EVMLiSA: Sound Static Control-Flow Graph Construction for EVM Bytecode

Vincenzo Arceri, Saverio Mattia Merenda, Luca Negrini, Luca Olivieri and Enea Zaffanella

PII:	S2096-7209(25)00111-3
DOI:	https://doi.org/10.1016/j.bcra.2025.100384
Reference:	BCRA 100384
To appear in:	*Blockchain: Research and Applications*
Received date:	21 December 2024
Revised date:	30 May 2025
Accepted date:	29 July 2025

EVMLiSA: Sound Static Control-Flow Graph Construction for EVM Bytecode

Vincenzo Arceri[a,*], Saverio Mattia Merenda[a], Luca Negrini[b], Luca Olivieri[b], Enea Zaffanella[a]

[a] *University of Parma, Parma, Italy*
[b] *University Ca' Foscari of Venice, Venice, 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 to build 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. Furthermore, EVMLiSA includes a checker for reentrancy detection, working on the constructed CFG.

Our experiments show that, by fine-tuning the analysis parameters, EVMLiSA is able to build sound CFGs for all real-world smart contracts in the considered benchmark suite. Moreover, EVMLiSA successfully detects all reentrancy vulnerabilities in EVM bytecode smart contracts, while producing

*Corresponding author
Email addresses: `vincenzo.arceri@unipr.it` (Vincenzo Arceri), `saveriomattia.merenda@studenti.unipr.it` (Saverio Mattia Merenda), `luca.negrini@unive.it` (Luca Negrini), `luca.olivieri@unive.it` (Luca Olivieri), `enea.zaffanella@unipr.it` (Enea Zaffanella)

a small number of false positives.

Keywords: Static analysis, Abstract interpretation, Smart contracts, EVM bytecode, Ethereum, Reentrancy vulnerabilities

1. Introduction

Ethereum [65] 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 Ethereum Virtual Machine (EVM) [6]. Smart contracts are computer programs immutably stored in the blockchain; once deployed, smart contracts cannot be modified: any bug or vulnerability in their 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. To ensure this, a common technique 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 Control Flow Graphs (CFGs) [5] that embed the control paths that might be traversed during the program's execution in their structure.

Ethereum supports different high-level languages for smart contract development (such as Solidity [23] or Vyper [62]), but the EVM runs only code compiled to a low-level language called EVM bytecode. According to [38, 49], 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 over-approximation of a CFG is trivial (since jump targets are identified by a specific opcode, one might 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 present 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 over-approximates the concrete CFG up to some configuration parameters;
- we develop EVMLiSA, an open-source, fully automated, Abstract Interpretation-based static analyzer for EVM bytecode, leveraging the sound CFG construction method described in this paper;
- we evaluate the precision of EVMLiSA in reconstructing CFGs using a benchmark suite of real-world smart contracts retrieved from Etherscan;
- we implement a checker in EVMLiSA to detect reentrancy vulnerabilities, and we compare EVMLiSA to a state-of-the-art static analyzer for EVM bytecode on two real-world smart contract benchmark suites, demonstrating that EVMLiSA detects *all* reentrancy bugs with few false positives.

This paper is a revised and extended version of [7]. The main extensions with respect to [7] are described in the following:

- we formally define the abstract stack set domain and improve the jump resolution algorithm to produce a sound CFG, providing a formal proof of its soundness;
- we refine the experimental evaluation on the same benchmark used in [7], now analyzing all smart contracts of the benchmark, rather than only a limited subset consisting of 500 smart contracts;
- we further extend the evaluation by comparing our jump resolution strategy, implemented in EVMLiSA, with four state-of-the-art tools: EtherSolve [50], Gigahorse [29], Mythril [12], and Vandal [10];

- we also implement a reentrancy checker that leverages the CFG constructed by EVMLiSA and compare it with EtherSolve, Mythril and Vandal, on two popular datasets of smart contracts, SolidiFI and SmartBugs.

Paper structure. Section 2 provides an overview of the EVM bytecode, discusses the problem of resolving the target of jumps, and introduces the necessary mathematical notions. Section 3 introduces the concrete semantics of EVM bytecode. Section 4 describes our approach for reconstructing CFGs by resolving the targets of jump instructions in EVM bytecode. In Section 5, we introduce EVMLiSA, an Abstract Interpretation-based static analyzer for EVM bytecode, in which we implement our CFG reconstruction approach. We evaluate its precision in constructing sound CFGs for real-world smart contracts collected from Etherscan and compare its performance with state-of-the-art tools. This section also presents the reentrancy checker implemented by EVMLiSA and compare its performance in detecting reentrancy bugs with state-of-the-art static analyzers for EVM bytecode on two benchmark suites. Section 6 discusses related work. Section 7 concludes the paper.

2. Preliminaries

This section provides an overview of the essential concepts and the terminology necessary for understanding the contents of this paper.

2.1. EVM Bytecode and Jump Instructions

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 1024-sized 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 a 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:

[1]The full list of EVM opcodes is available at [65].

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

and, after its execution, the item at the top of the stack is the 256-bit value `3` (see Figure 1).

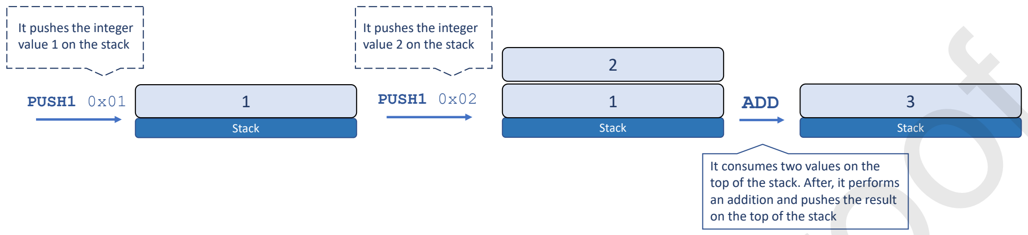

It pushes the integer value 1 on the stack
PUSH1 0x01
1
Stack
It pushes the integer value 2 on the stack
PUSH1 0x02
2
1
Stack
ADD
3
Stack
It consumes two values on the top of the stack. After, it performs an addition and pushes the result on the top of the stack

Figure 1: Execution of a EVM bytecode that computes the addition of two integers.

2.1.1. 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 the location at the address stored in the topmost item of the stack (which is popped off). For instance, let us consider the following fragment:

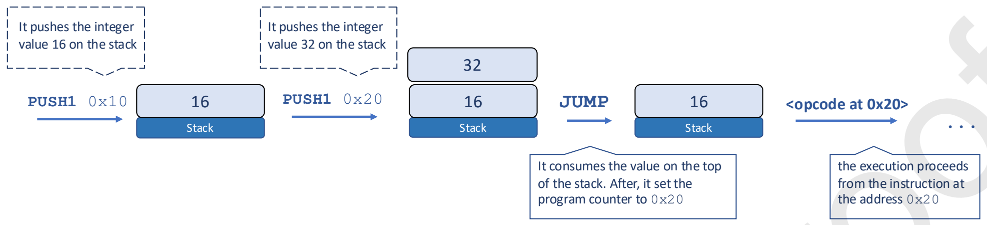
```
PUSH1 0x10 PUSH1 0x20 JUMP
```

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. Figure 2a shows the execution of the instructions until the `JUMP`.

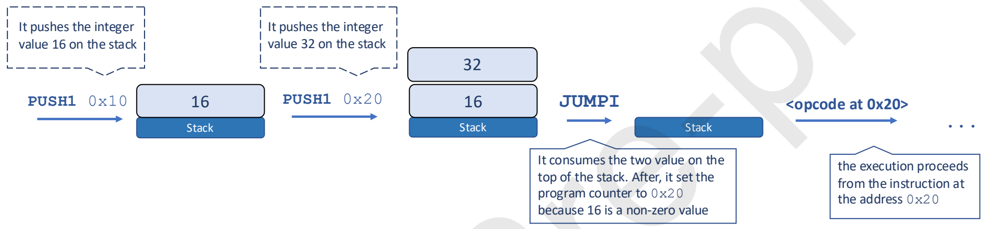
Instead, the `JUMPI` instruction consists of a conditional jump: the execution jumps to the address stored on the topmost item of the stack only if the second topmost item is non-zero; otherwise, the execution proceeds without jumping. In both cases, the two topmost stack elements are popped off the stack. Figure 2b shows the execution of the previous example using `JUMPI` instead of `JUMP`.

[2]Execution can halt: (a) implicitly, when the program runs to completion; (b) explicitly, when processing the halting opcodes `STOP`, `RETURN`, `REVERT`, `SELFDESTRUCT`, `INVALID`; or (c) exceptionally, when facing illegal conditions (e.g., stack underflow).

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


It pushes the integer value 16 on the stack
PUSH1 0x10
16
Stack
It pushes the integer value 32 on the stack
PUSH1 0x20
32
16
Stack
JUMP
16
Stack
It consumes the value on the top of the stack. After, it set the program counter to 0x20
<opcode at 0x20>
. . .
the execution proceeds from the instruction at the address 0x20

(a) JUMP


It pushes the integer value 16 on the stack
PUSH1 0x10
16
Stack
It pushes the integer value 32 on the stack
PUSH1 0x20
32
16
Stack
JUMPI
Stack
It consumes the two value on the top of the stack. After, it set the program counter to 0x20 because 16 is a non-zero value
<opcode at 0x20>
. . .
the execution proceeds from the instruction at the address 0x20

(b) JUMPI

Figure 2: Executions of EVM bytecodes that perform jump operations.

2.1.2. 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 [50], 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* [50]. A simple yet expressive example of an orphan jump is:

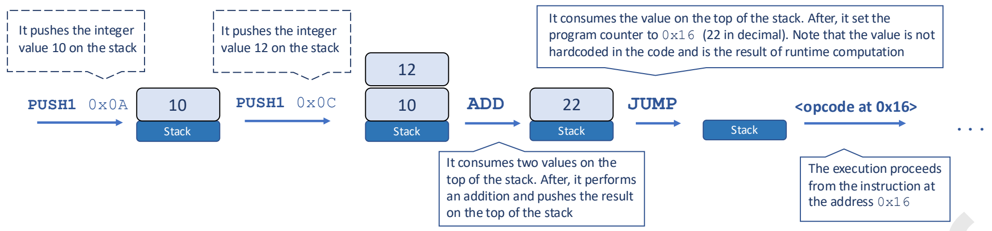

It pushes the integer value 10 on the stack
It pushes the integer value 12 on the stack
It consumes the value on the top of the stack. After, it set the program counter to 0x16 (22 in decimal). Note that the value is not hardcoded in the code and is the result of runtime computation
PUSH1 0x0A
10
Stack
PUSH1 0x0C
12
10
Stack
ADD
22
Stack
JUMP
Stack
<opcode at 0x16>
...
It consumes two values on the top of the stack. After, it performs an addition and pushes the result on the top of the stack
The execution proceeds from the instruction at the address 0x16

Figure 3: Execution of a EVM bytecode that performs an orphan jump.

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

In this case, as reported in Figure 3, the target of the `JUMP` instruction cannot be immediately determined from a syntactic inspection of the source code. To properly resolve the jump, one has to compute the result of the `ADD` opcode. Thus, to build a precise CFG, a program analysis that can deduce the possible contents of the stack at run-time is needed, taking into account each instruction's semantics.

2.2. *Order Theory and Abstract Interpretation*

We briefly recall some basic concepts of order theory and Abstract Interpretation, also introducing the required notation: interested readers are referred to [41, Section 2].

A *poset* $\langle X, \sqsubseteq \rangle$ is a set X equipped with a partial ordering relation $\sqsubseteq \subseteq X \times X$. A (bounded) *lattice* $\langle X, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ is a poset having a minimum element (bottom, $\bot \in X$), a maximum element (top, $\top \in X$) and closed under finitary applications of the least upper bound (lub, $\sqcup$) and the greatest lower bound (glb, $\sqcap$) operators; namely, for all $x_0, x_1 \in X$, both $x_0 \sqcup x_1 \in X$ and $x_0 \sqcap x_1 \in X$. A *complete* lattice is closed under arbitrary lub and glb, so that $\bigsqcup Y \in X$ and $\bigsqcap Y \in X$, for all $Y \subseteq X$. Given a set S, the powerset lattice $\langle \wp(S), \subseteq, \cup, \cap, \emptyset, S \rangle$ is complete.

Given a poset $\langle X, \sqsubseteq \rangle$, a chain $C \subseteq X$ is a totally ordered subset of X; a poset is *complete* (cpo) if all his chains C have a least upper bound $\bigsqcup C \in X$; all complete lattices are cpo. We will often denote a chain as a (possibly infinite) sequence of elements $x_0, \dots, x_i, x_{i+1}, \dots$, where $i \leq j \implies x_i \sqsubseteq x_j$; a poset satisfies the *ascending chain condition* (ACC) if for all its ascending chains $C = x_0, \dots, x_i, x_{i+1}, \dots$ there exists $k \in \mathbb{N}$ such that $\forall j \geq k \,.\, x_j = x_k$. Equivalently, we say that $\langle X, \sqsubseteq \rangle$ is ACC; note that an ACC poset is a cpo.

Given a poset $\langle X, \sqsubseteq \rangle$, a function $f \colon \langle X, \sqsubseteq \rangle \to \langle X, \sqsubseteq \rangle$ is *monotone* if $\forall x, y \in X : x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$; if $\langle X, \sqsubseteq \rangle$ is a cpo, then f is *continuous* if $f(\bigsqcup C) = \bigsqcup_{x_i \in C} f(x_i)$ for all chains C. The n-th iterate f^n of $f \colon \langle X, \sqsubseteq \rangle \rightarrow \langle X, \sqsubseteq \rangle$ starting at $x_0 \in X$, is defined as $f^0 = x_0$ and $f^{n+1} = f(f^n)$; the iterates of a monotone function starting at $\bot$ form an ascending chain. An element $x \in X$ such that $x = f(x)$ is a fixpoint of f; it is the least fixpoint if $\forall y \in X : y = f(y) \implies x \sqsubseteq y$.

