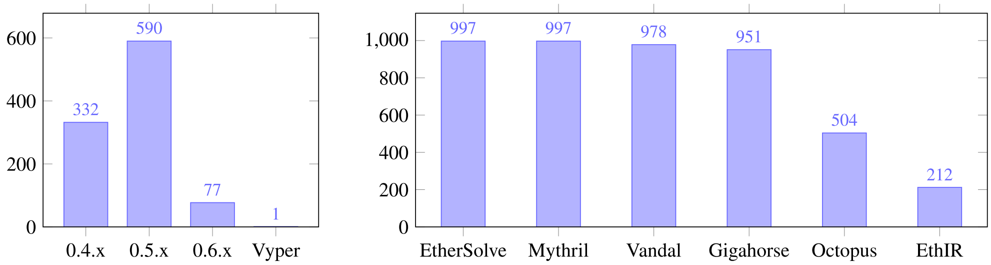


Fig. 6. Compiler versions used in the dataset contracts (left) and Success Rate for the different tools (right).

The second set of smart-contracts, dubbed *SolidiFI dataset*, consists in 50 Ethereum smart-contracts collected by Ghaleb and Pattabiraman (2020). They used the dataset to test their tool *SolidiFI*, that injects various vulnerabilities into Solidity code. We will use these smart-contracts, together with their injected versions, in order to answer RQ_4 and RQ_5.

5.2. RQ_1: Success Rate

In this research question we evaluate the number of contacts in the *Etherscan dataset* that EtherSolve is able to analyze without crashing, i.e., the *Success Rate*. The Success Rate is defined as the ratio of the smart-contracts analyzed without critical errors to the size of the dataset. EtherSolve managed to analyze all the smart-contracts in the dataset except three, obtaining a Success Rate of 0.997 (99.7%). The reason for these (very few) failures is that these three smart-contracts do not match the most common EVM bytecode patterns generated by the Solidity compiler. Indeed, one of them was written in Vyper rather than Solidity, another one was empty (with length of 0 bytes) and the last one presented an unusual begin section, that the tool failed to parse. The compiler version of Solidity is correctly identified by EtherSolve for all the analyzed contracts.

We have also performed a naive performance assessment of EtherSolve, w.r.t. analysis time. We measured the time spent by EtherSolve to parse the EVM bytecode, generate the basic blocks, solve orphan jumps and decorate the CFG. The average time is about 3 s per smart-contract. For 930 smart-contracts, the analysis took less than 1 s each, while 7 smart-contracts required more than 1 min of computation (with a maximum of 10 min), due to their big dimension and the consequent large number of edges.

> **Answer to** RQ_1
>
> *The Success Rate of EtherSolve when analyzing real-world smart-contracts compiled with different versions of the Solidity compiler is very high, since it was able to analyze without errors the 99.7% of the samples in the considered smart-contracts dataset.*

5.3. RQ_2 : Functions entry-point identification

We evaluate the accuracy of EtherSolve in detecting functions entry-point by means of well-known performance metrics such as *Precision*, *Recall* and *F-measure* (Powers and Ailab, 2011). These metrics are computed comparing the functions recognized in the EVM bytecode by EtherSolve w.r.t. the functions defined in the original Solidity source code, that are listed in the ABI of each smart-contract. The smart-contracts are taken from the *Etherscan dataset*. Given the set $\mathcal{H}_{\text{EtherSolve}}$ of function hashes identified by EtherSolve and the set $\mathcal{H}_{\text{ABI}}$ of hashes of the functions declared in the smart-contract ABI, we define: the *true positives* $T_P = \{h \mid h \in \mathcal{H}_{\text{EtherSolve}} \wedge h \in \mathcal{H}_{\text{ABI}}\}$, namely the functions declared in the ABI that EtherSolve is able to correctly identify; the *false positives* $F_P = \{h \mid h \in \mathcal{H}_{\text{EtherSolve}} \wedge h \notin \mathcal{H}_{\text{ABI}}\}$, namely the functions incorrectly detected by EtherSolve, since not present in the ABI; and the *false negatives* $F_N = \{h \mid h \notin \mathcal{H}_{\text{EtherSolve}} \wedge h \in \mathcal{H}_{\text{ABI}}\}$, namely the functions incorrectly not detected by EtherSolve, since present in the ABI. Performance metrics are computed based on the correlation between true positives and false positives/negatives.

$$\text{Precision}: \quad \mathrm{P} = \frac{|T_P|}{|T_P| + |F_P|} \qquad \text{Recall}: \quad \mathrm{R} = \frac{|T_P|}{|T_P| + |F_N|}$$

$$\text{F-measure}: \quad \mathrm{F}_1 = 2 \cdot \frac{\mathrm{P} \cdot \mathrm{R}}{\mathrm{P} + \mathrm{R}}$$

On a practical point of view, we have computed the metrics as follows. Once the functions entry-point and hashes has been identified by EtherSolve, we have downloaded the smart-contracts ABI (this can be done, because the smart-contracts in the dataset are open-source). We then proceeded with the comparison, computing Precision and Recall for each contract. When EtherSolve retrieves no function, or the ABI contains no function, we consider Precision and Recall to be 0. This analysis, however, included only 996 contracts due to internal errors of the tool during the function identification phase.

In the left of Fig. 7 we report the distribution of Precision w.r.t. our dataset, where we aggregate the contracts with the same value for Precision. The average Precision is 98.5%, and for 948 contracts the precision is 100%, meaning that the EtherSolve is able to identify all the functions for the majority of the contracts. In fact, 30 contracts have a value below 90% and only 6 contracts have a score of 0%.

In the middle of Fig. 7 we report the distribution of Recall w.r.t. our dataset, where we aggregate the contracts with the same value for Recall. The Recall is slightly lower than Precision, with an average score of 98.1% and with 811 contracts scoring 100%. This is due to the fact that the fallback detector is not perfect and sometimes fails. Nevertheless, only 32 contracts have a score below 90% and only 6 have a score of 0%.

We should note that the smart-contracts considered in the experiments have 19.2 functions on average, meaning that even a single miss in the detection would have a significant impact on Precision and Recall. Finally, in the right of Fig. 7 we compute F-measure, as a summary indicator. Overall, the results of EtherSolve are very good, with an average F-measure of 98.2% and with 803 contracts scoring 100%. Indeed, 31 contracts have a score below 90% and only 6 have a score of 0%.

