Towards a Sound Construction of EVM Bytecode Control-Flow Graphs

Vincenzo Arceri
vincenzo.arceri@unipr.it
University of Parma
Parma, Italy

Saverio Mattia Merenda
saveriomattia.merenda@studenti.unipr.it
University of Parma
Parma, Italy

Greta Dolcetti
greta.dolcetti@unive.it
Ca' Foscari University of Venice
Venice, Italy

Luca Negrini
luca.negrini@unive.it
Ca' Foscari University of Venice
Venice, Italy

Luca Olivieri
luca.olivieri@unive.it
Ca' Foscari University of Venice
Venice, Italy

Enea Zaffanella
enea.zaffanella@unipr.it
University of Parma
Parma, Italy

Abstract

Ethereum enables the creation and execution of decentralized applications through smart contracts, that are compiled to Ethereum Virtual Machine (EVM) bytecode. Once deployed in the blockchain, the bytecode is immutable; hence, ensuring that smart contracts are bug-free before their deployment is of utmost importance. A crucial preliminary step for any effective static analysis of EVM bytecode is the extraction of the control-flow graph (CFG): this presents significant challenges due to potentially statically unknown jump destinations. In this paper we present a novel approach, based on abstract interpretation, aiming at building a sound CFG from EVM bytecode smart contracts. Our analysis, which is implemented in our static analyzer EVMLiSA, is based on a parametric abstract domain that approximates concrete execution stacks at each program point as an l-sized set of abstract stacks of maximal height h; the results of the analysis are then used to resolve the jump destinations at jump nodes. In our preliminary experiments, by fine-tuning the analysis parameters, EVMLiSA builds sound CFGs for all smart contracts where permanent storage-related opcodes do not influence jump destinations.

CCS Concepts

• **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Automated static analysis**.

Keywords

Static analysis, Abstract interpretation, Smart contracts, EVM bytecode, Ethereum

ACM Reference Format:
Vincenzo Arceri, Saverio Mattia Merenda, Greta Dolcetti, Luca Negrini, Luca Olivieri, and Enea Zaffanella. 2024. Towards a Sound Construction of EVM Bytecode Control-Flow Graphs. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '24), September 20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3678721.3686227

FTfJP '24, September 20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1111-4/24/09
https://doi.org/10.1145/3678721.3686227

1 Introduction

Ethereum [17] is historically one of the most popular permissionless blockchain. The key feature of Ethereum is the ability to execute Turing-complete smart contracts through the EVM [2]. Smart contracts are computer programs immutably stored within the blockchain: once deployed, they cannot be modified, so any bug or vulnerability in the contract code can have irrevocable consequences, potentially causing the loss of funds or the execution of unwanted actions. Therefore, it is of utmost importance to make sure that smart contracts are bug-free before their deployment. A common technique to achieve this goal is static analysis, which analyzes code without actually executing it. To identify potential security issues, static analysis often relies on suitable intermediate representations of the program code, such as the widespread CFGs, providing a graphical representation of the control paths that might be traversed during the program's execution.

Ethereum supports different high-level languages for smart contract development (such as Solidity [6] or Vyper [16]), but the EVM runs only code compiled to a low-level language called EVM bytecode. According to [11], when compared to the analysis of high-level source code, the analysis of bytecode provides several advantages, such as the faithfulness of the instruction semantics. Also, being able to analyze bytecode is mandatory when the source code of the smart contracts is not available. However, building CFGs for EVM bytecode is a non-trivial task: contrary to other compiled languages, jump destinations are computed at run-time by using the values on the operand stack, which in general are unknown at compile-time. One thus has to resort to sound approximations of such CFGs, that must contain all possible execution paths that the smart contract can take at run-time to properly identify all the vulnerabilities of interest. While building a sound overapproximation of a CFG is trivial (since jump targets are identified by a specific opcode, one can connect each jump to all possible targets), reducing the number of spurious paths is crucial to ensure that the subsequent semantic analyses are precise enough to achieve the required level of accuracy.

Contribution. This paper presents a novel abstract interpretation-based CFG reconstruction procedure for Ethereum smart contracts.

Our approach targets the EVM bytecode produced by the compilation step and is thus agnostic w.r.t. the source-level language used to develop the contracts. In summary:

- we propose an abstract domain for tracking the operand stacks reaching each opcode;
- we provide an algorithm exploiting the abstract stacks to detect possible jump targets, introducing new edges into the (partial) CFGs;
- we iterate such algorithm up to a fixpoint, building a final CFG that soundly overapproximates the concrete CFG up to some configuration parameters.