Abstract Interpretation [15, 14] is a theoretical framework for sound reasoning about semantic properties of a program, establishing a correspondence between the concrete semantics of a program and an approximation of it, called abstract semantics. The first formulation of the Abstract Interpretation framework relied on *Galois Connection* (GC): let $\langle C, \leq \rangle$ and $\langle A, \sqsubseteq \rangle$ be posets, a pair of monotone functions $\alpha : \langle C, \leq \rangle \rightarrow \langle A, \sqsubseteq \rangle$ and $\gamma : \langle A, \sqsubseteq \rangle \rightarrow \langle C, \leq \rangle$ forms a GC if $\forall x \in C, \forall y \in A : \alpha(x) \sqsubseteq y \Leftrightarrow x \leq \gamma(y)$. A GC requires the existence of a best abstraction, which can be too restrictive. Abstract Interpretation has been thus formalized also under more relaxed hypotheses: in fact, a *monotone* function $\gamma : \langle A, \sqsubseteq \rangle \rightarrow \langle C, \leq \rangle$ is enough to establish a correctness (i.e., soundness) relation, without the need for a GC.

The concrete semantics of a program is classically defined as the least fixpoint lfp$f$ of a continuous function $f \colon C \rightarrow C$ on a complete lattice $\langle C, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ (the concrete computation domain). By Kleene's theorem, this can be computed as the limit of its iterates: lfp$f = \bigsqcup_{n \in \mathbb{N}} f^n$. In general, such a limit is not *finitely* computable. Hence, a corresponding monotone abstract semantic function $g \colon A \rightarrow A$ that correctly over-approximates $f \colon C \rightarrow C$ can be defined on the abstract computation domain $\langle A, \sqsubseteq^\sharp \rangle$ (having a bottom element $\bot^\sharp \in A$): namely, g is *sound* w.r.t. f if $\forall a \in A : f(\gamma(a)) \sqsubseteq \gamma(g(a))$. The abstract iterates of g starting from $\bot^\sharp$ form an increasing chain whose elements correctly over-approximate the corresponding concrete iterates, i.e., $f^n \sqsubseteq \gamma(g^n)$ for all $n \in \mathbb{N}$. The abstract domain is not required to be a complete lattice: if it is an ACC poset, then the abstract increasing chain will converge in a finite number $k \in \mathbb{N}$ of steps to an abstract element $g^k \in A$ that correctly over-approximates the concrete least fixpoint.

3. EVM Language and Semantics

We now formally define what an EVM program is, together with its concrete semantics. An EVM program is simply a sequence of opcodes `op` $\in \mathbb{O}$. When deployed on the blockchain, each opcode is stored at an address ℓ, that can be used to target the opcode through jumps, effectively working as a program counter (two opcodes cannot share the same ℓ). We denote the set of all possible addresses as $\mathbb{L} \subseteq \mathbb{N} \cup \{\epsilon\}$, with ϵ being the empty label, i.e., a label pointing to no opcode that will be used to signal termination of the execution. Given an address $\ell \in \mathbb{N}$, we denote with $\Pi(\ell)$ the opcode corresponding to that address. We denote $\mathtt{next}(\ell)$ the address of the opcode naturally following opcode $\Pi(\ell)$ in the program text. When ℓ is the final opcode in the program, we define $\mathtt{next}(\ell) = \epsilon$.

Following the documentation [65], each opcode pops $\delta \in \mathbb{N}$ existing elements from the stack, pushes $\rho \in \mathbb{N}$ new elements on the stack, and may have parameters (e.g., `PUSH1` is followed by the byte to push on the stack). Thus, we denote an opcode `op` happening at an address ℓ as ${}^{\ell}\mathtt{op}_{\delta}^{\rho}$, embedding the parameters in the opcode definition.

An operand stack $s = [z_0, \dots, z_{n-1}]$ is a LIFO list of length $|s| = n + 1 \leq 1024$ (with $|[]| = 0$), containing elements $z_i \in \mathbb{Z}$ (since the operand stack's elements are 256-bit words, we can refer to them as integer numbers regardless of the type of value they represent). The set $\mathbb{S}$ of all operand stacks also contains the invalid stack $\perp_{\mathbb{S}}$ resulting from erroneous computations.

Program states $\langle s, \ell \rangle \in \mathbb{M}$ are pairs of a stack $s \in \mathbb{S}$ and the address $\ell \in \mathbb{L}$ of the opcode to be executed (i.e., the program counter). The special state $\perp \triangleq \langle \perp_{\mathbb{S}}, \epsilon \rangle \in \mathbb{M}$ denotes an execution error, that corresponds to the EVM halting the execution by raising an exception.

Opcode's semantics is defined through function $[\![\mathtt{op}]\!] \colon \mathbb{M} \to \mathbb{M}$ that given an input program state in $\mathbb{M}$ yields the new state in $\mathbb{M}$ after the execution of `op`. Note that, for any opcode `op`, $[\![{}^{\ell}\mathtt{op}_{\delta}^{\rho}]\!]\perp = \perp$, and $[\![{}^{\ell}\mathtt{op}_{\delta}^{\rho}]\!]\langle \ell', s \rangle = \perp$ if $\ell \neq \ell'$. All non-jumping and non-halting opcodes have similar semantics (where $s = [z_0, \dots, z_{n-1}]$):

$$[\![{}^{\ell}\mathtt{op}_{\delta}^{\rho}]\!]\langle s, \ell \rangle = \begin{cases} \perp, & \text{if } |s| < \delta \text{ or} \\ & |s| - \delta + \rho > 1024; \\ \langle [z_0, \dots, z_{n-1-\delta}, \dot{z}_0, \dots, \dot{z}_{\rho-1}], \mathtt{next}(\ell) \rangle, & \text{otherwise.} \end{cases}$$

Namely, the semantics returns $\perp$ when facing a stack underflow (i.e., there are not enough stack elements to pop) or a stack overflow (i.e., the size of the stack after the execution would be greater than 1024). Otherwise, the top-most δ values are removed from the operand stack and used for the computation of ρ new values $\dot{z}_0, \dots, \dot{z}_{\rho-1}$; after adding these on top of the operand stack, the execution moves to the natural successor of the opcode.

Example 3.1. For instance, consider the opcode `ADD` which pops two values from the stack, sums them, and pushes the result on top of the stack. In this case, $\delta = 2$ and $\rho = 1$. Suppose the stack is $s = [z_0, z_1, z_2]$ and the opcode is at address ℓ. The execution of ADD will yield the new state $\langle [z_0, z_1 + z_2], \mathsf{next}(\ell)\rangle$ since (i) there are enough stack elements to pop (i.e., $|s| = 3 \geq 2$) and the resulting stack's height is within the limits (i.e., $|s| - 2 + 1 = 2 \leq 1024$). Instead, if the stack is $s = [z_0]$, the execution of ADD will yield $\perp$ since there are not enough values to pop from the stack (i.e., $|s| = 1 < 2$).

Since the computation of $\dot{z}_0, \ldots, \dot{z}_{\rho-1}$ is opcode-specific and outside the scope of our work (i.e., it does not modify the control flow), it is omitted from this manuscript. Instead, we define the semantics of JUMP and JUMPI, since they affect the control flow.

Unconditional jump. The semantics of the unconditional jump opcode JUMP is defined as (where $s = [z_0, \ldots, z_{n-1}]$):

$$[\![{}^{\ell}\mathtt{JUMP}_1^0]\!]\langle s, \ell\rangle = \begin{cases} \perp, & \text{if } |s| < 1 \lor \Pi(z_{n-1}) \neq \mathtt{JUMPDEST}; \\ \langle [z_0, \ldots, z_{n-2}], z_{n-1}\rangle, & \text{otherwise.} \end{cases}$$

Namely, the opcode's execution removes one value from the stack and interprets it as the address of the next instruction. If the opcode at that address is a JUMPDEST, the execution proceeds from there; otherwise, $\perp$ is returned as the EVM will raise an exception.

Conditional jump. The semantics of the conditional jump opcode JUMPI is defined as (where $s = [z_0, \ldots, z_{n-1}]$):

$$[\![{}^{\ell}\mathtt{JUMPI}_2^0]\!]\langle s, \ell\rangle = \begin{cases} \perp, & \text{if } |s| < 2 \lor \\ & (z_{n-2} \neq 0 \land \Pi(z_{n-1}) \neq \mathtt{JUMPDEST}); \\ \langle [z_0, \ldots, z_{n-3}], z_{n-1}\rangle, & \text{if } z_{n-2} \neq 0; \\ \langle [z_0, \ldots, z_{n-3}], \mathsf{next}(\ell)\rangle, & \text{otherwise.} \end{cases}$$

Namely, the opcode's execution removes two values from the stack: z_{n-2} determines whether the execution should branch, while z_{n-1} is used as the address of the next instruction if the branch is taken. If the opcode at that address is a JUMPDEST, the execution proceeds from there; otherwise, $\perp$ is returned as the EVM will raise an exception. Note that, JUMPI ignores z_{n-1} entirely if z_{n-2} is equal to 0: in that case, z_{n-1} is discarded (popped from the stack) without any check and does not lead to any exception if it would point to something other than a JUMPDEST.

Program Execution. An EVM program $\mathtt{P} = {}^{\ell_0}\mathtt{op}_{0\,\delta_0}^{\rho_0} \ \ldots \ {}^{\ell_k}\mathtt{op}_{k\,\delta_k}^{\rho_k}$ can be executed through function $\Xi \colon \mathbb{O}^{\star} \times \mathbb{M} \to \mathbb{M}$ that given a program (a sequence of $k \in \mathbb{N}$ opcodes in $\mathbb{O}$) and an input state in $\mathbb{M}$ returns an output state in $\mathbb{M}$. Ξ is defined recursively as:

$$\Xi(\mathtt{P}, \langle s, \ell\rangle) = \begin{cases} \langle [], \epsilon\rangle, & \text{if } \ell = \epsilon \text{ or } \Pi(\ell) \in \{\mathtt{STOP}, \mathtt{SELFDESTRUCT}\}; \\ \langle s, \epsilon\rangle, & \text{if } \Pi(\ell) \in \{\mathtt{RETURN}, \mathtt{REVERT}\}; \\ \Xi(\mathtt{P}, \langle s', \ell'\rangle), & \text{if } [\![\Pi(\ell)]\!]\langle s, \ell\rangle = \langle s', \ell'\rangle \neq \bot; \\ \bot, & \text{otherwise.} \end{cases}$$

Intuitively, the first two cases model program termination, either implicitly (when $\ell = \epsilon$) or explicitly (when executing a halting opcode): execution is stopped by setting the program counter to ϵ and providing the return values if needed; in the third case, we perform a single non-exceptional program step and proceed computing from state $\langle s', \ell'\rangle$; the last case models the program halting due to the computation of an erroneous state. Following the same technique, we also define the partial execution function $\Xi^{\overline{\ell}}$, which executes the program $\mathtt{P}$ until the program counter reaches $\overline{\ell} \neq \epsilon$. This function is defined as:

$$\Xi^{\overline{\ell}}(\mathtt{P}, \langle s, \ell\rangle) = \begin{cases} \langle s, \ell\rangle, & \text{if } \ell = \overline{\ell}; \\ \bot, & \text{if } \ell = \epsilon \text{ or } \Pi(\ell) \in \{\mathtt{STOP}, \mathtt{SELFDESTRUCT}\}; \\ \bot, & \text{if } \Pi(\ell) \in \{\mathtt{RETURN}, \mathtt{REVERT}\}; \\ \Xi^{\overline{\ell}}(\mathtt{P}, \langle s', \ell'\rangle), & \text{if } [\![\Pi(\ell)]\!]\langle s, \ell\rangle = \langle s', \ell'\rangle \neq \bot; \\ \bot, & \text{otherwise.} \end{cases}$$

The objective of $\Xi^{\overline{\ell}}$ is to extract the states that can reach $\overline{\ell}$: hence, it executes the program $\mathtt{P}$ until the execution reaches $\overline{\ell}$, returning the state produced (first case); the remaining cases are handled as before, except that if the execution terminates (no matter if in an erroneous state or with an opcode that halts the program), $\Xi^{\overline{\ell}}$ always returns $\bot$, so as to signal that $\overline{\ell}$ has not been reached.

Example 3.2. For instance, consider the program fragment:

$\mathtt{P} = {}^{0}\mathtt{PUSH1\ 0x01}^{1}_{0} \quad {}^{1}\mathtt{PUSH1\ 0x02}^{1}_{0} \quad {}^{2}\mathtt{ADD}^{1}_{2} \quad {}^{3}\mathtt{JUMP}^{0}_{1}$

The execution of $\Xi(\mathtt{P}, \langle [], 0\rangle)$ and $\Xi^{3}(\mathtt{P}, \langle [], 0\rangle)$ will start in the same way: the first opcode (i.e., the one labelled with 0) is executed, and the stack becomes $[1]$ with the program counter set to $\mathsf{next}(0) = 1$. The second opcode (i.e., the one labelled with 1) is then executed, and the stack becomes $[1, 2]$ with the program counter set to $\mathtt{next}(1) = 2$. The execution proceeds with the third opcode (labelled with 2), with the stack becoming [3] and the program counter becoming $\mathtt{next}(2) = 3$. Here, Ξ and Ξ^3 behave differently: the latter returns $\langle [3], 3\rangle$, since the destination label has been reached, while the former yields $\bot$, because the destination of the jump ($\Pi(3)$) is not a `JUMPDEST`.