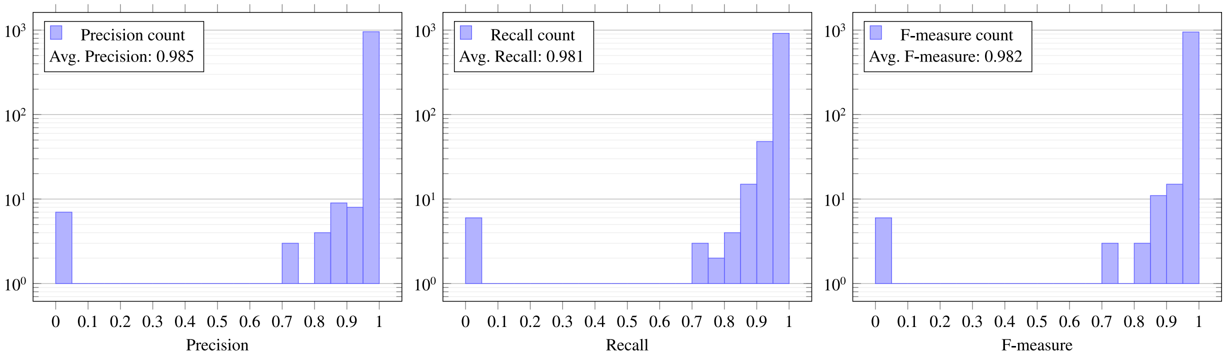


Fig. 7. Distribution of Precision, Recall and F-measure on the smart-contracts dataset (logarithmic scale).

> **Answer to** *RQ*$_2$
>
> *The accuracy of EtherSolve in identifying functions entry-point is very high, since it is able to correctly find the majority of functions declared in the smart-contracts dataset, with a very high number of true positives and very few false negatives. Indeed, EtherSolve obtained very good results for the adopted performance indicators, such as Precision* ($P = 0.985$ *on average), Recall* ($R = 0.981$ *on average) and F-measure* ($F_1 = 0.982$ *on average).*

5.4. RQ$_3$: Precision of the CFG reconstruction

To asses the precision of the CFG generated by EtherSolve, we compare our tool with the state-of-the-art analysis tools. We have selected for the comparison the tools respecting the following constraints: (i) perform a static analysis at the EVM bytecode level (no information from source code or ABI file); (ii) emit the CFG as output, or an alternative representation easily to compare with a CFG. These criteria led us to consider the following tools.

EthIR EthIR extends the Oyente framework and performs a high-level analysis of EVM bytecode. Oyente builds a CFG, to detect different kinds of vulnerabilities (Albert et al., 2018; costa-group, 2020). Due to the facts that Oyente is very old (and discontinued) and that EthIR is built on top of Oyente, we consider EthIR only for the CFGs comparison. Indeed, since EthIR is an improved version of Oyente, we can assume that the CFGs retrieved by EthIR are at least precise as those retrieved by Oyente.

Octopus Octopus is an analysis framework for EVM bytecode. It produces a CFG to support reverse engineering and understand the internal behavior of smart-contracts (Ventuzelo, 2020).

Mythril Mythril is a security analysis tool for EVM bytecode that detects security problems in smart-contracts. It does not build a CFG, but rather it generates a trace tree given by symbolic execution and Satisfiability Modulo Theories (SMT) solving (Mueller, 2020; ConsenSys, 2020).

Vandal Vandal is a static analysis framework for smart-contracts that decompiles the EVM bytecode to an intermediate representation that includes the code control flow (Brent et al., 2018; Usyd-blockchain, 2020).

Gigahorse Gigahorse is a decompiler that transforms EVM bytecode into a high-level 3-address code representation. The tool does not require the Solidity source code (Grech et al., 2019, 2018, 2020).

We discarded other analysis tools for the following reasons: *Securify* (Tsankov et al., 2018), *Ethersplay* (Crytic, 2020b), *Manticore* (Trail of Bits, 2020) and *Slither* (Crytic, 2020d) because they analyze Solidity source code instead of EVM bytecode; *evm_cfg_builder* (Crytic, 2020c) because we did not find an easy way to make it emit the CFG; *Jeb* (Software, 2020) and *MythX* (Consensys, 2020) because they are paid tools; *Porosity* because it requires the ABI as input and it is discontinued; *Panoramix* (Eveem.org, 2020) because it decompiles the code without building a CFG and it is discontinued. Finally, we did not consider the official Solidity compiler `solc`, that optionally outputs a CFG, since it exploits information contained in the source code in order to reconstruct the CFG, while EtherSolve is purely based on the bytecode. When a smart-contract is compiled to bytecode, some information useful to reconstruct the CFG is irremediably lost. Hence, it would be unfair to compare the CFGs obtained by `solc` with those computed by EtherSolve.

To confront EtherSolve with the state-of-the-art EVM bytecode static analysis tools we first compared the approaches in terms of Success Rate. We ran the tools on the smart-contracts taken from the *Etherscan dataset* and counted the successful executions (without crash) and the non-empty CFGs given in output. We also set a reasonable timeout of 10 min per contract, for all tools.

As shown in Fig. 6(right) and Table 2, EtherSolve and Mythril were able to analyze 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. Furthermore, we inspected the CFGs emitted by the tools, starting with an automatic numerical comparison of nodes and edges. Then, we proceeded with a manual inspection of 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 EVM 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.

While success rate represents a quantitative result to compare tools, the other columns will only be used for qualitative comments, rather than direct comparison. In fact, the average number of nodes and edges depend on which contracts could be

Table 2
Comparison with state-of-the-art tools.

	Success Rate	Average nodes	Average edges	Avg. blocks in static data
EtherSolve	0.997 (99.7%)	301.6	361.8	0
Mythril	0.997 (99.7%)	4.0	3.1	8.5
Vandal	0.978 (97.8%)	302.6	493.6	6.7
Gigahorse	0.951 (95.1%)	245.4	288.7	1.2
Octopus	0.504 (50.4%)	241.4	220.7	11.8
EthIR	0.212 (21.2%)	139.4	150.3	51.6

successfully analyzed. For instance, a tool able to analyze only large contracts would have an average value inflated.

It is worth observing that the different tools cannot be compared w.r.t. *analysis time*, since the candidate tools have been executed in a Docker (Docker.com, 2020) container (for compatibility reasons), instantiating a new process for each smart-contract. Whereas, EtherSolve has been executed directly with a batch of 1000 contracts.

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/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/edges that are present only in the first or in the second graph are highlighted in a different color.