Paper structure. Sect. 2 provides an overview of the EVM bytecode and discusses the problem of resolving the target of jumps. Sect. 3 reports our approach for reconstructing CFGs resolving the targets of jump instructions in EVM bytecode. In Sect. 4, we present EVMLiSA, an abstract interpretation-based static analyzer for EVM bytecode where we have implemented our solution, and reports its comparison with a state-of-the-art analyzer for EVM bytecode. Sect. 5 discusses recent related works. Sect. 6 concludes.

2 EVM Bytecode

EVM bytecode is a Turing-complete, stack-based, low-level language consisting of ~150 instructions called *opcodes*.[1] These are interpreted by the EVM to manipulate a stack whose items are 256-bit words. Each instruction is encoded as a hexadecimal number, starting with `0x`. Let us consider a simple fragment of EVM bytecode: `60 01 60 02 01`. The byte `60` corresponds to the `PUSH1` opcode, which pushes one byte onto the stack. The pushed byte is the one following the opcode, i.e., `01`. Similarly, the bytes `60 02` correspond to the EVM instruction `PUSH1 0x02`. The last byte, i.e., `01`, corresponds to the `ADD` opcode, whose semantics pops two items from the stack, sums them, and pushes the result onto the stack. Thus, the translated human-readable version of the bytecode string previously analyzed is

```
PUSH1 0x01 PUSH1 0x02 ADD
```

and, after its execution, the item at the top of the stack is the 256-bit value 3.

Altering the flow of execution. The execution flow of a contract written in EVM bytecode starts with the first opcode and proceeds sequentially. The only EVM opcodes that can alter the flow of execution of a smart contract, without halting,[2] are `JUMP` and `JUMPI`. The `JUMP` instruction consists of an unconditional jump to a specific location of the program, which is the one at the address stored in the topmost item of the stack (which is popped off). For instance, let us consider the following fragment:

```
PUSH1 0x10 PUSH1 0x20 JUMP
```

When the `JUMP` instruction is met, it finds the value `0x20` at the top of the stack. Thus, the value `0x20` is popped from the stack, the program counter is set to `0x20`, and the execution proceeds from the instruction at that address.

[1]The full list of EVM opcodes is available at [17].

[2]Execution can halt: (a) implicitly, when the program counter goes beyond the last opcode of the program; or (b) explicitly, when processing opcodes `STOP`, `RETURN`, `REVERT`, `SELFDESTRUCT`, `INVALID`; or (c) exceptionally, when facing illegal conditions (e.g., stack underflow).

Similarly, the `JUMPI` instruction consists of a conditional jump; the execution will jump to the address stored on the topmost item of the stack only if the item below it (i.e., the second topmost item) is non-zero; otherwise, the execution proceeds with the next opcode. In both cases, the two topmost items are popped off the stack.

Note that the target location of a jump instruction must correspond to a `JUMPDEST` opcode, otherwise, the execution will halt in exceptional mode. The `JUMPDEST` instruction does not alter the stack; its only purpose is to flag those locations of the program to which a (conditional or unconditional) jump is allowed.

2.1 Orphan Jumps

As shown above, the locations to which `JUMP` and `JUMPI` opcodes jump are not hardcoded as data in the instruction syntax (as, e.g., for the value pushed onto the stack by the `PUSH1` opcode); rather, the location is dynamically computed by inspecting the items on the stack. Nonetheless, there are cases where it is easy to statically predict the destination of a jump instruction without actually executing the smart contract. For instance, in the two fragments analyzed previously, the destinations of the `JUMP` and `JUMPI` instructions are easily deduced from the source code, because the two opcodes are syntactically preceded by a `PUSH` instruction (in both cases `PUSH1 0x20`). Borrowing the terminology from [14], we call these instructions *pushed jumps*.

Pushed jumps pose no problem for the construction of the CFG since jump targets can be syntactically resolved. A more challenging class of jumps to resolve is the one of the so-called *orphan jumps* [14]. A simple yet expressive example of an orphan jump is reported below:

```
PUSH1 0x0A PUSH1 0x0C ADD JUMP
```

In this case, the target of the `JUMP` instruction cannot be immediately determined from a syntactic inspection of the source code; to properly resolve the jump and build a precise CFG we need some form of program analysis that can deduce the possible contents of the stack at run-time. In the next section, we present an abstract interpretation-based solution for resolving orphan jump targets.

3 Construction of EVM Bytecode CFGs

For resolving jump destinations, our analysis relies on an abstract domain of *h-sized stacks* whose elements are k-sized sets of integers, with $h, k > 0$.