4. Construction of EVM Bytecode CFGs

In general, computing (properties of) the semantics of a program is known to be an undecidable problem [54]. Suitable restrictions can yield decidability, such as choosing a programming language that is not Turing complete and/or bounding the resources available (e.g., by providing a finite amount of *gas*). However, even when the semantics is formally decidable, its computation typically incurs an exponentially high, unfeasible computational cost. Thus, following the Abstract Interpretation framework, one can over-approximate the semantic function Ξ so as to make it computable in a finite and often reasonable amount of time. The over-approximation boils down to computing *abstract invariants* by, e.g., solving equations systems [17], executing the program on an inductive abstract interpreter [41], or computing a fixpoint over the program's control flow graph [15]. Regardless of the chosen strategy, computing such invariants requires considering the control flow of the EVM program given as argument of Ξ, that is not statically known.

The main contribution of this paper is an algorithm for constructing *sound* CFG (Definition 1) for EVM programs, thus enabling Abstract Interpretation. A CFG of a program $\mathtt{P}$ is a directed graph $\mathtt{G}_{\mathtt{P}} = (N, E)$ that expresses its control flow. In a CFG, the nodes N are the addresses of all opcodes and the edges $E \subseteq N \times N$ express how the execution flows from one node to another. This means that all the syntactic constructs that form loops, branches, and arbitrary jumps are directly encoded in the CFG structure, simplifying the code to analyze.

Definition 1 (Sound CFG)**.** Given an EVM program $\mathtt{P}$ whose first label is ℓ_0, $\mathtt{G}_{\mathtt{P}} = (N, E)$ is a *sound* CFG of $\mathtt{P}$ if, for all ${}^{\ell}\mathtt{op}_{\delta}^{\rho} \in \mathtt{P}$, we have $\ell \in N$ and

$$\left\{ \ell \to \ell' \;\middle|\; \exists s, s' \in \mathbb{S} \,.\, \Xi^{\ell}(\mathtt{P}, \langle [], \ell_0\rangle) = \langle s, \ell\rangle \wedge [\![{}^{\ell}\mathtt{op}_{\delta}^{\rho}]\!]\langle s, \ell\rangle = \langle s', \ell'\rangle \right\} \subseteq E$$

that is, if every opcode is connected to every possible successor it can have at runtime.

As stated in Section 1, building a sound CFG entails reasoning on the possible values (i.e., stacks) computed by the program. To build sound CFGs, we rely on an iterative algorithm based on the Abstract Interpretation framework, consisting of (i) computing an over-approximation of the possible operand stacks computed by the program, (ii) using the stacks reaching `JUMP` and `JUMPI` opcodes to determine where they might jump, and (iii) repeating the first two steps until the CFG is complete.

4.1. Abstracting computed values

For resolving jump destinations, our analysis relies on an abstract domain of *l-sized sets of h-sized stacks*, with $h, l > 0$, whose elements are an abstraction of the possible integer values computed by a program's opcodes. We define such a domain in a bottom-up fashion, starting from the abstraction of individual elements.

Definition 2 (Stack elements)**.** $\mathbb{Z}^\sharp \triangleq \mathbb{Z} \cup \{\varnothing, \top_{\overline{\mathbb{Z}}}, \top_{\mathbb{Z}^\sharp}\}$ is the set of possible stack elements, where $\varnothing$ is an empty (i.e., not yet initialized) stack element, $\top_{\overline{\mathbb{Z}}}$ is an element that is guaranteed to not be the label of a `JUMPDEST`, and $\top_{\mathbb{Z}^\sharp}$ is an unknown element.

$\mathbb{Z}^\sharp$ thus contains all integer values, together with special symbols to denote stack elements with special values. The introduction of $\top_{\overline{\mathbb{Z}}}$ is motivated by the need to distinguish values that are not valid jump destinations, as shown in the following example.

Example 4.1. Unusual and tricky sequences of opcodes may arise, being EVM bytecode a low-level language generated by high-level languages. For example, let us consider the following fragment: `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 generated by `TIMESTAMP` would be used as a jump destination. Thus, the semantics of `TIMESTAMP` will produce $\top_{\overline{\mathbb{Z}}}$.[3]

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

In the remainder of this paper, we denote concrete stack elements with $z \in \mathbb{Z}$ and abstract stack elements with $v \in \mathbb{Z}^\sharp$. We now define the h-sized abstract stacks, approximating concrete stacks with their top h elements.

Definition 3 (h-sized abstract stacks)**.** $\mathcal{S}_h \triangleq \{[v_0, \ldots, v_{h-1}] \mid \forall i \in [0, h-1] : v_i \in \mathbb{Z}^\sharp\} \cup \{\bot_{\mathcal{S}_h}\}$ is the set of stacks having exactly h elements from $\mathbb{Z}^\sharp$, where the top of the stack is the right-most element v_{h-1}, with a special stack $\bot_{\mathcal{S}_h}$ denoting an invalid stack.

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) $\varnothing \in \mathbb{Z}^\sharp$. For instance, Figure 4 depicts abstract stacks of $\mathcal{S}_4$.

An h-sized abstract stack provides an abstraction of a single stack, that is, of a single execution. To lift our reasoning to all possible executions, we abstract sets of stacks reaching each opcode instead (that is, elements of the concrete powerset lattice $\langle \wp(\mathbb{S}), \subseteq, \cup, \cap, \emptyset, \mathbb{S}\rangle$). Thus, we define the *abstract stack powerset domain* $\mathcal{S}_h^l$, consisting of sets with at most l abstract stacks of height h.

Definition 4 (l-sized abstract stacks set domain)**.** $\langle \mathcal{S}_h^l, \subseteq\rangle$, where $\mathcal{S}_h^l \triangleq \wp_{\leq l}(\mathcal{S}_h)$, is the poset of l-sized sets of h-sized abstract stacks.

Here, $\wp_{\leq l}(\mathcal{S}_h) \subseteq \wp(\mathcal{S}_h)$ is a subset of the powerset of $\mathcal{S}_h$ containing only sets with at most l abstract stacks, together with $\mathcal{S}_h$ itself at the top of the poset. In the remainder of this work, we denote concrete stacks and their sets with $s \in \mathbb{S}$ and $S \in \wp(\mathbb{S})$, respectively, and abstract stacks and their sets with $\hat{s} \in \mathcal{S}_h$ and $\widehat{S} \in \mathcal{S}_h^l$, respectively. Note that $\subseteq$ is still a partial order on $\wp_{\leq l}(\mathcal{S}_h)$. Also note that $\cup$ is no longer an appropriate lub operator, since it can produce sets with more than l elements; we thus define the lub $\sqcup_{\mathcal{S}_h^l}$ as:

$$\widehat{S}_1 \sqcup_{\mathcal{S}_h^l} \widehat{S}_2 \triangleq \begin{cases} \widehat{S}_1 \cup \widehat{S}_2, & \text{if } |\widehat{S}_1 \cup \widehat{S}_2| \leq l; \\ \mathcal{S}_h, & \text{otherwise.} \end{cases}$$

In Appendix A.1, we show that $\langle \mathcal{S}_h^l, \subseteq, \sqcup_{\mathcal{S}_h^l}, \cap, \emptyset, \mathcal{S}_h\rangle$ forms a complete ACC lattice.

Following the Abstract Interpretation framework, we now define means to *concretize* elements of $\langle \mathcal{S}_h^l, \subseteq, \sqcup_{\mathcal{S}_h^l}, \cap, \emptyset, \mathcal{S}_h\rangle$ to elements of $\langle \wp(\mathbb{S}), \subseteq, \cup, \cap, \emptyset, \mathbb{S}\rangle$. We start by defining function $\dot{\gamma}$, parametric on the program `P`, converting a stack element to the possible integers it corresponds to.

Definition 5 (Concretization of stack elements $\dot{\gamma}$)**.** Let $\texttt{JD}_\texttt{P} = \{\ell \in \mathbb{L} \mid \Pi(\ell) = \texttt{JUMPDEST}\}$ be all labels ℓ of program $\texttt{P}$ corresponding to $\texttt{JUMPDEST}$ opcodes. Function $\dot{\gamma} : \mathbb{Z}^\sharp \to \wp(\mathbb{Z})$ is defined as:

$$\dot{\gamma}(v) \triangleq \begin{cases} \emptyset, & \text{if } v = \varnothing; \\ \mathbb{Z} \setminus \texttt{JD}_\texttt{P}, & \text{if } v = \top_{\overline{\mathbb{Z}}}; \\ \mathbb{Z}, & \text{if } v = \top_{\mathbb{Z}^\sharp}; \\ \{v\}, & \text{otherwise.} \end{cases}$$

where concretizes individual stack elements to the concrete integers they abstract.

We then define $\overline{\gamma}$, responsible for transforming an h-sized abstract stack into the set of concrete stacks it over-approximates.

Definition 6 (Concretization of abstract stacks $\overline{\gamma}$)**.** Given two stacks $s_1 = [x_0, \dots, x_j]$ and $s_2 = [y_0, \dots, y_k]$, we denote $s_1 \circ s_2$ their concatenation, i.e., the stack $[x_0, \dots, x_j, y_0, \dots, y_k]$. For $\hat{s} = [v_0, \dots, v_{h-1}]$, let

$$k_\varnothing \triangleq \max\{k \in [0, h] \mid \forall i \in [0, k-1] \,.\, v_i = \varnothing\};$$
$$k_\top \triangleq \max\{k \in [0, h] \mid \forall i \in [0, k-1] \,.\, v_i = \top_{\mathbb{Z}^\sharp}\};$$
$$\overline{\gamma}'(\hat{s}) \triangleq \begin{cases} \{[z_{k_\varnothing}, \dots, z_{h-1}] \mid \forall i \in [k_\varnothing, h-1] \,:\, z_i \in \dot{\gamma}(v_i)\}, & \text{if } k_\varnothing > 0; \\ \{s \circ [z_{k_\top}, \dots, z_{h-1}] \mid \forall i \in [k_\top, h-1] \,:\, z_i \in \dot{\gamma}(v_i), s \in \mathbb{S}\}, & \text{if } k_\varnothing = 0. \end{cases}$$

Then, the concretization function $\overline{\gamma} : \mathcal{S}_h \to \wp(\mathbb{S})$ is defined as

$$\overline{\gamma}(\hat{s}) \triangleq \begin{cases} \bot_{\mathbb{S}}, & \text{if } \hat{s} = \bot_{\mathcal{S}_h}; \\ \overline{\gamma}'(\hat{s}), & \text{otherwise.} \end{cases}$$

Hence, an abstract stack concretizes to all possible concrete stacks whose heads (i.e., the top-most h elements) are compatible with the concretizations of the elements of the abstract stack. Note that, in the definition above, $k_\varnothing > 0$ implies that the abstract stack $\hat{s}$ can only describe concrete stacks having *exactly* $h - k_\varnothing$ elements (hence, if $k_\varnothing = h$, only the empty stack [] is described); otherwise, if $k_\varnothing = 0$, then $\hat{s}$ describes concrete stacks having *at least* $h - k_\top$ elements.

Finally, we define function γ to concretize elements of $\mathcal{S}_h^l$ to sets of concrete stacks.

Definition 7 (Concretization of sets of abstract stacks γ)**.** Function $\gamma : \mathcal{S}^l_h \to \wp(\mathbb{S})$, defined as:

$$\gamma(\widehat{S}) \triangleq \bigcup_{\hat{s} \in \widehat{S}} \overline{\gamma}(\hat{s})$$

concretizes a set of abstract stacks to the concrete stacks it approximates.

Thus, the concretization of a set of abstract stacks corresponds to all the concrete sets obtained by concretizing the abstract stacks one at a time. The monotonicity of γ is proven in Appendix A.2.

Abstract semantics of $\mathcal{S}^l_h$. Since the evolution of the program counter ℓ is embedded in the CFG (i.e., successors are made explicit through CFG edges), $\mathcal{S}^l_h$, as well as any other analyses running on the CFG, just needs to over-approximate the possible stacks in $s \in \mathbb{S}$ computed by each node. We thus define the abstract semantics of $\mathcal{S}^l_h$ through function $[\![\mathsf{op}]\!]^\sharp \colon \mathcal{S}^l_h \to \mathcal{S}^l_h$, that will provide an over-approximation of what $[\![\mathsf{op}]\!] \colon \wp(\mathbb{S}) \to \wp(\mathbb{S})$ computes over $\langle \wp(\mathbb{S}), \subseteq, \cup, \cap, \emptyset, \mathbb{S} \rangle$ (here, $[\![\mathsf{op}]\!]$ is defined, abusing notation, as the additive lift of function $[\![\mathsf{op}]\!]$ from Section 3).

We first define a few auxiliary functions operating on abstract stacks.