Table 2 contains a summary of the automatic analysis, with the smart-contracts successfully analyzed, the average number of both nodes and edges and the average number of basic blocks in the static data segment. The results of both automatic and manual analysis are discussed in the following paragraphs, one for each tool.

5.4.1. EthIR

In most cases, EthIR finds more blocks than EtherSolve, that have a high offset.

Considerations: Nodes that are found by EthIR only have a very high offset, so they are probably static data or metadata interpreted as code (Table 2). Instead, when nodes match then edges match as well.

Manual analysis: Because of its very low Success Rate (21%), we deemed not so interesting to continue with a manual analysis of this tool results.

5.4.2. Octopus

Similarly to EthIR, also Octopus finds more blocks with a high offset than EtherSolve.

Considerations: Similarly to the previous tool, also for Octopus we speculate that the additional nodes found are probably data or metadata and not code (Table 2). In the majority of cases, however, Octopus finds very few edges, sometimes even 0 edges.

Manual analysis: During the manual analysis, we discovered that Octopus misses some patterns for the metadata separation, so metadata are parsed as if it were code. Moreover, in many cases, Octopus does not detect edges that should be present according to source code.

As an example, in Fig. 8 we have an excerpt of the diff graph for the smart-contract ZipmexTokenP (address: 0xaa602de53347579f86b996d2add74bb6f79462b2). In this case, Octopus exhibits a lack of edges, found by EtherSolve, that should be present according to the Solidity source code. They are often return-edges, i.e., edges that go from the last block of a function to the block that follows the call. In the EVM, these types of edges originate from orphan jumps, meaning that Octopus cannot always resolve them. In some other contracts, Octopus evaluates the bytecode as creation code, thus it analyzes only the second part, starting with 60806040 (the most common begin sequence of a contract bytecode). However, this part of the code is a child contract and not the main one, so the computed graphs are completely different.

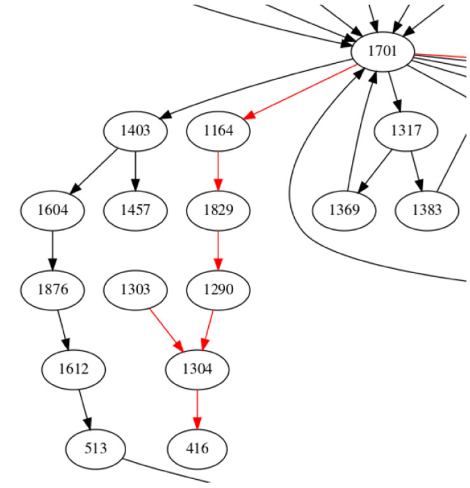


Fig. 8. An excerpt of the diff graph for the ZipmexTokenP smart-contract used for manual inspection. In red we highlight nodes/edges found by EtherSolve but not by Octopus; in blue we highlight nodes/edges found by Octopus but not by EtherSolve; in black we highlight nodes/edges found by both tools. Basic blocks are identified by the offset of their first instruction. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.4.3. Mythril

This tool 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, an indirect comparison can be performed by checking if the EtherSolve CFG misses nodes/edges that are found by the Mythril dynamic analysis. This case would correspond to imprecision in the CFG elaborated by EtherSolve.

Manual analysis: 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 analyzed code, but they indicate the end of the execution trace. 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. 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. The child contract code is computed at runtime, so the EtherSolve static analysis simply considers those bytes as part of the static data segment.

5.4.4. Vandal

The CFGs generated by Vandal have always one node more than EtherSolve, which is the ending node with the INVALID opcode. In the 36% of the analyzed contracts the edges match, but in the remaining cases there are differences that we analyzed 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 (Swende, 2020). Indeed, the SELFBALANCE opcode is considered as invalid by Vandal, obtaining a basic block break with no outgoing edges. In many other cases, Vandal detects a huge amount of edges.

Considerations: Probably, when Vandal is not able to correctly compute the destination address of a jump, it conservatively assumes all basic blocks as possible successors.

Manual analysis: 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. Indeed, as we can see in Fig. 9, where we have an excerpt of the diff graph for the smart-contract BCN20 (address: 0x1964f2f3ce45ac518b18ef4aa4265f8aadcef4ae), Vandal finds too many outgoing edges from basic blocks which are the end of functions (e.g., 770, 468 and 1402).

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 2, the average number of edges found by Vandal is significantly higher than EtherSolve, whereas the basic blocks are reconstructed in a similar way.

5.4.5. Gigahorse

Gigahorse 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, Gigahorse CFGs contain artificial blocks with the special "CALLPRIVATE" statement, introduced by Gigahorse to mark private function calls. Nevertheless, the data collected by the automatic comparison for Gigahorse and EtherSolve is very similar, so we proceeded with the manual inspection.

Manual analysis: For some contracts, 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 CFGs. A dual case happens on some other contracts, 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.

Considerations: Our speculation is that such basic blocks (which have a high offset) belong to a child contract or to an internal library, not a proper part of the main contract (e.g., called via STATICCALL), and thus they are skipped by EtherSolve (that only analyzes intra-contract calls).

Answer to RQ_3

The precision of the CFG reconstruction performed by EtherSolve is very high, since the CFGs reconstructed by EtherSolve have a comparable, and sometimes higher, precision w.r.t. the CFGs reconstructed by the state-of-the-art EVM bytecode static analysis tools. Furthermore, EtherSolve is the approach with the highest Success Rate (99.7%), on par with Mythrill, among the state-of-the-art analysis tools.

5.5. RQ_4: Reentrancy vulnerabilities detection

In order to validate the efficacy of the vulnerability detector built on top of EtherSolve, we compared it with specialized tools aiming at discovery Reentrancy vulnerabilities. The comparison is performed on the *SolidiFI dataset*, that consists in 50 Ethereum smart-contracts whose source code has been injected, using the tool *SolidiFI* (Ghaleb and Pattabiraman, 2020), with Reentrancy vulnerabilities. In total, 1343 bugs have been injected, coming from 42 code snippets. This benchmark has been used by Ghaleb and Pattabiraman (2020) to compare the most prominent vulnerability detection tools. While their comparison was performed at source code level, EtherSolve targets the EVM bytecode, so these injected contracts have been compiled before applying our analysis. Even if 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 analyze only one compiled contract per source file (the largest compiled contract), assuming that the remaining contracts were only supporting libraries or abstract contracts, whose CFGs were disconnected from the main one.