Definition 3.1 (Abstract domain of k-sets of integers $\mathbb{Z}^\sharp_k$).

$$\mathbb{Z}^\sharp_k \triangleq \langle \wp_{\leq k}(\mathbb{Z}) \cup \{\top_{\mathbb{Z}}, \top_{\overline{\mathbb{Z}}}, \top_{\mathbb{Z}^\sharp_k}\}, \sqsubseteq_{\mathbb{Z}^\sharp_k}, \sqcup_{\mathbb{Z}^\sharp_k}, \sqcap_{\mathbb{Z}^\sharp_k}, \top_{\mathbb{Z}^\sharp_k}, \varnothing \rangle,$$

where elements in $\wp_{\leq k}(\mathbb{Z})$ are sets of integers having cardinality *at most* k. There are three special elements: $\top_{\mathbb{Z}}$, denoting an unknown set of integers that may correspond to valid jump destinations; $\top_{\overline{\mathbb{Z}}}$, denoting an unknown set of integers that do not correspond to valid jump destinations (see Ex. 3.2); and $\top_{\mathbb{Z}^\sharp_k}$, denoting an unknown set of integers. The motivation behind the choice of differentiating between $\top_{\mathbb{Z}}$ and $\top_{\overline{\mathbb{Z}}}$ is explained in the following example.

Example 3.2. Unusual and tricky sequences of opcodes may arise, being EVM bytecode a low-level language and generated by high-level languages. For example, let us consider the following fragment:

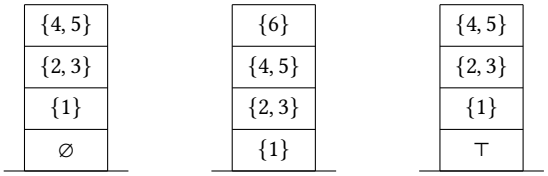


Figure 1: Examples of abstract stacks, elements of $\mathrm{St}^{\sharp}_{2,4}$.

`TIMESTAMP JUMP`. The first opcode pushes the current block's timestamp onto the stack. Then, the `JUMP` opcode takes the value on top of the stack and attempts to jump to that position in the code. Although it is a valid operation, in a real-world scenario it is unlikely that the value inserted by `TIMESTAMP` would be used as a jump destination. Thus, the semantics of `TIMESTAMP` returns $\top_{\overline{\mathbb{Z}}}$. Similarly, we model the semantics of several other opcodes to return $\top_{\overline{\mathbb{Z}}}$.[3]

The partial order $\sqsubseteq_{\mathbb{Z}^{\sharp}_k}$ is the subset inclusion over elements of $\wp_{\leq k}(\mathbb{Z})$ (so that $\varnothing$ is the bottom element and $\top_{\mathbb{Z}}$ is the top element of the sub-lattice), enriched with $\varnothing \sqsubseteq_{\mathbb{Z}^{\sharp}_k} \top_{\overline{\mathbb{Z}}}$, $\top_{\overline{\mathbb{Z}}} \sqsubseteq_{\mathbb{Z}^{\sharp}_k} \top_{\mathbb{Z}^{\sharp}_k}$, and $\top_{\mathbb{Z}} \sqsubseteq_{\mathbb{Z}^{\sharp}_k} \top_{\mathbb{Z}^{\sharp}_k}$. The least upper bound and greatest lower bound operators $\sqcup_{\mathbb{Z}^{\sharp}_k}, \sqcap_{\mathbb{Z}^{\sharp}_k} : \mathbb{Z}^{\sharp}_k \times \mathbb{Z}^{\sharp}_k \to \mathbb{Z}^{\sharp}_k$ are the ones induced by $\sqsubseteq_{\mathbb{Z}^{\sharp}_k}$.

Then, we define the domain of h-sized abstract stacks, approximating concrete stacks with their top h elements. In particular, abstract elements belong to the following set

$$\mathcal{S}_{\mathbb{Z}^{\sharp}_k,h} \triangleq \{[s_0, s_1, \ldots, s_{h-1}] \mid \forall i \in [0, h-1] \,.\, s_i \in \mathbb{Z}^{\sharp}_k\},$$

namely the set of stacks having exactly h elements from $\mathbb{Z}^{\sharp}_k$, where the top of the stack is the rightmost element s_{h-1}. Note that concrete stacks having fewer than h elements are modeled by abstract stacks with exactly h elements, filling the missing elements with (a prefix made of) $\varnothing \in \mathbb{Z}^{\sharp}_k$. For instance, Fig. 1a depicts an abstract stack of $\mathcal{S}_{\mathbb{Z}^{\sharp}_2,4}$ with size 3.