Function $\mathsf{top} \colon \mathcal{S}_h \to \mathbb{Z}^\sharp$ returns the top (i.e., rightmost) element of the stack (without removing it); it is defined as

$$\mathsf{top}(\hat{s}) \triangleq \begin{cases} v_{h-1}, & \text{if } \hat{s} = [v_0, \dots, v_{h-1}]; \\ \varnothing, & \text{otherwise } (\hat{s} = \bot_{\mathcal{S}_h}). \end{cases}$$

Function $\mathsf{push} \colon \mathcal{S}_h \times \mathbb{Z}^\sharp \to \mathcal{S}_h$ pushes a new stack element at the top an abstract stack, taking into account the maximal size h. This translates to *shifting down* (i.e., left) all the elements of the stack, removing the bottom (i.e., left-most) element and adding the new element in the top (i.e., right-most) position. Letting $\hat{s} = [v_0, v_1, \dots, v_{h-1}] \in \mathcal{S}_h$ and $v \in \mathbb{Z}^\sharp$, push is defined as:

$$\mathsf{push}(\hat{s}, v) \triangleq [v_1, \dots, v_{h-1}, v].$$

We also define $\mathsf{push}(\bot_{\mathcal{S}_h}, v) \triangleq \bot_{\mathcal{S}_h}$. By extension, for each $n \in \mathbb{N}$, we define $\mathsf{push}^n \colon \mathcal{S}_h \times {\mathbb{Z}^\sharp}^n \to \mathcal{S}_h$ pushing a sequence of n stack elements on top of a stack as:

$$\mathsf{push}^n(\hat{s}, v_1, v_2, \dots, v_n) \triangleq \begin{cases} \hat{s}, & \text{if } n = 0; \\ \mathsf{push}^{n-1}(\mathsf{push}(\hat{s}, v_1), v_2, \dots, v_n), & \text{otherwise.} \end{cases}$$

Function $\mathsf{pop}\colon \mathcal{S}_h \to \mathcal{S}_h$ pops the top (i.e., right-most) element from an abstract stack, while filling the bottom (i.e., left-most) element with either $\top_{\mathbb{Z}^\sharp}$ or $\varnothing$. Specifically, $\varnothing$ is used when it is guaranteed that no other element exist beyond the ones already present in the stack: this only happens when the current bottom element is $\varnothing$ itself. Otherwise, there is no guarantee on the size of the concrete stacks (intuitively, the bottom element of the abstract stack might be the last element as well as an intermediate one), and we use $\top_{\mathbb{Z}^\sharp}$ as the new bottom element. Formally, if $\hat{s} = [v_0, \ldots, v_{h-2}, v_{h-1}] \in \mathcal{S}_h$:

$$\mathsf{pop}(\hat{s}) \triangleq \begin{cases} [\varnothing, v_0, \ldots, v_{h-2}], & \text{if } v_0 = \varnothing; \\ [\top_{\mathbb{Z}^\sharp}, v_0, \ldots, v_{h-2}], & \text{otherwise.} \end{cases}$$

Again, we define $\mathsf{pop}(\bot_{\mathcal{S}_h}) \triangleq \bot_{\mathcal{S}_h}$.

By extension, for each $n \in \mathbb{N}$, we define $\mathsf{pop}^n\colon \mathcal{S}_h \to \mathcal{S}_h$ popping n stack elements from top of a stack as:

$$\mathsf{pop}^n(\hat{s}) \triangleq \begin{cases} \hat{s}, & \text{if } n = 0; \\ \mathsf{pop}^{n-1}(\mathsf{pop}(\hat{s})), & \text{otherwise.} \end{cases}$$

Note that the abstract semantics will never push the abstract element $\varnothing \in \mathbb{Z}^\sharp$ on an abstract stack; hence, when starting from the abstract stack $[\varnothing, \ldots, \varnothing]$ (describing the empty concrete stack) and applying sequences of $\mathsf{push}/\mathsf{pop}$ operations, all $\varnothing$ elements (if any) will always appear at the bottom end of the stack. Hence, we can define function $\mathsf{height}\colon \mathcal{S}_h \to \mathbb{N}$, counting the number of non-$\varnothing$ elements in an abstract stack, as follows

$$\mathsf{height}(\hat{s}) \triangleq \begin{cases} h - k_\varnothing, & \text{if } \hat{s} \neq \bot_{\mathcal{S}_h}; \\ 0, & \text{otherwise,} \end{cases}$$

where for each $\hat{s} = [v_0, \ldots, v_{h-1}] \in \mathcal{S}_h$ the index $k_\varnothing$ is computed as specified in Definition 6.

As an example, in Figure 4b we show the result of abstractly executing the EVM bytecode `PUSH1 0x06` when starting from the abstract stack of Figure 4a. It should be noted that all the concrete stacks approximated by the abstract stack in Figure 4a are known to have exactly 3 elements (because the element at depth 4 is $\varnothing$); in contrast, the abstract stack of Figure 4b describes a set of concrete stacks having at least 4 elements, but possibly

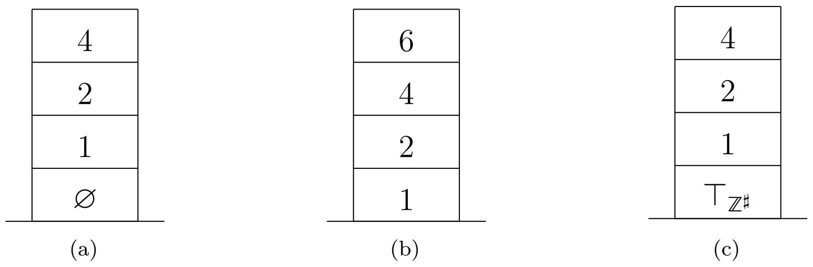


Figure 4: Examples of abstract stacks, elements of $\mathcal{S}_4$.

more.[4] Instead, the abstract stack in Figure 4c, which is obtained by popping an element from the abstract stack of Figure 4b, describes concrete stacks having at least 3 elements. Finally, the height of the three abstract stacks is respectively 3, 4, and 4.

We can now define the abstract semantics $[\![\mathsf{op}]\!]^\sharp$ of $\mathcal{S}_h^l$ as:

$$[\![{}^{\ell}\mathsf{op}_{\delta}^{\rho}]\!]^\sharp \widehat{S} \triangleq \bigcup_{\hat{s} \in \widehat{S}} \{[\![{}^{\ell}\mathsf{op}_{\delta}^{\rho}]\!]^\sharp \hat{s}\}$$

where we define, by abusing notation, $[\![\mathsf{op}]\!]^\sharp \colon \mathcal{S}_h \to \mathcal{S}_h$ as:

$$[\![{}^{\ell}\mathsf{op}_{\delta}^{\rho}]\!]^\sharp \hat{s} \triangleq \begin{cases} \perp_{\mathcal{S}_h}, & \text{if } \hat{s} = \perp_{\mathcal{S}_h} \vee \mathsf{height}(\hat{s}) < \delta; \\ \mathsf{push}^{\rho}(\mathsf{pop}^{\delta}(\hat{s}), \dot{v}_0, \dots, \dot{v}_{\rho-1}), & \text{otherwise.} \end{cases}$$

The abstract semantics thus evolves each individual stack in isolation. The abstract semantics on individual stacks returns $\perp_{\mathcal{S}_h}$ if there are not enough elements to pop, or if $\perp_{\mathcal{S}_h}$ is passed as input stack. Otherwise, it pops δ stack elements and pushes ρ new elements $\dot{v}_0, \dots, \dot{v}_{\rho-1}$ on top of the stack. Similarly to the concrete semantics, the computation of $\dot{v}_0, \dots, \dot{v}_{\rho-1}$ is omitted from this work as it is opcode-specific and does not modify the control flow.[5] Note that we do not need to define specific semantics for `JUMP` and `JUMPI`: the peculiarity of these operands is the modification of the program counter, that will be addressed in Section 4.2. Appendix A.3 reports the soundness proof of $[\![\mathsf{op}]\!]^\sharp$.

[4]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 $h < 1024$.

[5]Intuitively, the computation happens concretely whenever all operands are elements of $\mathbb{Z}$; instead, $\top_{\overline{\mathbb{Z}}}$ and $\top_{\mathbb{Z}^\sharp}$ are propagated in the result as-is.

Pseudocode 1 (Build CFG algorithm.)

```
1: function buildCFG(P, $h$, $l$, $conservative$)
2:     $G_P \leftarrow$ partialCFG(P)
3:     do
4:         $changed \leftarrow$ jumpSolver($G_P, h, l, conservative$)
5:     while $changed$;
6:     return $G_P$;
7: end function
```

4.2. Resolving Jumps

This section introduces the general iterative algorithm to build a sound CFG G_P of an EVM program P by exploiting $\mathcal{S}_h^l$. To illustrate how the algorithm works, let us consider as running example the bytecode fragment shown in Figure 5a, with an orphan JUMPI (each opcode is preceded by the program counter).

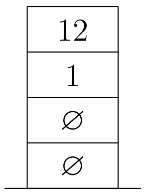
The pseudocode implementing the construction of the CFG of an EVM bytecode program is reported in Pseudocode 1 with function buildCFG. The function takes as input an EVM bytecode program, the parameters h and l for the abstract stacks sets analysis of Definition 4, and the Boolean flag *conservative*, whose meaning will be explained shortly; it starts by building a *partial CFG*, i.e., a control-flow graph with no jump destination resolved (line 2 with function partialCFG). Considering the running example reported in Figure 5a, the CFG obtained by this operation is the one reported in Figure 5c. 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 to the opcode syntactically occurring after the JUMPI instruction in the source code.

buildCFG then proceeds by iteratively adding edges to this CFG by successive invocations of jumpSolver (Pseudocode 2) at line 4, until no more edges are added.

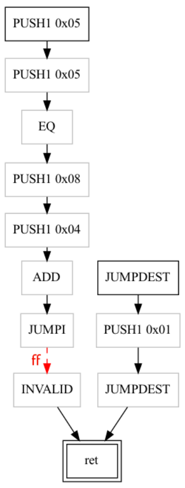
jumpSolver relies on the analysis of abstract stacks sets of Definition 4 (line 2 of Pseudocode 2) to resolve the target destination of the orphan jumps on the input (and potentially partial) CFG. This operation computes the entry and exit invariants for each node of the CFG, i.e., the abstract stacks that the node takes as input and the resulting stacks after applying the abstract semantics of the opcode. Note that nodes that are unreachable (i.e., no path exists leading to them from the first opcode) are associated with a bottom state. In our running example we consider $h = 4$ and $l = 1$.

```
0:   PUSH1 0x05
2:   PUSH1 0x05
4:   EQ
5:   PUSH1 0x08
7:   PUSH1 0x04
9:   ADD
10:  JUMPI // orphan jump
11:  INVALID
12:  JUMPDEST
13:  PUSH1 0x01
15:  JUMPDEST
```

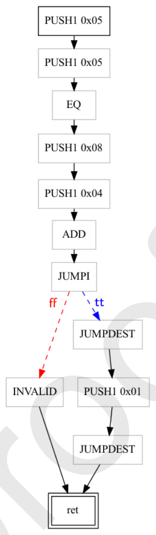
(a)



(b)



(c)



(d)

Figure 5: (a) Running example, (b) `JUMPI`'s input abstract stack, (c) starting CFG, (d) final CFG.

Lines 4–19 inspect the analysis results of each jump node ℓ (`JUMP` and `JUMPI`). Lines 11–16 inspect the element at the top of each abstract stack incoming to the jump node ℓ. If it is a valid jump destination (lines 12–13, where $\mathtt{JD}_{\mathtt{P}} \subseteq \mathbb{L}$ are all labels ℓ of program `P` such that $\Pi(\ell) = \mathtt{JUMPDEST}$), an edge is added from ℓ to v (the address of the jump destination) using function buildEdge(ℓ, v), that returns a `true` edge between ℓ and v if $\Pi(\ell) = \mathtt{JUMPI}$, or a normal edge between the two if $\Pi(\ell) = \mathtt{JUMP}$. In our running example, the input abstract stack of the orphan `JUMPI` is the one depicted in Figure 5b, thus line 13 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 Figure 5d.

jumpSolver handles two more cases: when the input stack $\widehat{S}_{in}$ is $\mathcal{S}_h$ (lines 6–8), and when the top of the stack is a valid but unknown jump destination (i.e., $\top_{\mathbb{Z}^\sharp}$ but not $\top_{\overline{\mathbb{Z}}}$, line 15). In both cases, an edge between the jump and every possible jump destination is added only if the *conservative* parameter is `true`: one might try tuning h and l to obtain soundness on a specific program without defaulting to a conservative analysis that would add spurious edges to achieve soundness, possibly degrading the results of the analyses run on the generated CFG.

jumpSolver yields `true` if and only if at least one edge is added to the

Pseudocode 2 (Jump solver algorithm.)

```
 1: function jumpSolver($\mathtt{G_P} = (N, E), h, l, \mathit{conservative}$)
 2:     $\mathcal{A} \leftarrow$ runAnalysis($\mathtt{G_P}, h, l$);
 3:     $E' \leftarrow E$;
 4:     for all $(\widehat{S}_{in}, \ell, \widehat{S}_{out}) \in \mathcal{A} : \Pi(\ell) \in \{\mathtt{JUMP}, \mathtt{JUMPI}\}$ do
 5:         if $\widehat{S}_{in} = \mathcal{S}_h$ then
 6:             if $\mathit{conservative}$ then
 7:                 $E \leftarrow E \cup \{$buildEdge$(\ell, \ell') \mid \ell' \in \mathtt{JD_P}\}$;
 8:             end if
 9:         else
10:             for all $\hat{s} \in \widehat{S}_{in}$ do
11:                 $v \leftarrow \mathsf{top}(\hat{s})$;
12:                 if $v \in \mathtt{JD_P}$ then
13:                     $E \leftarrow E \cup \{$buildEdge$(\ell, v)\}$;
14:                 else if $v = \top_{\mathbb{Z}^\sharp} \wedge \mathit{conservative}$ then
15:                     $E \leftarrow E \cup \{$buildEdge$(\ell, \ell') \mid \ell' \in \mathtt{JD_P}\}$;
16:                 end if
17:             end for
18:         end if
19:     end for
20:     return $E \neq E'$;
21: end function
```

CFG; the main procedure buildCFG keeps calling jumpSolver until no edge is added to the final CFG (lines 3–5), after which it stops and returns the CFG (line 6). In our running example, the final CFG is reported in Figure 5d.

If a program `P` contains n `JUMP`/`JUMPI` opcodes and m `JUMPDEST` opcodes, then at most $n \times m$ edges can be added by procedure buildEdge (note that procedure partialCFG already adds all false edges of `JUMPI` opcodes): this yields a very conservative upper bound to the number of calls of procedure JumpSolver. An in depth investigation over the worst-case complexity of the analysis is beyond the scope of this paper. In practice, the number of executions of JumpSolver in an analysis is expected to be much lower: first of all, most jumps (e.g., all pushed jumps) resolve to a single `JUMPDEST` opcode; it is also reasonable to assume that a large fraction of the jumps will resolve to at most k `JUMPDEST` opcodes, where k is a small constant; also, a single call to JumpSolver typically adds many edges at once.