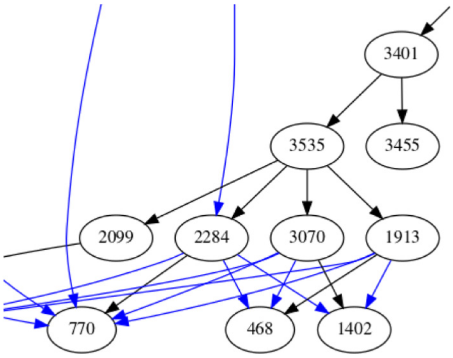


Fig. 9. An excerpt of the diff graph for the BCN20 smart-contract used for manual inspection. In red we highlight nodes/edges found by EtherSolve but not by Vandal; in blue we highlight nodes/edges found by Vandal but not by EtherSolve; in black we highlight nodes/edges found by both tools. Basic blocks are identified by the offset of their first instruction. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Another problematic aspect, acknowledged by Ghaleb and Pattabiraman, is that the dataset already contained vulnerabilities before the SolidiFI injection, but they were not documented. To gather comparable results, we considered only vulnerabilities added by the SolidiFI injection, by running EtherSolve before and after injection, and by keeping only those new vulnerabilities that are detected by the second analysis and not by the first one. We have taken an analogous approach for the other tools: we ran each tool on the original dataset (without injections) and on the injected contracts, then we have considered as Reentrancy bugs those present in the injected contracts but not in the original contracts.

Concerning tools selection for the comparison, we have considered: *Mythril* (Mueller, 2020), *Securify* (Tsankov et al., 2018), *Slither* (Crytic, 2020d), *Vandal* (Brent et al., 2018) and *SmartCheck* (Tikhomirov et al., 2018). They are the state-of-the-art static analysis tools that are able to detect Reentrancy vulnerabilities. We had to exclude from the comparison *Oyente* (Luu et al., 2016) and *Manticore* (Trail of Bits, 2020) since we did not manage to make the tools working (they are very old and discontinued). Recall that, the considered approaches inspect the Solidity source code of smart-contracts, not the EVM bytecode, as happens for EtherSolve.

Fig. 10 shows the results of the comparison, subdivided in six plots, one for each tool under comparison. In each plot we have the number of bugs injected per contract (numbered from 1 to 50) and the number of bugs found by a analysis tool per contract. In other words, for each tool we plot the distribution of bugs found in the smart-contracts, compared with the baseline

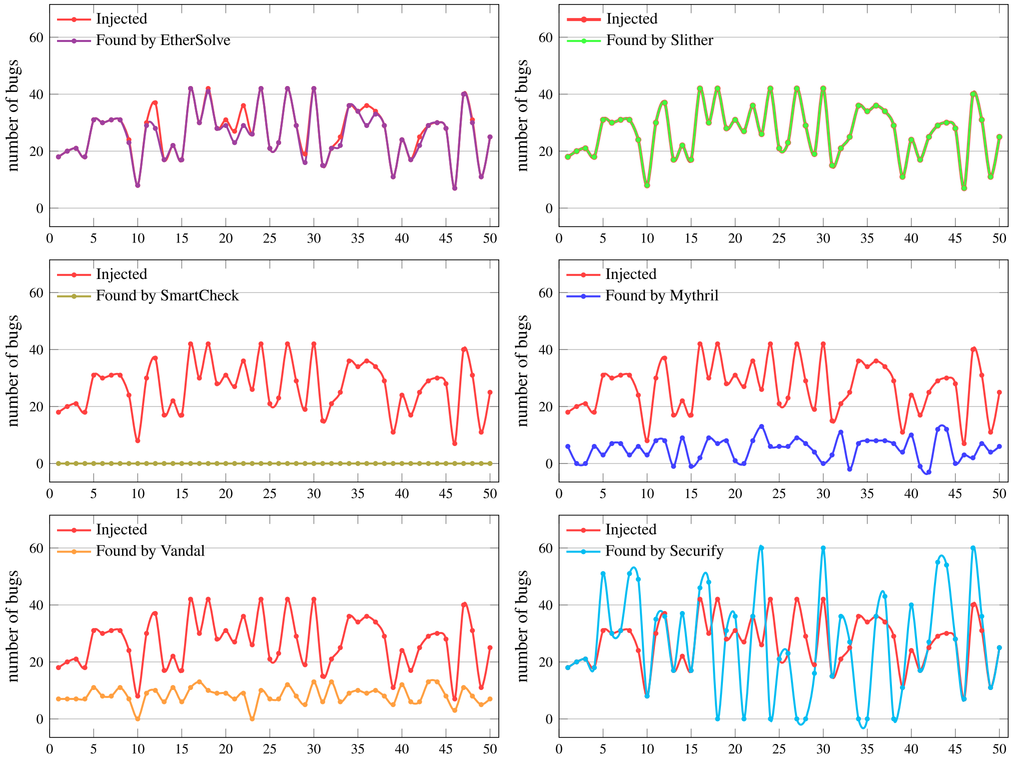


Fig. 10. Reentrancy analysis results for the different tools (smart-contracts on the x-coordinate).

distribution of injected bugs. The efficacy of a tool increases as the distance between its plot and the baseline distribution decreases. Furthermore, the points above the baseline distribution indicate potential false positives (since the tool find more bugs than the ones that have been injected), while the points below the baseline distribution indicate potential false negatives (since the tool find less bugs than the ones that have been injected).

The results highlight that EtherSolve is the second-best analysis tool, immediately after Slither. Indeed, the plot for EtherSolve follows very closely the baseline distribution, except for few contracts where EtherSolve misses some bugs (i.e., we have false negatives). Slither, instead, performs very well since its plot coincide exactly with the baseline distribution, meaning that Slither do not have false positives nor false negatives. Conversely, SmartCheck have the worst plot, indeed it is not able to find any of the injected bugs (this has been highlighted also by the experimental evaluation of Ghaleb and Pattabiraman (2020)).

Even if EtherSolve exhibits slightly lower results w.r.t. Slither, we have to recall that it is the only tool using EVM bytecode only, thus having less information. Inspecting the smart-contracts source code is of great help in improving bug detection results. So, EtherSolve results are even more valuable: it is able to detect Reentrancy vulnerabilities even in closed-source smart-contracts, with very high efficacy. Its results are comparable, and very often superior, to analysis tools that can also exploit smart-contracts source code.

Answer to *RQ*$_4$

The efficacy of the Reentrancy vulnerability detector built on top of EtherSolve is very high when compared with state-of-the-art tools, because it is the second-best, in terms of detected Reentrancy bugs, among the state-of-the-art analysis tool, even if it is the only one that does not exploit smart-contracts source code.