Definition 3.3 (h-sized stack abstract domain).

$$\mathrm{St}^{\sharp}_{k,h} \triangleq \langle \mathcal{S}_{\mathbb{Z}^{\sharp}_k,h} \cup \{\bot_{\mathrm{St}^{\sharp}_{k,h}}\}, \sqcup_{\mathrm{St}^{\sharp}_{k,h}}, \sqcap_{\mathrm{St}^{\sharp}_{k,h}}, \top_{\mathrm{St}^{\sharp}_{k,h}}, \bot_{\mathrm{St}^{\sharp}_{k,h}} \rangle,$$

where $\bot_{\mathrm{St}^{\sharp}_{k,h}}$ is a special bottom element, describing an invalid stack. It differs from the h-sized stack $[\varnothing, \ldots, \varnothing]$, which describes an empty stack.[4] The top element is the h-sized stack whose elements are all $\top_{\mathbb{Z}^{\sharp}_k}$, i.e., $\top_{\mathrm{St}^{\sharp}_{k,h}} = [\top_{\mathbb{Z}^{\sharp}_k}, \ldots, \top_{\mathbb{Z}^{\sharp}_k}]$. Lattice operators are element-wise applications of the ones over $\mathbb{Z}^{\sharp}_k$, with a special case for handling $\bot_{\mathrm{St}^{\sharp}_{k,h}}$.

The abstract function $\mathsf{push} : \mathrm{St}^{\sharp}_{k,h} \times \mathbb{Z}^{\sharp}_k \to \mathrm{St}^{\sharp}_{k,h}$ pushes a k-sized set onto an abstract stack, taking into account that it can only keep track of the top h elements. If the top (resp. bottom) element is taken as input, then the top (resp. bottom) element is returned. Otherwise, letting $\mathbb{st} = [s_0, s_1, \ldots, s_{h-1}] \in \mathrm{St}^{\sharp}_{k,h}$ and $s \in \mathbb{Z}^{\sharp}_k$, we have

$$\mathsf{push}(\mathbb{st}, s) \triangleq [s_1, s_2, \ldots, s_{h-1}, s].$$

Namely, when pushing element s onto $\mathbb{st}$, we *shift down* (i.e., left) all the elements of the stack, removing the bottom (i.e., left-most) element s_0 and adding the new element s in the top (i.e., right-most) position. For instance, in Fig. 1b we show the result of abstractly executing the EVM bytecode `PUSH1 0x06` when starting from the abstract stack of Fig. 1a. It should be noted that all the concrete stacks approximated by the abstract stack in Fig. 1a are known to have stack size at most 3, since the element at depth 4 is $\varnothing$; in contrast, the abstract stack of Fig. 1b describes a set of concrete stacks having an arbitrary size.[5]

Similarly, the abstract function $\mathsf{pop} : \mathrm{St}^{\sharp}_{k,h} \to \mathrm{St}^{\sharp}_{k,h}$ pops an element from an abstract stack. If the top element is taken as input, then top is propagated. If the bottom or the empty stack elements are taken as input, bottom is returned. Otherwise, letting $\mathbb{st} = [s_0, s_1, \ldots, s_{h-2}, s_{h-1}] \in \mathrm{St}^{\sharp}_{k,h}$ we have:

$$\mathsf{pop}(\mathbb{st}) \triangleq \begin{cases} [\varnothing, s_0, s_1, s_2, \ldots, s_{h-2}] & \text{if } s_0 = \varnothing; \\ [\top_{\mathbb{Z}^{\sharp}_k}, s_0, s_1, s_2, \ldots, s_{h-2}] & \text{otherwise.} \end{cases}$$

Intuitively, when popping the topmost element s_{h-1} from stack $\mathbb{st}$, we have two cases: we *shift up* (i.e., right) all the other elements of the stack and fill the bottom (i.e., leftmost) position with either $\varnothing$ (if $s_0 = \varnothing$, so that the stack was known to have size less than h), or $\top_{\mathbb{Z}^{\sharp}_k}$ (if $s_0 \neq \varnothing$, so that the size of the stack was unbounded). As an example, the abstract stack in Fig. 1c is obtained by popping an element from the abstract stack of Fig. 1b.

3.1 Jump Resolution

In the previous section, we defined a parametric abstract stack domain and we want to use it to resolve orphan jumps. To show how the proposed jump resolution algorithm works, let us consider as running example the bytecode fragment shown in Fig. 2a, with an orphan jump (`JUMPI`).

Note that each CFG node is identified by the corresponding program counter *pc* (i.e., its position in the list of opcodes); we will write $stmt(pc)$ to denote the opcode at node *pc*; provided no ambiguity can arise, when referring to figures we will informally denote a CFG edge $pc_1 \to pc_2$ as $stmt(pc_1) \to stmt(pc_2)$.