4.3. Soundness of the generated CFG

The proof of soundness of buildCFG is given in Appendix A.4. Here, we discuss soundness informally by classifying jump opcodes exploiting the results of $\mathcal{S}_h^l$ on the result of buildCFG. In the following, we assume that a jump is reached with a possibly empty set of abstract stacks $\widehat{S} = \{\hat{s}_0, \ldots, \hat{s}_{n-1}\}, n \in \mathbb{N}$. We say that an abstract stack $\hat{s}_i \in \widehat{S}$ is *erroneous* for jump opcode `op` if either $\mathsf{height}(\hat{s}_i) = 0$,[6] or `op` = `JUMPI` and $\mathsf{height}(\hat{s}_i) = 1$; that is, the stack describes an already erroneous execution or it will immediately cause an error when trying to execute the jump. We classify jump opcodes as:

- *unreachable*, if $\widehat{S} = \varnothing$, i.e., no stack reaches the jump node;
- *erroneous*, if all the stacks $\hat{s}_i \in \widehat{S}$ are erroneous;
- *resolved*, if all non-erroneous $\hat{s}_i \in \widehat{S}$ satisfy $\mathsf{top}(\hat{s}_i) \neq \top_{\mathbb{Z}^\sharp}$; that is, all the top values of $\widehat{S}$ are integer values or $\top_{\overline{\mathbb{Z}}}$;
- *unknown*, otherwise; that is, either $\widehat{S} = \mathcal{S}_h$ or there exists a non-erroneous stack $\hat{s}_i \in \widehat{S}$ such that $\mathsf{top}(\hat{s}_i) = \top_{\mathbb{Z}^\sharp}$ (i.e., the jump destination is an unknown numerical value).

Resolved jumps are thus guaranteed to have all of their outgoing edges correctly added to the CFG, since the sound analysis determined precisely the possible jump destinations they can target. *Unknown* jumps are instead over-approximated with all possible destinations only if the *conservative* analysis is enabled; otherwise, they are (i) unsoundly ignored if $\widehat{S} = \mathcal{S}_h$, (ii) unsoundly under-approximated with the definite jump destinations if at least one reaching stack has $\top_{\mathbb{Z}^\sharp}$ as its top element. Both unsound resolutions are by-design: one can use such jumps as a hint that the parameters h or l are too restrictive, and may try new combinations until these jumps become *resolved*. Finally, *unreachable* jumps are deemed as not reachable by the analysis since no stack (i.e., no execution) ever reaches the jump, and *erroneous* jumps correspond to jumps that will never be executed by the EVM.

Whenever *conservative* = `true`, *unreachable* jumps are definitely unreachable: since they cannot be reached in a sound over-approximation of the control flow of the program, no concrete execution can ever traverse such jumps. Otherwise, unreachability might be due to the under-approximation

[6]Notice that this includes the case $\hat{s}_i = \bot_{\mathcal{S}_h}$.

of *unknown* jumps. Hence, jumps are truly unreachable only if no *unknown* jump exists. The same holds for *erroneous* jumps. Soundness can thus be stated as "either *conservative* = `true`, or no *unknown* jumps are found".

5. Experimental Evaluation

The proposed algorithm BUILDCFG and the powerset of abstract stack domain $\mathcal{S}_h^l$ have been implemented in EVMLiSA, an Abstract Interpretation-based static analyzer for EVM bytecode based on LiSA (**Li**brary for **S**tatic **A**nalysis) [44, 28, 43]. Both EVMLiSA and the dataset used for the experimental evaluation are available at:

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

For evaluating our approach, we used a dataset of existing smart contracts (the same used in [7]) retrieved from the main public network of Ethereum by querying the Etherscan APIs [25], obtaining a list of 5000 open-source verified smart contracts published by Etherscan.[7] From this list, we extracted those with less than 3000 opcodes (to keep the experiment time reasonable and allow for manual inspection) and with at least one `JUMP` or `JUMPI` opcode, obtaining a final benchmark suite of 1697 distinct smart contracts. All experiments in this section were ran on a Macbook Pro with Apple Silicon M3 Pro CPU and 18GB of RAM. Overall, the suite contains ∼3M opcodes, of which ∼240K correspond to jump opcodes. We ran EVMLiSA on such benchmark with the powerset of abstract stacks domain from Definition 4, using the non-conservative analysis (i.e., *conservative* = `false`) and varying the parameters h and l to investigate their impact on the generated CFGs. Specifically, we use values of h of 16, 32, 64, and 128, and values of l between 1 and 10. For each combination, Figure 6 reports the percentage of jumps classified as resolved, the number of generated edges (including those having the form $\ell \to \mathsf{next}(\ell)$), and the average execution time (in milliseconds) for the analysis of a single smart contract. The plots show that increasing h beyond 32 does not lead to improvements in the analysis, nor it degrades the analysis performances: this is likely due to the contained size of the

[7]Smart contracts were downloaded on June 11, 2024. The list of contract addresses is publicly available at `https://github.com/lisa-analyzer/evm-lisa/blob/master/benchmark/5000-benchmark-2024-06-11.txt`.

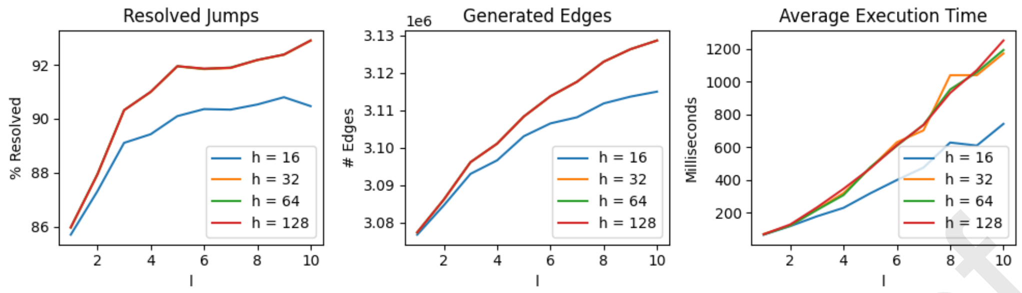


Figure 6: Impact of h and l on the number of resolved jumps, generated edges, and execution time. $h = 32$ and $h = 64$ overlaps with $h = 128$ in the first two charts.

Table 1: Overall classification of jump opcodes (239895 in total).

Classification	**Conservative**	**Non-Conservative**
resolved %	90.53	92.89
unknown %	5.96	0.19
unreachable %	3.44	6.92
erroneous %	0.07	0.00
unsound CFGs	0	243
CFG edges	6116858	3122929
Average time	~1.6s	~1.1s

contracts, that limit the number of stack elements used by each opcode to a small fraction of the available ones. Instead, increasing l generally leads to a higher number of resolved jumps and generated edges, but also requires a higher execution time. This is expected, as increasing l allows the analysis to precisely track more abstract stacks generated in conditional branches and loops before incurring in over-approximation, which in turn leads to a more precise analysis. One may wonder about the local non-monotonicity in the graph showing the percentage of resolved jumps as l increases. This behavior arises because, as l increases, the analysis explores additional paths (as shown by the graph of the generated edges), and some jumps that were previously classified as resolved at lower values of l may become unknown. This is due to the introduction of new paths and abstract stacks that propagate imprecise information (i.e., $\top_{\mathbb{Z}^\sharp}$), ultimately reducing the precision for those particular jumps. However, despite these local fluctuations, the overall trend shows that increasing l generally leads to a higher percentage of resolved jumps.

We now compare the performance of the best non-conservative analysis in terms of resolved jumps (i.e., $h = 32$ and $l = 10$) with the conservative analysis ran with the same parameters, to investigate the difference between the two approaches. Table 1 shows the results obtained on the target benchmark, produced by the **Conservative** analysis (i.e., when $\textit{conservative} = \texttt{true}$), and the **Non-Conservative** one, in terms of jump classifications. For each analysis we report the percentage of jumps in each class (resolved, unknown, unreachable, and erroneous), the total number of edges in the CFG produced by our algorithm, the number of unsound CFGs produced, and the average time in seconds for the analysis of a single smart contract. It is worth noting that, out of 1697 smart contracts analyzed, only 243 of them got a *possibly* unsound CFG (i.e., had at least one unknown jump) with the non-conservative analysis. This hints that even with small values for h and l, our analysis is able to produce sound CFGs with reasonable resource requirements. Furthermore, there is also a visible decrease of resolved jumps, with an increase of unknown ones. This can be imputed to the higher number of edges generated by the conservative mode (almost twice as many): having more edges, more abstract stacks are propagated in the CFG and eventually reach jump opcodes, causing the threshold l to be exceeded more often. This also has the effect of reducing the number of unreachable jumps. Nevertheless, in our opinion, such a small decrease in resolution rate is a reasonable compromise for the soundness guarantee. Another source of unknown jumps is the usage of $\top_{\mathbb{Z}^\sharp}$ as jump destination. As already discussed in [7], in our benchmark, $\top_{\mathbb{Z}^\sharp}$ only appears in abstract stacks as result of the `SLOAD` opcode, that depends from the dynamic blockchain state and is thus inherently statically unknown. To address this problem, we implemented an hybrid approach by integrating a runtime feature: specifically, during the analysis phase, EVMLiSA performs an API request to Etherscan[8] to retrieve the current value stored in the blockchain storage. This feature minimizes the ambiguity of the `SLOAD` opcode by resolving jumps that are unknown due to this operator. While this approach can improve the accuracy of the generated CFG (i.e., it reduces the number of spurious edges generated), it is not fully static and it fails at generating a CFG that is valid for all possible executions. We thus omit it from our evaluation.

5.1. Comparison on CFG reconstruction

[8]`https://etherscan.io/`

In this section, we compare EVMLiSA with state-of-the-art tools for reconstructing CFGs from EVM bytecode. Specifically, we selected tools that meet the following criteria: (i) they adopt a static analysis approach, (ii) they operate on EVM bytecode, and (iii) they dump the reconstructed CFG as output, in some format. Thus, we selected the following state-of-the-art tools to be compared with EVMLiSA:

- EtherSolve [50]: a static analyzer based on symbolic execution that reconstructs CFG from EVM bytecode;
- Gigahorse [29]: a decompiler from EVM bytecode into a high-level three-address code representation;
- Vandal [10]: a static analysis framework for smart contracts decompiling the EVM bytecode to an intermediate representation that includes the code control flow;
- Mythril [12]: a security analysis tool for EVM bytecode based on symbolic execution.

We run EVMLiSA with parameters $h = 64$ (maximum abstract stack height), $l = 20$ (maximum abstract stack set size), but *disabling* the conservative option to ensure a fairer comparison: we recall that enabling the conservative option ensures a sound construction of the CFG connecting all unknown jumps to all `JUMPDEST` opcodes. We ran all tools on the same benchmark set of 1697 smart contracts retrieved from Etherscan, discussed in the previous section. A timeout of 5 minutes was set to limit the analysis time for each contract. For each tool, we measured the number of contracts for which it successfully produced a CFG without timing out or returning an error, the average basic blocks and edges discovered by each tool, and average execution time needed to complete the analysis (on the contract on which the tool did not timed-out or raised execution errors).

Note that, since the tools output CFGs in different formats, we needed to standardize them. For each tool, we converted its output into a common representation, containing a list of basic blocks, each identified by the program counter of the opcode that starts the block, and a list of edges between the blocks. Additionally, we removed the last basic block of each EtherSolve's CFGs, since it is an instrumented block that is not part of the EVM bytecode and that is connected to all other blocks.

Table 2: Quantitative comparison of CFG reconstruction tools over 1697 contracts from Etherscan. Tools labeled with (∗) ignore opcodes.