5.6. *RQ*$_5$: *Tx.origin vulnerabilities detection*

Similarly to the previous section, in order to validate the efficacy of the vulnerability detector built on top of EtherSolve we compared it with specialized tools aiming at discovery Tx.origin vulnerabilities, again using the *SolidiFI dataset*. For this research question, the 50 Ethereum smart-contracts has been injected with Tx.origin vulnerabilities. In total, 1336 bugs have been injected, coming from 40 code snippets. Also in this case, we had to compile the injected contracts before applying the analysis, and we used one compiled contract per source file (the largest one). Furthermore, we considered only vulnerabilities added by the SolidiFI injection, by running EtherSolve before and after injection, and by keeping only those new vulnerabilities that are detected by the second analysis and not by the first one. The same applies for the other tools.

Concerning tools selection for the comparison, we have considered: *Mythril* (Mueller, 2020), *Slither* (Crytic, 2020d), *SmartCheck* (Tikhomirov et al., 2018) and *Vandal* (Brent et al., 2018). They are the state-of-the-art static analysis tools that are able to detect Tx.origin vulnerabilities. Note that, tools such as *Oyente* (Luu et al., 2016), *Manticore* (Trail of Bits, 2020) and *Securify* (Tsankov et al., 2018) are not designed to find Tx.origin vulnerabilities. Also in this case, the considered approaches inspect the Solidity source code of the smart-contracts, not the EVM bytecode, as happens for EtherSolve.

Fig. 11 shows the results of the comparison, subdivided in five plots, one for each tool under comparison. Analogously to the previous section, for each tool we plot the distribution of bugs found in the smart-contracts, compared with the baseline distribution of injected bugs. The efficacy of a tool increases as the distance between its plot and the baseline distribution decreases. Again, points above the baseline distribution are considered false positives, while points below the baseline distribution are considered false negatives.

Also in this case, the results highlight that EtherSolve is the second-best analysis tool, again immediately after Slither. Indeed,

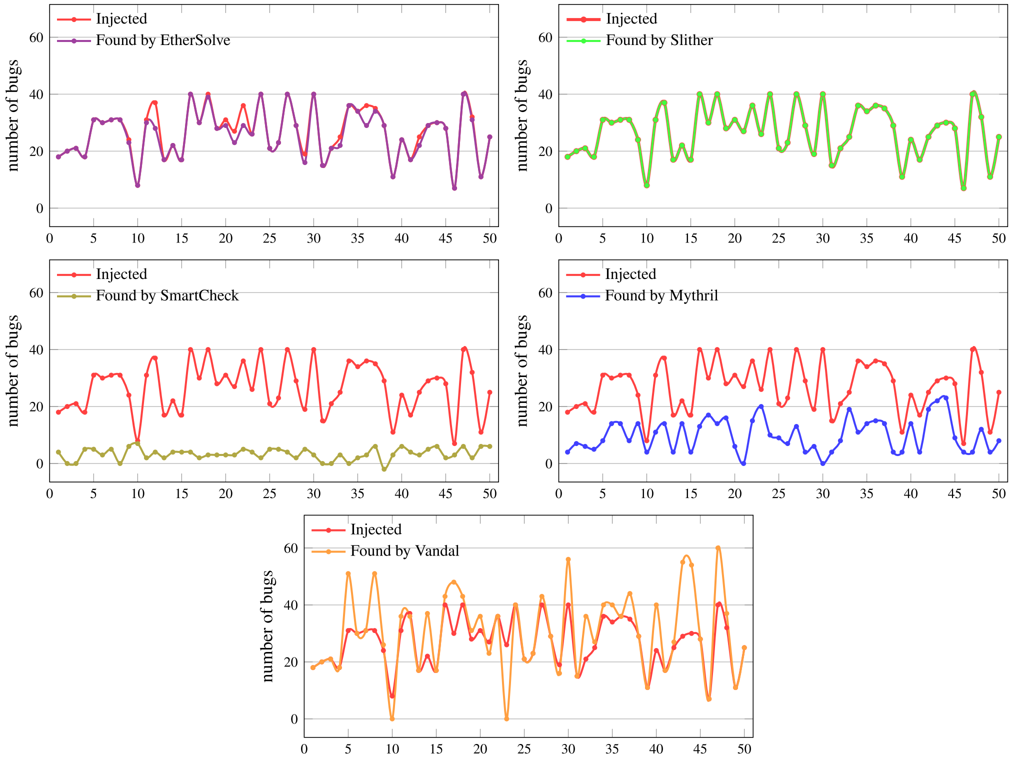


Fig. 11. Tx.origin analysis results for the different tools (smart-contracts on the x-coordinate).

the plot for EtherSolve follows very closely the baseline distribution, except for few contracts where EtherSolve misses some bugs (i.e., we have false negatives). Slither, instead, performs very well, since its plot coincide exactly with the baseline distribution, meaning that Slither do not have false positives nor false negatives. The worst results are obtained by SmartCheck, that is not able to detect the majority of the injected bugs. Vandal results are quite good in general, except for some contracts for which it signals either lots of false positives or lots of false negatives.

Even if EtherSolve exhibits slightly lower results w.r.t. Slither, we recall that it is the only tool using EVM bytecode only. So, EtherSolve results are even more valuable: it is able to detect Tx.origin vulnerabilities even in closed-source smart-contracts, with very high efficacy.

Answer to *RQ*$_5$

The efficacy of the Tx.origin vulnerability detector built on top of EtherSolve is very high, since the number of false positives and false negatives is very low. Indeed, it is the second-best, in terms of detected Tx.origin bugs, among the state-of-the-art analysis tool, even if it is the only one that does not exploit smart-contracts source code.

5.7. Discussion

The results obtained in the experimental validation suggest that EtherSolve reconstructs very precise CFGs: 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 capable of resolving the destination address of *orphan jumps*. However, there are particular cases of very complex smart-contracts with peculiar structures for which EtherSolve is not able to identify certain edges. For instance, the new *try-catch* feature introduced in the Solidity 0.6.x versions, is translated in a particular bytecode structure. EtherSolve is able to deal with such bytecode, but its particular structure deceives the tool, inducing it to take wrong execution paths, causing little internal exceptions.

Among the compared tools, only Gigahorse showed a CFG precision similar to EtherSolve. However, they seems to be complementary, because each one could precisely represent cases that the other one could not.