The pseudocode implementing the construction of the CFG of an EVM bytecode program is reported in Pseudocode 1, with the procedure BUILDCFG (lines 1–8) as entry point. The function takes as input an EVM bytecode program; it starts by building a *partial CFG*, i.e., a control-flow graph with no jump destination resolved (line 2). Considering the running example reported in Fig. 2a, the CFG obtained by this operation is the one reported in Fig. 2c. Note that just the `false` branch of the orphan `JUMPI` has been resolved in this phase (i.e., the red edge `JUMPI` $\xrightarrow{\text{ff}}$ `INVALID`), since it corresponds

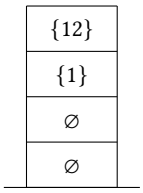
[3] The full list of opcode that pushes $\top_{\overline{\mathbb{Z}}}$ is: `ORIGIN`, `CALLER`, `CALLVALUE`, `CALLDATASIZE`, `CODESIZE`, `GASPRICE`, `RETURNDATASIZE`, `COINBASE`, `TIMESTAMP`, `NUMBER`, `DIFFICULTY`, `GASLIMIT`, `CHAINID`, `SELFBALANCE`, `GAS`, `MSIZE`, `BASEFEE`, `SHA3`, `BALANCE`, `CALLDATALOAD`, `EXTCODESIZE`, `EXTCODEHASH`, `BLOCKHASH`, `CREATE`, `CREATE2`, `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`.

[4] For space reasons, we omit the details regarding the normalization of abstract stacks; intuitively, any non-bottom element s_i followed by $s_j = \varnothing$, where $j > i$, can be replaced by $s'_i = \varnothing$.

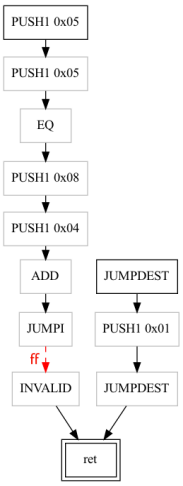
[5] The maximum size of any concrete EVM stack is 1024 and the program execution halts exceptionally if the stack grows beyond this limit; we obviously consider here the case $h < 1024$.

```
PUSH1 0x05
PUSH1 0x05
EQ
PUSH1 0x08
PUSH1 0x04
ADD
JUMPI // orphan jump
INVALID
JUMPDEST
PUSH1 0x01
JUMPDEST
```

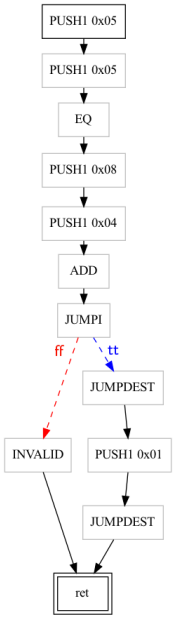
(a)



(b)



(c)



(d)

Figure 2: (a) Running example, (b) JUMPI's input abstract stack, (c) starting CFG, (d) final CFG.

to the opcode syntactically occurring after the JUMPI instruction in the source code.

To resolve the target destination of the orphan jumps, we rely on the analysis of the obtained partial CFG, using the abstract stack domain of Def. 3.3. First, we set the two parameters needed for the analysis, h corresponding to the stack height and k the maximum size of its set elements (line 3), then we rely on the procedure JUMPRESOLVER (lines 10-32), that runs the analysis based on abstract stacks on the input (and potentially partial) CFG (line 12); in our running example we consider $h = 4$ and $k = 2$. This operation computes the entry and exit invariants for each node of the CFG, i.e., the abstract stack that the node takes as input and the resulting stack after applying the abstract semantics of the opcode on the input abstract stack.

Lines 14–30 inspect the analysis results of each jump node pc_1 (JUMP and JUMPI nodes). Following the running example, let us focus on the latter case. Lines 16–17 inspect the element at the top of the abstract stack incoming to the jump node pc_1, ignoring it if

Pseudocode 1 (Jump solver algorithm.)

```
1:  function BUILDCFG(𝒫)
2:      𝒢 ← PARTIALCFG(𝒫)
3:      let h, k ∈ ℕ ∖ {0};
4:      do
5:          changed ← JUMPSOLVER(𝒢, h, k)
6:      while changed;
7:      return 𝒢;
8:  end function
9:
10: function JUMPSOLVER(𝒢, h, k)
11:     let 𝒢 = ⟨N, E⟩;
12:     𝒜 ← runAnalysis(𝒢, h, k);
13:     changed ← false;
14:     for all (𝕊𝕋^in, pc_1, 𝕊𝕋^out) ∈ 𝒜 do
15:         if stmt(pc_1) ∈ {JUMP, JUMPI} then
16:             s ← top(𝕊𝕋^in);
17:             if s ∉ {⊤ℤ, ⊤ℤ̄, ⊤ℤ#_k} then
18:                 for all pc_2 ∈ s do
19:                     if stmt(pc_2) = JUMPDEST then
20:                         if stmt(pc_1) = JUMP then
21:                             E ← E ∪ {pc_1 → pc_2};          ▷ JUMP case
22:                         else
23:                             E ← E ∪ {pc_1 -tt-> pc_2};      ▷ JUMPI case
24:                         end if
25:                         changed ← true;
26:                     end if
27:                 end for
28:             end if
29:         end if
30:     end for
31:     return changed;
32: end function
```

it is one of the special ⊤ elements. Then, an edge is added from pc_1 to pc_2 (a true edge, in the case of JUMPI) for each program counter pc_2 contained in the integer set, provided the target of the edge is a valid JUMPDEST opcode. In our running example, the input abstract stack of the orphan JUMPI is the one depicted in Fig. 2b, thus line 23 adds the true edge from the JUMPI node to the node with program counter equal to 12, i.e., the JUMPDEST node, as depicted in Fig. 2d.