Tool	**# Contracts**	**Basic blocks (mean)**	**Edges (mean)**	**Time ms (mean)**
EVMLiSA	**1697** (100%)	145.97	**193.79**	2877.47
Vandal (∗)	1685 (99.29%)	**160.76**	190.98	2946.15
EtherSolve (∗)	1678 (98.88%)	60.19	68.37	**223.96**
Mythril	1406 (82.85%)	137.03	166.68	17395.41
Gigahorse	1393 (82.09%)	72.64	61.55	2270.51

Table 2 reports the comparison of CFG reconstruction tools on the benchmark suite. As shown in the second column of the table, EVMLiSA is the only tool to obtain a 100% success rate, being able to successfully analyze and compute a CFG for all the contracts contained in the benchmark. Slightly lower success rates were observed for Vandal and EtherSolve, with 99.29% and 98.88%, respectively. However, upon manual inspection, we found that both EtherSolve and Vandal wrongly handle `TLOAD` and `TSTORE` opcodes, that manipulate transient storage, and the `PUSH0` opcode, introduced on April 12, 2023 in EIP-1153[9] and EIP-3855[10], respectively. Both Vandal and EtherSolve silently skip such instructions, which causes the program counter tracking to become inaccurate. We computed that 1391 contracts out of 1697 contain occurrences of these opcodes: the CFGs these tools reconstruct may not reflect the actual control flow, and their reported success rates should be interpreted with caution. For instance, let us consider the contract at the address `0x00000000006756ce25b95124e0FC879Cc2B7F1DA`, that starts with the following sequence of instructions (each opcode is preceded by the program counter):

```
0:    PUSH1 0x80
2:    PUSH1 0x40
4:    MSTORE
5:    PUSH1 0x04
7:    CALLDATASIZE
8:    LT
9:    PUSH2 0x327
12:   JUMPI
13:   PUSH0 // skipped by Vandal and EtherSolve
14:   CALLDATALOAD
15:   PUSH1 0xe0
17:   SHR
18:   DUP1
19:   PUSH4 0x54d1f13d
24:   GT
25:   PUSH2 179
```

[9]`https://eips.ethereum.org/EIPS/eip-1153`
[10]`https://eips.ethereum.org/EIPS/eip-3855`

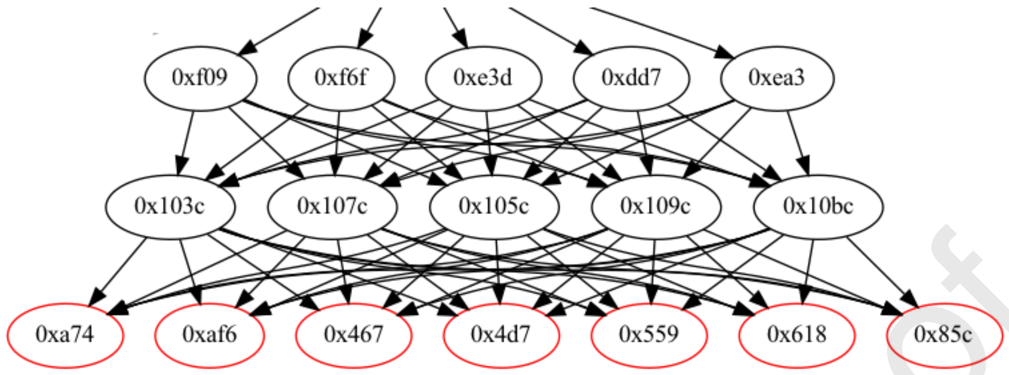


Figure 7: Fragment of the CFG produced by Vandal for the contract at address `0x098008be8a62635979635babe85dafbae31f0f0a`.

```
28:  JUMPI
```

While EVMLiSA correctly adds the edge (12, 13) (corresponding the false edge of the `JUMPI` node at program counter 12), both Vandal and EtherSolve wrongly add the edge (12, 14), skipping the `PUSH0` opcode. This leads to a misidentification of basic blocks and edges. As a result, comparing EVMLiSA with these tools was challenging, as their CFGs structurally differ due to the missing instruction, given that these opcodes occurred frequently in the benchmark suite.

The third and fourth columns of Table 2 report the average number of basic blocks and edges identified by the tools, respectively. It is worth noting that Vandal identified the highest number of basic blocks. As already explained, these values should be interpreted with some care, as Vandal and EtherSolve mishandle some opcodes. Compared to Mythril, EVMLiSA shows a comparable average number of basic blocks and edges, while EtherSolve and Gigahorse identify, on average, fewer basic blocks and edges than EVMLiSA.

Regarding execution time, EtherSolve is the fastest tool among those evaluated, while Mythril is the slowest. Nevertheless, EVMLiSA maintains competitive execution times, even when compared to Vandal and Gigahorse.

On the same benchmark suite, Table 3 reports a qualitative evaluation where each tool is compared against EVMLiSA in terms of edge coverage. For each tool, we report the total number of generated CFGs, and then we split them in two groups. The **Comparable** group contains CFGs whose edge set can be compared with the one generated by EVMLiSA for the same contract, that we refer to as the baseline. By comparable we mean that the

Table 3: Comparison of CFG reconstruction tools over 1697 contracts from Etherscan, using the edges of the CFGs generated by EVMLiSA as baseline. Tools labeled with (∗) ignore opcodes. For each tool, we report the CFGs successfully generated (**Total**), and we partition them depending on whether their edge set can be compared (**Comparable**) or not (**Incomparable**) with the baseline in terms of set inclusion. For comparable ones, we report: CFGs with the same edges as the baseline (=), CFGs with a strict subset of the baseline (⊂), and CFGs with a strict superset of the baseline (⊃). For incomparable ones, we report: CFGs with less edges than the baseline (<), CFGs with more edges than the baseline (>), and CFGs with the same number of edges as the baseline (=).

Tool	**Total**	**Comparable**				**Incomparable**			
		=	⊂	⊃	**Total**	<	>	=	**Total**
Mythril	1406	19	**97**	0	116	**1134**	153	3	1290
Gigahorse	1393	7	**163**	0	170	**1223**	0	0	1223
Vandal (∗)	1685	45	8	**48**	101	**1158**	425	1	1584
EtherSolve (∗)	1678	26	**43**	20	89	**1400**	188	1	1589

two edge sets are either equal (column '='), or one of them strictly contains the other (columns '⊂' and '⊃', where column '⊂' contains the number of CFGs for which the considered tool computes a strict subset of the edges computed by the baseline). The **Incomparable** group instead contains CFGs for which no inclusion relation holds. For these, we resort to the comparison of the number of generated edges: column '<' contains the number of CFGs having less edges than those computed by the baseline, column '>' contains the number of CFGs having more edges than those computed by the baseline, and column '=' contains the number of CFGs that, while being different, still contain the same number of edges than the baseline.

Notably, Mythril and Gigahorse never generate CFGs that fully contain EVMLiSA's ones, while the converse is true in 97 and 163 smart contracts, respectively. Instead, EtherSolve's and Vandal's CFGs fully include all of EVMLiSA's CFG edges in 20 and 48 smart contracts, respectively, whereas the converse is true in 43 and 8 smart contracts, respectively. However, as previously discussed, Vandal and EtherSolve skip the more recent opcodes (e.g., `PUSH0`): some inclusion relations might be missed due to the incorrect edges generated by such tools. Manual inspection also confirmed what was already guessed in [50]: when Vandal cannot resolve the destination of a jump, it over-approximates it assuming all basic blocks as possible destinations (in manner similar to our conservative approach). For instance, let us consider the contract with address `0x098008be8a62635979635babe85dafbae31f0f0a`.

Fig. 7 reports a fragment of the CFG reconstructed by Vandal. The red nodes are basic blocks corresponding to the end blocks of functions, and when Vandal cannot precisely infer a jump destination, as it happens in the nodes at the top of the figure, it connects them to all possible exit points. Since the conservative mode of EVMLiSA is disabled in this experiment, Vandal's over-approximation explains why few of EVMLiSA's CFGs include ones generated by Vandal.

In the Incomparable group, EVMLiSA generates more edges on significantly more smart contracts compared to all the other tools. Nevertheless, Vandal is the one that most frequently produces additional edges not present in EVMLiSA's output (425 contracts in total). Again, this is due to Vandal's over-approximation strategy, which is applied when it cannot resolve a jump destination, as previously discussed.

5.2. Reentrancy vulnerabilities detection

To investigate the benefits of sound CFGs, we implemented a checker in EVMLiSA that detects one of the most well-known and critical issues for the Ethereum blockchain, i.e., the *reentrancy* vulnerability [9, 56], which allowed over \$50 million to be hacked in 2016 [51]. EVMLiSA adopts a similar approach to the one used by EtherSolve, leveraging the previously built CFG to detect reentrancy vulnerabilities. The checker traverses the CFG produced by EVMLiSA to identify specific execution paths that might lead to reentrancy attacks. First, we identify the `CALL` opcodes, which execute external calls. For each of these nodes, the checker examines the set of abstract stacks reaching the node, checking the address used by the `CALL` opcode for the external call, i.e., the second top-most element of each abstract stack. If the address is unknown, the node is flagged as potentially unsafe. Next, the checker determines whether an `SSTORE` opcode (i.e., an opcode that modifies contract's state) is reachable from an unsafe `CALL` opcode. If a path exists between a `CALL` and a `SSTORE` opcode, the checker reports a potential reentrancy vulnerability.

In the following subsections, we will compare EVMLiSA with EtherSolve [50], Mythril [12], and Vandal [10], assessing the precision of EVMLiSA in detecting reentrancy vulnerabilities in EVM bytecode smart contracts. We decided to compare EVMLiSA with tool working at the EVM bytecode level, excluding the ones operating at source level (e.g., Solidity or Vyper), such as Slither [27] or SmartCheck [61]. EtherSolve has already conducted a comprehensive evaluation of these tools in the context of reentrancy vulnerability detection in [50], making a redundant comparison unnecessary. Additionally, the authors of [50] show that EtherSolve outperforms the aforementioned tools, with the exception of Slither. Their evaluation used the SolidiFI dataset, which consists of 50 Ethereum smart contracts written in Solidity whose source code had been injected with reentrancy vulnerabilities.

In Sections 5.2.1 and 5.2.2 we consider two datasets (SolidiFI, already used in [50], and SmartBugs) and we measure the following performance metrics of the selected tools in detecting reentrancy vulnerabilities on these datasets (T_P, F_P, F_N denote true positives, false positives, and false negatives, respectively):

$$\begin{aligned} \text{Precision :} \quad & P = \frac{|T_P|}{|T_P| + |F_P|} \\ \text{Recall :} \quad & R = \frac{|T_P|}{|T_P| + |F_N|} \\ \text{F-measure :} \quad & F_1 = 2 \cdot \frac{P \cdot R}{P + R} \end{aligned}$$

5.2.1. SolidiFI

The SolidiFI dataset[11] includes 50 Solidity files containing several smart contract implementations intentionally injected with reentrancy vulnerabilities. When evaluating EVMLiSA on this dataset we first observed that some of the vulnerabilities, injected in the Solidity source code, were disappearing in the corresponding compiled EVM bytecode. We manually inspected each EVM bytecode contract compiled from the corresponding Solidity source code and discovered that the vulnerabilities injected in contract interfaces whose methods were not invoked elsewhere in the source code disappeared in the corresponding compiled EVM bytecode. Let us consider the Solidity code fragment reported in Figure 8a, corresponding to a fragment of the file no. 12 (i.e., `buggy_12.sol`) of the SolidiFI dataset. The source code includes some contracts, and interfaces definitions that correspond to the ERC-20 and ERC-223 standards. These interfaces define the functions `withdrawFunds_re_ent31` and `withdrawFunds_re_ent3`, respectively. As noted by the authors of the

[11]Available at: `https://github.com/DependableSystemsLab/SolidiFI-benchmark`.

SolidiFI dataset, these functions are expected to be vulnerable to reentrancy attacks due to the use of `msg.sender.send(_weiToWithdraw)` (line 8, Figure 8a) and `msg.sender.call.value(_weiToWithdraw)("")` (line 19, Figure 8a), respectively. However, while both functions appear susceptible to reentrancy vulnerabilities, a closer inspection of the compiled bytecode reveals that only one of them actually results in a vulnerability at the EVM level. As shown in Figure 8b, after compilation, the interface contracts do not contain any executable EVM bytecode (i.e., both the `"bin"` and `"bin-runtime"` fields are empty at lines 9–10 and 17–18 of Figure 8b) though their methods remain visible in the ABI [60] (e.g., `"abi"` fields at lines 6–8 and 14–16 of Figure 8b). This behavior is expected, since in Solidity interfaces are abstract and do not generate executable bytecode unless inherited by a concrete contract. For example, the `ERC223Token` contract inherits from the `ERC223` interface (line 26, Figure 8a) and, as a result, the function `withdrawFunds_re_ent3` appears in the ABI of `ERC223Token` (lines 15 and 24, Figure 8b). Additionally, `ERC223Token` does produce EVM bytecode, meaning that the `"bin"` and `"bin-runtime"` fields are populated (lines 26–27, Figure 8b). This means that the reentrancy vulnerability in `withdrawFunds_re_ent3` is effectively exploitable at runtime. In contrast, the `ERC20` interface is not inherited by any contract within the file number 12. As a result, no concrete contract implements the `withdrawFunds_re_ent31` function, and thus no EVM bytecode is generated for it. Despite its presence in the ABI, the function cannot be invoked on-chain, and the potential vulnerability it represents cannot be exploited in practice.

Thus, for this experiment, we refined the Solidity source code by excluding vulnerabilities confined to interfaces that are not reflected in the compiled bytecode, and recompiled all the programs to ensure consistency in the analysis.[12]

We then followed the same experimental setup adopted in [50] for SolidiFI (i.e., analyzing only the longest bytecode produced by the compiler), ensuring consistency and comparability in the benchmarking process. We run EVMLiSA, with parameters $h = 40$, $l = 10$, and $conservative =$ `false`.

Table 4 reports the comparison among the selected tools on the SolidiFI

[12]For each file, the number of removed vulnerabilities w.r.t. the original ground truth provided by SolidiFI (reported as <*sequential id of contract*>:<*number of vulnerabilities*>) are: `11:1, 12:9, 18:1, 20:2, 21:4, 22:7, 29:3, 33:3, 36:7, 37:1, 42:3, 48:1.`

```
pragma solidity >=0.4.23 <0.6.0;
// ...
contract ERC20 {
  // ...
  function withdrawFunds_re_ent31 (uint256 _weiToWithdraw) public {
    require(balances_re_ent31[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(msg.sender.send(_weiToWithdraw));  //bug
    balances_re_ent31[msg.sender] -= _weiToWithdraw;
  }
  //...
}
//...
contract ERC223 {
// ...
  function withdrawFunds_re_ent3 (uint256 _weiToWithdraw) public {
    require(balances_re_ent3[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    (bool success,)= msg.sender.call.value(_weiToWithdraw)("");
    require(success);  //bug
    balances_re_ent3[msg.sender] -= _weiToWithdraw;
  }
// ...
}
// ...
contract ERC223Token is ERC223 {
  // ...
}
// ...
```

(a) Solidity code snippet

```
{
  "contracts":
  {
    "./solidifi/reentrancy/source-code/buggy_12.sol:ERC20":
    {
      "abi": "[{ ...
        "name" : "withdrawFunds_re_ent31",
          ... ]",
      "bin": "",
      "bin-runtime": ""
    },
    "./solidifi/reentrancy/source-code/buggy_12.sol:ERC223":
    {
      "abi": "[{ ...
        "name" : "withdrawFunds_re_ent3",
          ... ]",
      "bin": "",
      "bin-runtime": ""
    },
    ...
    "./solidifi/reentrancy/source-code/buggy_12.sol:ERC223Token":
    {
      "abi": "[{ ...
        "name" : "withdrawFunds_re_ent3",
          , ... ]",
      "bin": "6080...c0032",
      "bin-runtime": "6080...c0032"
    },
    ...
  },
  "version": "0.5.12+commit.7709ece9.Darwin.appleclang"
}
```

(b) Compilation output

Figure 8: buggy_12.sol from SolidiFI

Table 4: Tool comparison on the SolidiFI dataset.