The results of the vulnerability detectors 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 experimental evaluation highlights that EtherSolve analysis results, in terms of false positives and false negatives, are second only to Slither results, for both Reentrancy and Tx.origin vulnerabilities. This is a very valuable, since Slither works at source code level while EtherSolve works at bytecode level, hence with way less information about the smart-contracts under analysis.

EtherSolve is affected by some limitations. One of them is in the way we analyze loops. In fact, to prevent the analysis to iterate forever, we pose a limit to the number of iterations by enforcing a maximum size to the symbolic stack that we propagate. This might in principle cause the final result to be imprecise. However, in our experience, in those cases when the size limit is reached, a larger limit would not help in delivering a better result, so a larger limit would not improve the precision of the tool. A possible improvement that may mitigate the issue consists in developing an analysis to detect when a loop recursively increases the symbolic stack size, and cut-off such paths.

6. Related work

Comparison with the conference paper. The present paper is a deeply revised and extended version of the companion conference paper (Contro et al., 2021). In particular, we better explained the problem (Section I of Contro et al. (2021)) and we better described the background material (Section II of Contro et al. (2021)). We also added the description of the Tx.origin vulnerability (with examples) in the Background. Furthermore, we improved the overall presentation of the proposed static analysis that reconstructs the smart-contracts CFG (Section III of Contro et al. (2021)). We revised the experimental validation (Section IV of Contro et al. (2021)), performing a more detailed assessment of the Reentrancy detector built on top of EtherSolve, comparing its Reentrancy detection efficacy with the state-of-the-art source code level analysis tools. As novel contributions, we extended EtherSolve with a mechanism to identify functions entry-point and a specific analysis detecting Tx.origin vulnerabilities. Consequently, we extended the experimental validation, adding an assessment of the accuracy of EtherSolve in identifying functions entry-point and a comparison of Tx.origin detection efficacy with the state-of-the-art source code level analysis tools.

Related approaches. In the last years, many tools have been developed in order to analyze Ethereum smart-contracts, with different approaches and objectives. A quite recent and detailed survey of them has been written by Praitheeshan et al. (2019). Some tools analyze directly the EVM bytecode, often trying to build a CFG. Among these tools we can find *Oyente* (Luu et al., 2016), *EthIR* (Albert et al., 2018) and *Octopus* (Ventuzelo, 2020). Their approaches are very similar, as they try to execute symbolically the code to create logic predicates which, once resolved with theorem prover such as the *Z3* (Z3Prover, 2020), can determine the destinations of orphan jumps. However, these tools aim at detecting vulnerabilities, and the extracted CFG is only an intermediate output.

A slightly different approach is the one proposed by *Vandal* (Brent et al., 2018), which translates the EVM bytecode into registry based operations, identifies the basic blocks and then tries to resolve jumps address through a fixed point analysis. Even in this case the CFG is only an intermediate output, as the target of Vandal is a vulnerability analysis based on the *Souffle suite* (Jordan et al., 2016). 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 and resulting in a more precise CFG reconstruction.

A related tool which extracts CFGs from bytecode is *Jeb* (Software, 2020), a professional decompiler with the ability to analyze Ethereum smart-contracts. However, it is closed-source with a subscription fee. Another decompiler is *Porosity* (Comaeio, 2020), one of the first tool developed to analyze EVM bytecode, but it needs the contract ABI to work properly. Furthermore, it is discontinued since January 2018. A relevant decompiler is *Gigahorse* (Grech et al., 2019, 2018, 2020), a recent tool which builds a CFG and tries to find internal functions with heuristics, obtaining an approximation of the original Solidity source code. Another decompiler is *Panoramix* (Eveem.org, 2020) which, however, does not generate a CFG.

The majority of tools that perform vulnerability analysis of Ethereum smart-contracts 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 EVM bytecode together with additional information that are not always available for closed-source contracts.

A completely different approach is the one implemented by *Mytril* (ConsenSys, 2020), which uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities. 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* (Crytic.io, 2020) is an application that collects many tools for smart-contract analysis, such as *Manticore* (Trail of Bits, 2020), *Ethersplay* (Crytic, 2020b), *Echidna* (Crytic, 2020a), *Slither* (Crytic, 2020d) and more, but they do not use a CFG or they do not analyze the bytecode only. In fact their objective is the vulnerabilities detection inside the Solidity source code.

Finally, there are other tools such as *Securify* (Tsankov et al., 2018), which analyzes Solidity source code, *Maian* (Nikolić et al., 2018), which performs dynamic analysis on a private blockchain, and *Gasper* (Chen et al., 2017), which analyzes the gas cost of contracts. Furthermore, in Albert et al. (2021) the authors define a symbolic execution-based gas analysis built on top of the notion of stack-sensitive CFG (S-CGF). The latter provides a sound approximation of a CFG, based on a sound static analysis (based on abstract interpretation in Albert et al. (2021)), as described in the cited technical report (Albert et al., 2020). Hence, even if with S-CFGs the authors aim at solving the same problem as our tool, the methodologies are different: with S-CFGs they compute a sound approximation of the CFGs, based on a sound static analysis; while EtherSolve computes precise (but not necessarily sound) CFGs, by means of symbolic execution. Hence, the paper (Albert et al., 2021) does not use symbolic execution to compute CFGs (as EtherSolve) but only to perform the gas analysis (on top of a previously computed S-CFG). In Grossman et al. (2017) the authors define correctness criteria (ECF) for callbacks, in a general setting, and by means of such criteria they encode Reentrancy-like bugs for Solidity. Then, the authors provide a monitoring technique to dynamically spot ECF, based on an stack-based intermediate language. Such contribution is not related to EtherSolve, that aims to reconstruct precise CFGs from EVM bytecode and that uses Reentrancy as a simple application scenario.

7. Conclusion

Automatically analyze Ethereum smart-contracts is crucial in order to detect potential defects and vulnerabilities. Nevertheless, most of the existing analysis tools for the EVM bytecode come with shortcomings and limitations. Indeed, the precise extraction of the CFG from the EVM bytecode is very challenging (e.g., resolving the target of jumps), due to engineering decisions concerning the underlying infrastructure. For this reason, most of the state-of-the-art analysis tools compute imprecise CFG or use alternative code representations, resulting in poor analysis performance.

We propose a novel approach to extract a precise CFG from the EVM bytecode. We believe that our solution 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. To validate our approach we have implemented a tool, EtherSolve, and compared it on the state-of-the-art analysis tools using a CFG-based code representation. Furthermore, we have added to EtherSolve the capability to detect two of the most prominent Ethereum smart-contracts vulnerabilities (Reentrancy and Tx.origin) and compared its analysis precision with the state-of-the-art Ethereum analyzers, obtaining excellent results.