JUMPSOLVER yields true if and only if at least one edge is added to the CFG; the main procedure BUILDCFG keeps calling JUMPSOLVER until no edge is added to the final CFG (lines 4–6), after which it stops and returns the CFG (line 7). In our running example, the final CFG is reported in Fig. 2d.

3.2 From Abstract Stacks to Sets of Abstract Stacks

In the previous section, we analyzed EVM bytecode programs using a single abstract stack of size h, containing elements of the $\mathbb{Z}^\sharp_k$ domain. To improve the number of solved jumps, we now lift the previous abstract stack domain to *sets of abstract stacks*. Indeed, when a loop occurs in the source code, the previous analysis uses the least upper bound (lub) operator to merge abstract stacks into a single abstract stack; this may cause a precision loss when the top-most elements of the collected abstract stack are in turn merged

via the lub of the $\mathbb{Z}^\sharp_k$ domain and we happen to exceed k: the result would be $\top_{\mathbb{Z}^\sharp_k}$, losing all information about the integers in the set (and the possible targets of a potential jump statement).

Thus, we define *abstract stack powerset domain* $\text{SetSt}^\#_{k,h,l}$, consisting of sets of abstract stacks with at most l elements (with a special element $\top_{\text{SetSt}^\#_{k,h,l}}$ denoting the top element), whose height is at most h. Since we keep a collection of abstract stacks, we no longer need to compute the lub on them and hence each element of an abstract stack can now be an integer value (i.e., in our setting, an integer set of at most $k = 1$ elements).

Definition 3.4 (l-sized abstract stacks set domain).

$$\langle \wp_{\leq l}(\mathcal{S}_{\mathbb{Z}^\sharp_1,h}) \cup \{\top_{\text{SetSt}^\#_{1,h,l}}\}, \sqcup_{\text{SetSt}^\#_{1,h,l}}, \sqcap_{\text{SetSt}^\#_{1,h,l}}, \top_{\text{SetSt}^\#_{1,h,l}}, \varnothing \rangle$$

Partial order, least upper bound and greatest lower bound operations corresponds to the set inclusion, union and intersection, respectively, and when the size of the result exceeds l, then $\top_{\text{SetSt}^\#_{k,h,l}}$ is returned.

Pseudocode 1, working on single abstract stacks, can be adapted to work on sets of abstract stacks. In particular, line 12 of Pseudocode 1 will return a set of abstract stacks, and lines 14–32 are applied to each element of the set.

4 Experimental Evaluation

We implemented EVMLiSA, an abstract interpretation-based static analyzer for EVM bytecode based on LiSA (**Li**brary for **S**tatic **A**nalysis) [8, 13], that implements the approach described above to generate CFGs from EVM bytecodes. EVMLiSA, along with the dataset used for the experimental evaluation, is available at

https://github.com/lisa-analyzer/evm-lisa.

For experimental evaluation, we used a dataset of existing smart contracts from the main public network of Ethereum. They were obtained by querying the Etherscan APIs [7]. From this list, we extracted those with less than 3000 opcodes (to keep the experiment time reasonable and allow for manual inspection), obtaining a benchmark suite consisting of 1697 smart contracts. Overall, the benchmark suite contains ~3M opcodes, of which ~240K correspond to jump opcodes.

Our experimental evaluation measures the number of resolved jumps, using the following classification; if a jump node is not reached in the CFG by a path from its entry node, we label the jump as *maybe unreachable*. Otherwise, if a jump is reached with a set of possible stacks $S = \{\mathbb{st}_0, \ldots, \mathbb{st}_n\}$, with $\mathbb{st}_i = [s^i_0, \ldots, s^i_{h-1}] \in \text{St}^\#_{k,h}$, $n \in \mathbb{N}$, then we label it as:

- *resolved* if $\forall i \in [0..n] \,.\, s^i_{h-1} \notin \{\top_{\mathbb{Z}}, \top_{\mathbb{Z}^\sharp_k}\}$; that is, all the top values of S are integer values or $\top_{\overline{\mathbb{Z}}}$;
- *unresolved* if $\exists i \in [0..n] \,.\, s^i_{h-1} \in \{\top_{\mathbb{Z}}, \top_{\mathbb{Z}^\sharp_k}\}$; that is, if there is at least one stack reaching the jump with an unknown numerical value that may correspond to a valid jump destination;
- *definitely unreachable* if $S = \varnothing$; that is, no stack reaches the jump node;
- *maybe unresolved* if $S = \top_{\text{SetSt}^\#_{k,h,l}}$; that is, the stack set exceeded the maximal stack size k;

Table 1: Overall classification of jump opcodes.