Tool	Precision (%)	Recall (%)	F-measure (%)
EVMLiSA	100.00	100.00	100.00
EtherSolve	100.00	100.00	100.00
Vandal	100.00	28.36	44.19
Mythril	100.00	16.45	28.25

dataset. Both EVMLiSA and EtherSolve achieve best results on the SolidiFI dataset. In contrast, Vandal and Mythril only detect true positives, but fail to identify several vulnerabilities, leading to a lower recall, as shown in the third column of Table 4. A graphical breakdown of tool performance per smart contract is shown in Figure 9. Overall, EVMLiSA confirms its effectiveness and precision in reentrancy detection, showing results aligned with those of EtherSolve.

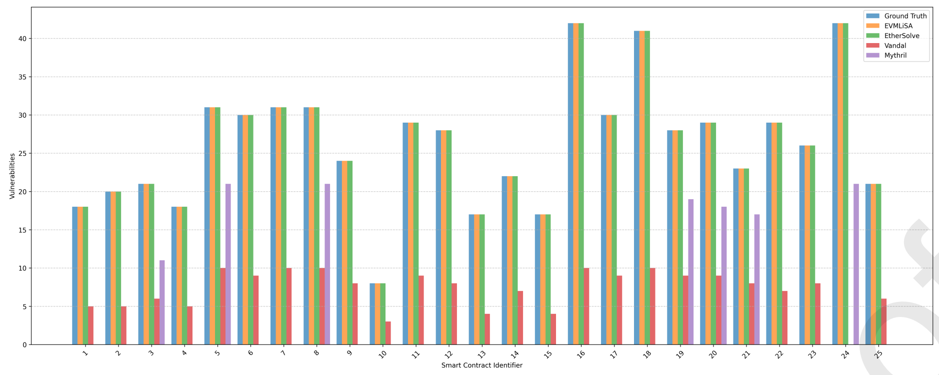
5.2.2. SmartBugs

For the second experiment, we used the SmartBugs dataset,[13] which consists of 31 Solidity files with contracts affected by reentrancy vulnerabilities. As with the SolidiFI dataset, we compiled the Solidity source code into EVM bytecode. We were forced to exclude the last smart contract of the dataset, because we obtained a compilation error both when compiling it locally using `solc` and when compiling it online using Remix;[14] thus, we were left with a final dataset of 30 smart contracts.

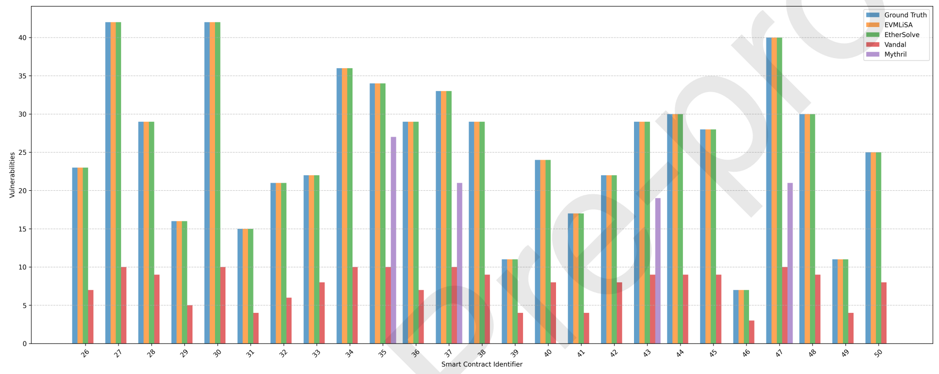
We ran EVMLiSA on this slightly reduced dataset, using the powerset of abstract stacks domain with the parameters $h = 40$, $l = 10$, and $\textit{conservative} = \texttt{false}$. Table 5 reports the comparison among the selected tools on the SmartBugs dataset. EVMLiSA achieves the highest overall performance, with a Precision of 93.75% and perfect Recall, confirming the effectiveness of EVMLiSA in detecting true reentrancy vulnerabilities while producing very few false positives. While EtherSolve achieves a good precision, it suffers from false negatives, resulting in a much lower recall. Vandal attains perfect Recall, successfully identifying all reentrancy vulnerabilities in the dataset, but exhibits significantly lower Precision (41.67%) as it raises several false positives across most smart contracts. On the other hand, Mythril

[13]Available at: `https://github.com/smartbugs/smartbugs-curated`.

[14]`https://remix.ethereum.org/`



(a) Smart contracts 1–25



(b) Smart contracts 26–50

Figure 9: Reentrancy analysis results on the SolidiFI dataset. Bars show the number of reentrancy vulnerabilities on each smart contract as reported by EVMLiSA (orange), EtherSolve (green), Vandal (red), and Mythril (purple), with the correct number of vulnerabilities in blue.

shows both a high false positive rate and some false negatives, resulting in the lowest value for F-measure. Figure 10 reports a graphical representation of the experiment of the SmartBugs dataset, where it is possible to note that EVMLiSA reports false positives for only two smart contracts (11 and 21). Overall, EVMLiSA stands out as the only tool that achieves both soundness and high precision in reentrancy vulnerability detection.

6. Related Work

The Ethereum protocol and EVM have been adopted for the creation of several mainstream blockchain solutions. Indeed, it is not limited to the main network of Ethereum but is also adopted by several other blockchain networks

Table 5: Tool comparison on the SmartBugs dataset.

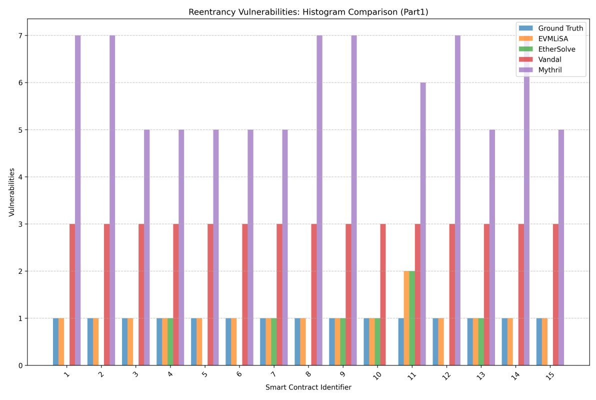
Tool	**Precision** (%)	**Recall** (%)	**F-measure** (%)
EVMLiSA	93.75	100.00	96.77
EtherSolve	85.71	60.00	70.59
Vandal	41.67	100.00	58.82
Mythril	21.01	96.67	34.52

such as Ethereum, Tron, Polygon, Arbitrium, and Avalanche. Furthermore, Ethereum's popularity continues to grow, dominating the decentralized finance (DeFi) with the majority of the Total Value Locked (TVL) among all currently existing blockchains [20]. The abundance of smart contract verification tools and frameworks on Ethereum is driven by its widespread adoption and the critical need for security [9], as these blockchain solutions host a large variety of dApps and financial assets, necessitating reliable methods to audit, test, and ensure the integrity of smart contracts. Static techniques detect issues and software properties without code execution [55]. Advanced verification tools, such as [39, 10, 27, 2, 4, 58, 61, 50] and EVMLiSA, are required for in-depth code analysis and to ensure the detection of non-trivial security issues, bugs, and vulnerabilities. These tools can be applied as automated processes that run periodically in the development pipelines (e.g., at every commit, tag, or before a code release) and provide a comprehensive analysis of code behaviors by accurately considering the semantics of instructions and providing more precise and detailed analysis results. However, their adoption may require non-trivial hardware requirements and their usage may imply heavy computations, with execution time ranging from several seconds to hours. Quite often, the effectiveness of these tools strongly depends on the availability of a precise CFG representation, that accurately models the possible execution paths of a program.

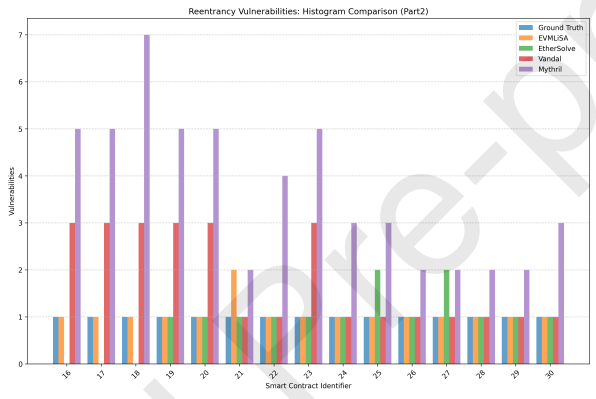
6.1. CFG Reconstruction

Reconstructing a CFG in Ethereum-based smart contracts is typically approached in two main ways: some tools directly operate on the high-level source code, while others analyze the compiled EVM bytecode; note that some tool may apparently accept high-level source code, just to implicitly compile it to EVM bytecode prior to analysis.

A notable tool that operates at the source-level is Slither [27], which constructs an intermediate representation (IR) based on CFGs and a static



(a) Smart contracts 1–15



(b) Smart contracts 16–30

Figure 10: Reentrancy analysis results on the SmartBugs dataset. Bars show the number of reentrancy vulnerabilities on each smart contract as reported by EVMLiSA (orange), EtherSolve (green), Vandal (red), and Mythril (purple), with the correct number of vulnerabilities in blue.

single assignment (SSA) form, preserving semantic information from Solidity and Vyper. It then applies data-flow and taint analyses to detect issues and vulnerabilities. However, Slither does not compute a fully sound CFG in the formal sense, as the reconstruction omits complex flows such as those involving dynamic dispatch and inline assembly, that would require analysis at the bytecode level. In general, tools that analyze high-level languages, such as Solidity and Vyper, do not handle orphan or dynamic jumps, as such constructs are absent in the high-level syntax. Indeed, they work on explicit control structures (e.g., `if`, `for`, `while`), thereby avoiding the complexities of resolving low-level, indirect control flows introduced during compilation. As a consequence, these tools are unable to detect potential vulnerabilities that could have been introduced due to a bug in the compilation process. The applicability of these tools is also limited in practice, since deployed smart contracts typically expose only their bytecode, and the corresponding source code is often unavailable or unverifiable.

Several tools working on EVM bytecode rely on CFGs derived from it, but the CFG reconstruction is often not their main focus. For instance, Albert et al. [4] considers CFGs as a symbolic denotation of transition systems, with the goal of ensuring modularity of programming languages and detecting reentrancy issues. Since the main focus is not on CFG reconstruction, they provide little information on it. Only a small fraction of the literature provides details on the techniques used for CFG reconstruction and the soundness guarantees they provide [59], which in most cases involve symbolic execution techniques, possibly combined with SMT solvers, to explore execution paths and resolve control flow. One of the main limitations of symbolic execution is that not all execution paths are taken into account during the analysis, thus failing to achieve soundness. SMT solvers may also require considerable time to execute, with no termination guarantees. In contrast, Abstract Interpretation trades precision for efficiency: by approximating program behavior, it guarantees that the analysis terminates with sound results, even if this means losing accuracy in the interpretation of program semantics.

A milestone for the smart contract verification based on symbolic execution is certainly Oyente [39, 21]. It computes edges that cannot be statically determined on the fly during a symbolic execution, exploiting the Z3 solver [19] to eliminate provably infeasible traces from consideration. However, Oyente's CFG reconstruction does not support dynamic jumps [3], and it sometimes fails to terminate within time limits [39], making the tool unsound and susceptible to termination issues.

Other noteworthy tools emerged following Oyente's success. Vandal [10] combines decompilation, jump discovery, node-splitting, and incremental dataflow analysis for CFG reconstruction. The decompilation process consists of two phases. First, a symbolic execution translates stack operations into a registry-based IR, and identifies basic blocks and their data dependencies. Then, a data flow analysis computes and propagates constant values that may carry jump addresses until either a fixed-point is reached, or the time/iteration limit is reached. Node splitting is applied to resolve as many jump destinations as possible: a basic block with two different jump destinations gets duplicates. This results in CFG where each path in the original code is a separate branch, each starting from a shared root node. Once the dataflow analysis is complete, any duplicated nodes are merged back together, ensuring the final output matches the original program. Several other notable tools have fully or partially relied on Vandal's components such as MadMax [30] and Gigahorse [29].

Gigahorse [29] decompiles EVM bytecode into a high-level three-address code representation to make data- and control-flow dependencies explicit, applying symbolic execution to compute dynamic jumps' destinations. Elipmoc [31] (aka Gigahorse 2.0, as stated by its authors) is a more recent evolution of Gigahorse. While they share a similar architecture, their algorithms differ, leading to variations in precision, completeness, and scalability. Differences include an improved inference of high-level calls (e.g., Gigahorse fails to recognize some private calls) and a significantly less noisy function argument inference. At a high-level, the argument inference algorithms in Elipmoc and Gigahorse are similar. However, Gigahorse employs a polynomial (non-path-sensitive) algorithm, while Elipmoc maintains paths separately, resulting in a worst-case exponential algorithm. To control complexity, Elipmoc employs several approximations, such as treating paths as sets (rather than sequences) and truncating them at key points. To the best of our knowledge, Vandal, Gigahorse, and Elipmoc are not provably sound.