CRediT authorship contribution statement

Michele Pasqua: Writing – original draft, Formal analysis, Investigation, Validation. **Andrea Benini:** Software, Investigation. **Filippo Contro:** Software, Investigation. **Marco Crosara:** Software, Investigation. **Mila Dalla Preda:** Conceptualization, Resources. **Mariano Ceccato:** Writing – review & editing, Supervision, Investigation, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A., 2020. Analyzing smart contracts: From EVM to a sound control-flow graph. CoRR abs/2004.14437. URL: https://arxiv.org/abs/2004.14437, arXiv:2004.14437.

Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A., 2021. Don't run on fumes – Parametric gas bounds for smart contracts. J. Syst. Softw. 176, 110923. http://dx.doi.org/10.1016/j.jss.2021.110923, URL: https://www.sciencedirect.com/science/article/pii/S0164121221000200.

Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I., 2018. EthIR: A framework for high-level analysis of ethereum bytecode. In: Lahiri, S.K., Wang, C. (Eds.), Automated Technology for Verification and Analysis. Springer International Publishing, Cham, pp. 513–520.

Antonopoulos, A.M., 2017. Mastering Bitcoin: Programming the Open Blockchain, second ed. O'Reilly Media, Inc..

Antonopoulos, A., Wood, G., Wood, G., 2018. Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media, Incorporated, URL: https://books.google.it/books?id=SedSMQAACAAJ.

Baliga, A., 2017. Understanding blockchain consensus models.

Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B., 2018. Vandal: A scalable security analysis framework for smart contracts. arXiv:1809.03981.

Buterin, V., 2015. A next-generation smart contract and decentralized application platform.

Chambers, C., 2020. Forbes - Ethereum starts its DeFi moon shot. URL: https://www.forbes.com/sites/investor/2020/07/23/ethereum-starts-its-defi-moon-shot/#7d34b7ff6ae3. [Accessed: 2020-07-28].

Chen, T., Li, X., Luo, X., Zhang, X., 2017. Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 442–446. http://dx.doi.org/10.1109/SANER.2017.7884650.

Coindesk, 2020. Soaring DeFi usage drives ethereum contract calls to new record. URL: https://www.coindesk.com/soaring-defi-usage-drives-ethereum-contract-calls-to-new-record. [Accessed: 2020-07-28].

Comaeio, 2020. Github - Porosity. URL: https://github.com/comaeio/porosity. [Accessed: 2020-07-20].

ConsenSys, 2020. Github - Mythril. URL: https://github.com/ConsenSys/mythril. [accessed: 2020-07-07].

Consensys, 2020. MythX: Smart contract security service for ethereum. URL: https://mythx.io/. [Accessed: 2020-07-28].

Contro, F., Crosara, M., Ceccato, M., Dalla Preda, M., 2021. EtherSolve: Computing an accurate control-flow graph from ethereum bytecode. In: 29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021. IEEE, pp. 127–137. http://dx.doi.org/10.1109/ICPC52881.2021.00021.

costa-group, 2020. Github - EthIR. URL: https://github.com/costa-group/EthIR#ethir. [accessed: 2020-07-07].

Crytic, 2020a. Echidna: A fast smart contract fuzzer. URL: https://github.com/crytic/echidna. [Accessed: 2020-07-28].

Crytic, 2020b. Ethersplay. URL: https://github.com/crytic/ethersplay. [Accessed: 2020-07-28].

Crytic, 2020c. EVM CFG builder. URL: https://github.com/crytic/evm_cfg_builder. [Accessed: 2020-07-28].

Crytic, 2020d. Slither, the Solidity source analyzer. URL: https://github.com/crytic/slither. [Accessed: 2020-07-28].

Crytic.io, 2020. Crytic: continuous assurance for smart contracts. URL: https://crytic.io/. [Accessed: 2020-07-20].

Dannen, C., 2017. Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, first ed. A Press, USA.

Dika, A., Nowostawski, M., 2018. Security vulnerabilities in ethereum smart contracts. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). pp. 955–962. http://dx.doi.org/10.1109/Cybermatics_2018.2018.00182.

Docker.com, 2020. Docker. URL: https://www.docker.com/. [Accessed: 2020-07-28].

Ethereum, 2020a. Solidity documentation. URL: https://solidity.readthedocs.io/. [accessed: 2020-07-02].

Ethereum, 2020b. Solidity documentation - Creating contracts. URL: https://docs.soliditylang.org/en/v0.8.0/contracts.html#creating-contracts. [accessed: 2020-12-29].

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

Etherscan.io, 2020. Etherscan. URL: https://etherscan.io/. [accessed: 2020-07-02].

Ethervm.io, 2020. Ethervm. URL: https://ethervm.io/. [accessed: 2020-07-02].

Eveem.org, 2020. Panoramix. URL: https://github.com/eveem-org/panoramix. [accessed: 2020-12-08].

Ghaleb, A., Pattabiraman, K., 2020. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 415–427, URL: https://doi.org/10.1145/3395363.3397385.

Grech, N., Brent, L., Scholz, B., Smaragdakis, Y., 2019. Gigahorse: Thorough, declarative decompilation of smart contracts. In: Proceedings of the 41st International Conference on Software Engineering. ICSE '19, IEEE Press, pp. 1176–1186, URL: https://doi.org/10.1109/ICSE.2019.00120.

Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y., 2018. MadMax: Surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Program. Lang. 2 (OOPSLA), URL: https://doi.org/10.1145/3276486.

Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y., 2020. MadMax: Analyzing the out-of-gas world of smart contracts. Commun. ACM 63 (10), 87–95, URL: https://doi.org/10.1145/3416262.

Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y., 2017. Online detection of effectively callback free objects with applications to smart contracts. Proc. ACM Program. Lang. 2 (POPL), http://dx.doi.org/10.1145/3158136.

Jordan, H., Scholz, B., Subotić, P., 2016. Soufflé: On synthesis of program analyzers. In: Chaudhuri, S., Farzan, A. (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 422–430.

Li, X., Chen, T., Luo, X., Zhang, T., Yu, L., Xu, Z., 2020. STAN: Towards describing bytecodes of smart contract. In: 20th IEEE International Conference on Software Quality, Reliability and Security, QRS 2020, Macau, China, December 11-14, 2020. IEEE, pp. 273–284, URL: https://doi.org/10.1109/QRS51102.2020.00045.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, Association for Computing Machinery, New York, NY, USA, pp. 254–269, URL: https://doi.org/10.1145/2976749.2978309.