	% jumps	
classification	all tests	refined tests
resolved	96.73	97.81
maybe unreachable	2.41	0.00
definitely unreachable	0.69	2.17
unresolved	0.16	0.00
maybe unresolved	0.01	0.02

Note that, one can try to reduce the number of *unresolved* jumps by fine-tuning the parameters l and h. We ran EVMLiSA on the benchmark suite of 1697 smart contracts described above with the powerset of abstract stacks domain described in Sect. 3.2, with $h = 128$, and $l = 32$, corresponding to the maximal height of abstract stacks and the maximal size of abstract stack sets, respectively. Experiments have been performed on the HPC architecture [5] of the University of Parma, Italy.

The 2nd column in Tab. 1 shows the results obtained on all the considered benchmarks. It is important to highlight why a jump node that is not reached by a path from the CFG's entry point is marked as *maybe* unreachable: this is because if there exist other jump nodes classified as *unresolved* or *maybe unresolved*, then this "unreachable" jump node could be reached passing through some of those unresolved jumps; thus, it cannot be classified as *definitely* unreachable. However, if a smart contract contains no *unresolved* and *maybe unresolved* jumps, then all of its *maybe unreachable* jumps can be safely labeled as *definitely unreachable*.

When manually inspecting the experimental results, we noticed that the cause for jumps classified as (maybe) unresolved was the presence of the `SLOAD` opcode. This operator pops an element from the stack and uses it as a key to retrieve the value stored in the permanent memory of the blockchain, which is intrinsically statically unknown; hence, the abstract semantics models `SLOAD` by popping an element from the stack and pushing the abstract value $\top_{\mathbb{Z}}$. In particular, we noticed that the retrieved value was used as a jump destination of a jump node, causing the jump to be labeled as unresolved. It is important to note that this may happen due to the over-approximation occurring during the static analysis process, i.e., at run-time, the value returned by `SLOAD` is not actually used to resolve a jump destination. While leaving the handling of this specific precision problem as future work, we performed a further experiment by refining the aforementioned benchmark, selecting only the smart contract where the value returned by `SLOAD` does not affect the jumps' destination resolution. The refined benchmark consists of 549 smart contracts, containing ~837K opcodes, of which ~59K are jump opcodes. The results obtained on the reduced benchmark suite (again with $h = 128$ and $l = 32$) are shown in the 3rd column of Tab. 1. Specifically, 12 smart contracts out of 549 contain a jump marked as maybe unresolved. We recall that these jumps are labeled this way because the stack set exceeded the maximal stack set size $l = 32$, and thus the abstract value reaching the jump is $\top_{\text{SetSt}^\#_{1,128,32}}$. By fine-tuning the maximal stack set size to $l = 150$, all these jumps can be marked as resolved.

Soundness Thanks to the abstract interpretation framework, the soundness of the sets of stacks we compute is guaranteed. Note that

our analysis provides a conditional result: for the whole process to be formally sound (i.e., for a formal guarantee that the obtained CFG contains *every* possible jump edge that might be traversed at runtime), it is required that no *unresolved*, *maybe unresolved* and *maybe unreachable* jumps are left.

5 Related Work

Ethereum enjoys a wide range of smart contract verification tools, although many of them only support high-level source analysis rather than EVM bytecode analysis [9].

Regarding the static construction of CFGs for EVM bytecode, the state-of-art mainly involves symbolic execution techniques and sometimes SMT solvers. The main limitation of this approach is the fact that symbolic execution generally does not provide guarantees about soundness (i.e not all execution paths are taken into account during the analysis) and especially the execution of an SMT solver may require a considerable amount of time or in the worst case not terminate (i.e. satisfiability problem is undecidable) leading to not being able to analyze the contract. For instance, Oyente [12], also employed by EthIR [1], computes edges that cannot be statically determined on the fly during a symbolic execution also exploiting the SMT solver Z3 [4] to eliminate provably infeasible traces from consideration. However, as reported by the authors, the analyses in some cases took more than 30 minutes for a single contract or ended without results due to some timeouts. Mythril [3] also employs symbolic execution and SMT solving. Instead, EtherSolve [14] computes CFGs (not necessarily sound) using symbolic execution only, but it empirically shows a high degree of precision.

6 Conclusion

In this paper, we presented a new approach aimed at building a sound CFG from EVM bytecode smart contracts, and we implemented our approach in EVMLiSA. As noted in the experimental evaluation, the main drawback in reaching our goal is due to the `SLOAD` opcode, which operates on permanent storage, that is unknown at compile time. Nevertheless, we achieve a sound CFG on all smart contracts if this opcode does not affect the destinations of jump nodes, by fine-tuning the analysis parameters regarding the maximal stack height and the maximal stack set size. While in our experiments the analysis parameters are fixed and independent of the specific smart contract, future work will explore heuristics to choose the best setting for a given smart contract. This would help us both to increase the precision of the CFG construction and to reduce the fake execution paths, such as those likely leading to the usage of `SLOAD`-retrieved values as jump destinations.

As mentioned in Sect. 5, the most closely related work to ours is Ethersolve, despite it not claiming soundness. Preliminary experiments show that EVMLiSA achieves soundness on more EVM bytecode smart contracts than Ethersolve while also resolving more jump destinations. Nevertheless, future work will provide a detailed comparison between the two tools in terms of resolved jumps.

Finally, this paper takes the first step towards making EVMLiSA a static analyzer for detecting run-time errors on EVM bytecode. By building the CFG, EVMLiSA will provide specific checkers for statically detecting popular smart contract vulnerabilities, such as reentrancy [15] and numerical under/overflows [10].

Acknowledgments

Work partially supported by Bando di Ateneo per la Ricerca 2022, funded by University of Parma, (MUR_DM737_2022_FIL_PROGETTI_-B_ARCERI_COFIN, CUP: D91B21005370003), *"Formal verification of GPLs blockchain smart contracts"*.

References

[1] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11138)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 513–520. https://doi.org/10.1007/978-3-030-01090-4_30

[2] A.M. Antonopoulos and G. Wood. 2018. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly. https://books.google.it/books?id=SedSMQAACAAJ

[3] Consensys. [n. d.]. Mythril. "https://github.com/ConsenSys/mythril" Accessed: 08-02-2023.

[4] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[5] Calcolo Scientifico dell'Università di Parma in collaboration with INFN. 2017. High Performance Computing. https://www.hpc.unipr.it/ Accessed: 15-06-2024.

[6] Ethereum. [n. d.]. Solidity documentation. https://docs.soliditylang.org/en/v0.8.24/ Accessed: 12-02-2024.

[7] Etherscan. 2024. Data Export - Open Source Contract Codes. https://etherscan.io/exportData?type=open-source-contract-codes Accessed: Accessed 10-06-2024.

[8] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static analysis for dummies: experiencing LiSA. In *SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021*, Lisa Nguyen Quang Do and Caterina Urban (Eds.). ACM, 1–6. https://doi.org/10.1145/3460946.3464316

[9] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Ethereum Smart Contract Analysis Tools: A Systematic Review. *IEEE Access* 10 (2022), 57037–57062. https://doi.org/10.1109/ACCESS.2022.3169902

[10] Enmei Lai and Wenjun Luo. 2020. Static Analysis of Integer Overflow of Smart Contracts in Ethereum. In *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy* (Nanjing, China) *(ICCSP 2020)*. Association for Computing Machinery, New York, NY, USA, 110–115. https://doi.org/10.1145/3377644.3377650

[11] Francesco Logozzo and Manuel Fähndrich. 2008. On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In *Compiler Construction*, Laurie Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–212.

[12] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 254–269. https://doi.org/10.1145/2976749.2978309

[13] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. 2023. LiSA: A Generic Framework for Multilanguage Static Analysis. In *Challenges of Software Verification*, Vincenzo Arceri, Agostino Cortesi, Pietro Ferrara, and Martina Olliaro (Eds.). Springer Nature Singapore, Singapore, 19–42. https://doi.org/10.1007/978-981-19-9601-6_2

[14] Michele Pasqua, Andrea Benini, Filippo Contro, Marco Crosara, Mila Dalla Preda, and Mariano Ceccato. 2023. Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode. *J. Syst. Softw.* 200 (2023), 111653. https://doi.org/10.1016/J.JSS.2023.111653

[15] Noama Fatima Samreen and Manar H. Alalfi. 2021. Reentrancy Vulnerability Identification in Ethereum Smart Contracts. *CoRR* abs/2105.02881 (2021). arXiv:2105.02881 https://arxiv.org/abs/2105.02881

[16] Vyper. [n. d.]. Vyper documentation. https://docs.vyperlang.org/en/stable/toctree.html Accessed: 12-02-2024.

[17] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32. https://cryptodeep.ru/doc/paper.pdf Accessed: 12-02-2024.

Received 2024-06-26; accepted 2024-07-24