Mueller, B., 2020. Smashing ethereum smart contracts for fun and real profit. URL: https://github.com/muellerberndt/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf. [accessed: 2020-12-08].

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A., 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proc. of the 34th Annual Computer Security Applications Conference. ACSAC '18, ACM, New York, NY, USA, pp. 653–663, URL: https://doi.org/10.1145/3274694.3274743.

Powers, D., Ailab, 2011. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation. J. Mach. Learn. Technol. 2, 2229–3981. http://dx.doi.org/10.9735/2229-3981.

Praitheeshan, P., Pan, L., Yu, J., Liu, J., Doss, R., 2019. Security analysis methods on ethereum smart contract vulnerabilities: A survey. arXiv:arXiv:1908.08605.

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

Pulse, D., 2021. Defi total value locked. URL: https://defipulse.com/. [Accessed: 2021-10-04].

Software, E., 2020. Jeb - Ethereum contract decompiler. URL: https://www.pnfsoftware.com/jeb/evm. [Accessed: 2020-07-27].

Swende, M.H., 2020. EIP 1884: Repricing for trie-size-dependent opcodes. URL: https://eips.ethereum.org/EIPS/eip-1884. [Accessed: 2020-07-28].

Tabora, V., 2020. The Ethereum Virtual Machine (EVM) runtime environment. URL: https://medium.com/0xcode/the-ethereum-virtual-machine-evm-runtime-environment-d7969544d3dd. [Accessed: 2020-07-15].

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y., 2018. SmartCheck: Static analysis of Ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. WETSEB '18, Association for Computing Machinery, New York, NY, USA, pp. 9–16, URL: https://doi.org/10.1145/3194113.3194115.

Trail of Bits, 2020. Manticore. URL: https://github.com/trailofbits/manticore. [Accessed: 2020-07-28].

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M., 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS '18, Association for Computing Machinery, New York, NY, USA, pp. 67–82.

Usyd-blockchain, 2020. Github - Vandal. URL: https://github.com/usyd-blockchain/vandal. [accessed: 2020-07-07].

Ventuzelo, P., 2020. Github - Octopus. URL: https://github.com/pventuzelo/octopus. [accessed: 2020-07-07].

Wood, G., 2014. Ethereum: A secure decentralised generalised transaction ledger.

Z3Prover, 2020. Github - z3. URL: https://github.com/Z3Prover/z3. [Accessed: 2020-07-07].

Zheng, Z., Xie, S., Dai, H.-N., Chen, W., Chen, X., Weng, J., Imran, M., 2020. An overview on smart contracts: Challenges, advances and platforms. Future Gener. Comput. Syst. 105, 475–491, URL: http://dx.doi.org/10.1016/j.future.2019.12.019.

Michele Pasqua received the B.Sc. and M.Sc. degrees in Computer Science and Engineering from the University of Verona (Italy) in 2013 and 2015, respectively, and the Ph.D. in Computer Science from the University of Verona (Italy) in 2019. He was a Postdoctoral Researcher at the University of Verona (Italy) from 2019 to 2020, working on security aspects of IoT systems. He was a Postdoctoral Researcher at the University of Udine (Italy), from 2020 to 2021, working on new decentralized programming paradigms for the IoT. He is currently an Assistant Professor at the University of Verona (Italy), working in the department of Computer Science. His research interests focus on the security of computational systems, programs verification (based on abstract interpretation), mathematical foundations of computer science, programming languages design, cybersecurity and software testing.

Andrea Benini is an IT Engineer, working as a web developer in an IT consulting company since my graduation in 2021. He studied Computer Science and Engineering at University of Verona, specializing in cyber-security. During these years he has been interested in blockchain and he was able to contribute to the development of some projects such as Hotmoka, Commercio.network, Ethersolve. Network systems, along with cryptography,are the things that fascinate him the most and what stimulates him are the technological innovations that are constantly developing around this area. He loves programming and using technology to improve his life and the lives of others.

Filippo Contro is a software engineer at Bit Mobility, where he is mainly focused on the development of a software layer for shared mobility. He got his master degree in Computer Science in 2020 at university of Verona with the thesis: "EtherSolve: computing an accurate control-flow graph from Ethereum bytecode". He is interested in software analysis and software engineering. He loves to think about structures and design principles to develop better code, to help both developers and improve code quality.

Marco Crosara is currently working as a programmer in an IT consulting company after having obtained the master's degree in Computer Engineering at the University of Verona, in March 2021. He loves programming and creating software that solves the problems that are submitted to him. He is motivated by the satisfaction of creating innovative software solutions that are useful to him and others. His great passion is blockchain technology,which has been his main area of research for the last few years of his student's career, supporting standard studies up to the thesis day after day. Among his best skills, there is certainly a great inventiveness and the ability to see a problem from new points of view without ever forgetting the final goal.

Mila Dalla Preda is an Assistant Professor at the Dipartimento di Informatica, University of Verona. She had the Computer Science Laurea degree from the University of Verona in 2003 and doctoral degree in Computer Science in 2007. She has been active in various research projects and groups devoted to the design and development of automatic tools based on formal methods and abstract interpretation theory for program analysis, automatic program certification, system testing, and security analysis of software. She has been the principal investigator of the Italian FIRB project FACE: Formal Avenue for Change Malware (2013–2018). She was one of the organizers of the Dagstuhl Seminar "Software Protection Decision Support and Evaluation Methodologies" from August 11th to August 16th 2019. She served in the program committee of major international conferences including Compiler Construction CC, Symposium on Foundations & Practice of Security FPS, Principles of Programming Languages POPL, International Symposium on Code Generation and Optimization CGO, Programming Language Design and Implementation PLDI. She has over 40 publications related to software security, program analysis, malware detection, and abstract interpretation.

Mariano Ceccato is Associate Professor in the Computer Science department in University of Verona, Italy. Until 2019, he was tenured researcher in the Security & Trust and in the Software Engineering research units in Fondazione Bruno Kessler, Trento, where was principal investigator of several publicly funded research projects. He received the PhD in Computer Science from the University of Trento in 2006 with the thesis "Migrating Object Oriented code to Aspect Oriented Programming". He is author or coauthor of more than 70 research papers published in international journals and conferences/workshops. He was recently visiting research scientist in the Software Verification and Validation Laboratory Centre for ICT Security, Reliability, and Trust (SnT), University of Luxembourg. His research interests include security testing, penetration testing, code hardening and empirical studies.