EthIR [3] constructs CFGs for EVM bytecode and generates a rule-based representation of the program, where code parsing and graph structure are based on Oyente. The authors report that they enhanced Oyente's graph reconstruction to support more jump addresses, providing a sound approach. Specifically, a context-sensitive static analysis is applied to compute a stack-sensitive control flow graph. It analyzes each block separately for every possible stack that can reach jump operations, cloning blocks that might be executed with different execution states [1]. The EVM stack is represented as explicit variables during the analysis. EthIR is however limited to programs without recursions, as only in such cases can the stack be effectively flattened. Hence, the tool might fail to generate a CFG [2] primarily due to recursion and higher-order programming constructs. Nevertheless, when the CFG is successfully generated, it has been formally proven to be sound [1].

eThor [58] performs an analysis using HoRSt specifications, defining a state for each program point. This approach allows it to infer potential jump destinations for each block. Once the control flow of a contract is built, rules are encoded and fed to Z3 to determine if the contract is vulnerable. eThor comes with a rigorous soundness proof but only supports the verification of reachability properties realized by Horn clause resolution. The reconstructed control flow is obtained by a Soufflé [34] program, which was created by manually translating a HoRSt specification. The challenge of sound reconstruction is addressed by basing the corresponding pre-analysis on a proper relaxation of the provably sound abstract semantics in the Soufflé format, ensuring that the original soundness guarantees are inherited. A complementary tool of eThor is HoRStify [32], which provides an automatic way to perform sound security analysis, allowing the verification of a special class of 2-safety properties (including trace non-interference). However, regarding the CFG reconstruction, the sound computation of jump destinations has been left as orthogonal/future work.

Finally, EtherSolve [50] can be considered a state-of-the-art tool for CFG reconstruction, empirically demonstrating a high degree of precision. Specifically, EtherSolve achieves a success rate of 99.7% for the CFG reconstruction on a dataset of real-world smart contracts, outperforming Vandal (97.8%), Gigahorse (95.1%), and EthIR (21.2%). EtherSolve's algorithm executes the stack symbolically, walking through the CFG using a DFS keeping a snapshot of the stack state for each basic block. Only the opcodes in the `PUSH`, `DUP` and `SWAP` families are considered, along with `AND` and `POP`. For every other opcode the symbolic stack pops and pushes unknown elements, as they do not deal with the jump addresses. Furthermore, it applies the following constraints to avoid infinite computations: (i) an edge cannot be analyzed more than once with the same symbolic stack state, and (2) there is a limit on the number of elements to compare when checking for stack equivalence. Additionally, when the DFS encounters a block ending with a jump, only its destination block (obtained from the symbolic stack) is added to the queue, avoiding infeasible paths. During CFG computation, decorators are also applied to provide additional information (e.g., the dispatcher, the fallback function, and the last basic block of the contract) which can be useful for other purposes, such as vulnerability detection, debugging, or human code assessments. In essence, the approach followed by EtherSolve tends to make under-approximations, as it considers only certain execution flows while ignoring others that are also feasible. As noted by the authors, EtherSolve does not guarantee sound CFG reconstructions.

Compared to these tools, EVMLiSA computes an over-approximation of the invariants at each opcode, considering the semantics of all opcodes present in the analyzed EVM bytecode during the jump computation and resolution. Being based on Abstract Interpretation, EVMLiSA applies over-approximations to consider all possible execution paths, also in case of orphan jumps or recursions, and is thus inherently sound. Moreover, even when the soundness guarantees are turned off (i.e., when *conservative* is set to `false`), EVMLiSA is shown to be on par with EtherSolve, outperforming the other tools.

6.2. Other Verification Techniques, Tools, and Directions

Building on the previous discussion of static analysis and CFG reconstruction, this section expands the review of related work by exploring additional methodologies, tools, and emerging research directions.

6.2.1. Dynamic Analysis

Dynamic analysis detects issues during and after code execution by applying checks in controlled environments or, in some cases, directly within the live environment. A typical example is unit testing, where small parts of the smart contract code are executed to verify that individual functions work correctly with respect to expected results. Frameworks such as Truffle [13] and Hardhat [45] offer tools specifically designed for writing, running, and checking unit tests for Ethereum smart contracts. However, testing presents unique challenges in the blockchain context and, moreover, can only reveal the presence of bugs, not their absence, since it observes only a finite set of finite executions [49].

Another popular testing technique is dynamic fuzzing. Tools based on this approach inject random, unexpected, or malformed input (commonly referred to as fuzz or fuzzed inputs) into a running smart contract. The contract's behavior is then observed under these varying inputs to identify potential vulnerabilities, bugs, or unexpected outcomes. In practice, fuzzing tools are favored by attackers [40], as they are more effective for discovering exploits than for ensuring comprehensive protection. Some notable examples of dynamic fuzzing tools for Ethereum smart contracts include ContractFuzzer [33] and ReGuard [36].

Monitoring tools also fall under the category of dynamic analysis. However, they typically work at the data layer, focusing on analyzing transaction data and logs rather than the code being executed. As a result, if unexpected values or behaviors are detected, these tools offer limited insight into the root cause within the smart contract's code. Furthermore, monitoring is generally applied in the final stages of development or post-deployment, phases in which fixing bugs may not be feasible due to code immutability.

Other dynamic solutions involve dynamic symbolic execution (e.g., Manticore [42]), dynamic analyses combined with machine learning (e.g., [22]), and also mixed approaches between dynamic and static techniques (e.g., SmartScan [57], SMARTIAN [11], Mythril [12]).

6.2.2. Mixed Static Analysis and AI Approaches

Static analysis can also be leveraged in the verification of smart contracts through artificial intelligence (AI) [53, 52]. Many state-of-the-art verification tools based on machine learning techniques require transforming smart contract code into graph-based representations (e.g., CFGs) to be fed to neural networks or to be combined with graph neural network models. Furthermore, verification tools based on static analysis are often employed to generate labeled datasets, which are then used to train deep learning models. Compared to traditional static verification tools, AI-based approaches offer advantages in terms of scalability and adaptability. However, they rely on probabilistic models rather than rigorous formal techniques, and therefore do not provide sound guarantees about their results. Notable examples of AI-based tools include CGE [37], Eth2Vec [8] and ContractWard [64].

6.2.3. Static analysis and Blockchain Interoperability

Recent studies have also applied existing static analyses to deal with blockchain interoperability issues. In particular, they focus on analyzing bridge smart contracts to detect cross-chain inconsistent behaviors, primarily within homogeneous interoperable solutions, e.g., between different Ethereum-based blockchains. For instance, XGuard [63] extracts and analyzes semantic information and the source/destination blockchains involved in bridging operations from Solidity smart contracts. It builds upon the static analysis tool Slither [27], extending it with custom verification modules specifically designed for detecting interoperability issues. In contrast, SmartAxe [35] targets EVM bytecode directly to identify vulnerabilities in cross-chain bridge contracts. It employs both data-flow and control-flow analyses to detect issues such as access control incompleteness and semantic inconsistencies. However, these tools do not offer formal guarantees about their findings, and they suffer from both false positives and false negatives. Moreover, current solutions remain limited to homogeneous settings, leaving heterogeneous interoperability and other forms of cross-chain interactions as open challenges for future research [47].

7. Conclusion

In this paper, we presented a new approach aimed at building sound CFGs from EVM bytecode smart contracts, applying static analysis techniques within the Abstract Interpretation framework. In particular, for each instruction of a smart contract, we compute the approximation of possible stack values involving a parametric abstract domain. Then, we targeted the jump instructions and used the computed stack information to resolve possible jump destinations and build CFG edges thus over-approximating the execution paths of a smart contract. Finally, we implemented our approach in EVMLiSA and evaluated it by reconstructing CFGs for existing smart contracts and detecting reentrancy vulnerabilities in well-known benchmark suites of real-world smart contracts, comparing our results with those of state-of-the-art static analyzers.

In future work, we will investigate how to build sound CFGs in the cross-contract and cross-chain contexts, where for instance parametric *calls* [18] (e.g., `CALL` , `STATICCALL`, and `DELEGATECALL` opcodes) and *events* [24] require an approximation of their possible values to compute the targets of cross-contract and cross-chain invocations [26], respectively. Furthermore, EVMLiSA can potentially be extended to support analysis across homogeneous blockchains based on EVM smart contracts or combined with other LiSA front-ends [44] to achieve an analysis across heterogeneous blockchains (e.g., Hyperledger Fabric [48], Cosmos [48], and Tezos [46]) with smart contracts written in different programming languages. We also plan to implement new checkers for EVMLiSA to soundly detect other critical and well-known vulnerabilities, such as timestamp and randomness dependency [9].

8. Data Availability

The source code of EVMLiSA is publicly available at its official GitHub repository: `https://github.com/lisa-analyzer/evm-lisa`. The materials required to replicate the experimental evaluation presented in this paper are available on Zenodo at `https://zenodo.org/records/15516665` (DOI: 10.5281/zenodo.15516664).

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).

Luca Negrini, *Luca Olivieri*: Work partially supported by SERICS (PE000-00014 - CUP H73C2200089001) and iNEST (ECS00000043 – CUP H43C220005-40006) projects funded by PNRR NextGeneration EU

References

[1] Elvira Albert, Jesús Correas, Pablo Gordillo, Alejandro Hernández-Cerezo Guillermo Román-Díez, and Albert Rubio. Analyzing smart contracts: from evm to a sound control-flow graph. *arXiv preprint arXiv:2004.14437*, 2020. https://doi.org/10.48550/arXiv:2004.14437.

[2] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Don't run on fumes—parametric gas bounds for smart contracts. *Journal of Systems and Software*, 176:110923, 2021.

[3] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 513–520. Springer, 2018. https://doi.org/10.1007/978-3-030-01090-4_30.

[4] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. https://doi.org/10.1145/3428277.

[5] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery. https://doi.org/10.1145/800028.808479.

[6] A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps.* O'Reilly, Sebastopol, CA, USA, 2018.

[7] Vincenzo Arceri, Saverio Mattia Merenda, Greta Dolcetti, Luca Negrini, Luca Olivieri, and Enea Zaffanella. 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 2024, page 11–16, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3678721.3686227.

[8] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. *Blockchain: Research and Applications*, 3(4):100101, 2022. https://doi.org/10.1016/j.bcra.2022.100101.

[9] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[10] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, abs/1809.03981, 2018. http://arxiv.org/abs/1809.03981.

[11] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239, 2021. https://doi.org/10.1109/ASE51524.2021.9678888.

[12] Consensys. Mythril. https://github.com/ConsenSys/mythril Accessed: 08-02-2023.

[13] ConsenSys Software Inc. Truffle Official Website. https://archive.trufflesuite.com/ Accessed: 04-11-2024.

[14] Patrick Cousot. *Principles of Abstract Interpretation.* MIT Press, 2021.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. https://doi.org/10.1145/512950.512973.

[16] Patrick Cousot and Radhia Cousot. Constructive versions of tarski's fixed point theorems. *Pacific journal of Mathematics*, 82(1):43–57, 1979.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2):103–179, 1992. `https://doi.org/10.1016/0743-1066(92)90030-7`.

[18] Cyfrin. Calling other contract. `https://solidity-by-example.org/calling-contract/` Accessed: 12-02-2024.

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

[20] DefiLlama. Total value locked all chains, 2024. `https://defillama.com/chains` Accessed 10/2024.

[21] Enzyme Finance. Oyente: An Analysis Tool for Smart Contracts, 2028. `https://github.com/melonproject/oyente` Accessed: 12-05-2025.

[22] Mojtaba Eshghie, Cyrille Artho, and Dilian Gurov. Dynamic vulnerability detection on smart contracts using machine learning. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, page 305–312, New York, NY, USA, 2021. Association for Computing Machinery. `https://doi.org/10.1145/3463274.3463348`.

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

[24] Ethereum. Solidity documentation - events. `https://docs.soliditylang.org/en/v0.8.24/contracts.html#events` Accessed: 12-02-2024.

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

[26] Ghareeb Falazi, Uwe Breitenbücher, Frank Leymann, and Stefan Schulte. Cross-chain smart contract invocations: A systematic multi-vocal literature review. *ACM Comput. Surv.*, 56(6), January 2024. `https://doi.org/10.1145/3638045`.

[27] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019. `https://doi.org/10.1109/WETSEB.2019.00008`.

[28] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. Static analysis for dummies: experiencing lisa. In Lisa Nguyen Quang Do and Caterina Urban, editors, *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*, pages 1–6. ACM, 2021. `https://doi.org/10.1145/3460946.3464316`.

[29] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019. `https://doi.org/10.1109/ICSE.2019.00120`.

[30] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: analyzing the out-of-gas world of smart contracts. *Commun. ACM*, 63(10):87–95, September 2020. `https://doi.org/10.1145/3416262`.

[31] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: advanced decompilation of ethereum smart contracts. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. `https://doi.org/10.1145/3527321`.

[32] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. Horstify: Sound security analysis of smart contracts. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 245–260, 2023. `https://doi.org/10.1109/CSF57540.2023.00023`.

[33] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 259–269, New York, NY, USA, 2018. Association for Computing Machinery. `https://doi.org/10.1145/3238147.3238177`.

[34] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing. `https://doi.org/10.1007/978-3-319-41540-6_23`.

[35] Zeqin Liao, Yuhong Nan, Henglong Liang, Sicheng Hao, Juan Zhai, Jiajing Wu, and Zibin Zheng. Smartaxe: Detecting cross-chain vulnerabilities in bridge smart contracts via fine-grained static analysis. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. `https://doi.org/10.1145/3643738`.

[36] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 65–68, New York, NY, USA, 2018. Association for Computing Machinery. `https://doi.org/10.1145/3183440.3183495`.

[37] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 35(2):1296–1310, 2023. `https://doi.org/10.1109/TKDE.2021.3095196`.

[38] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In Laurie Hendren, editor, *Compiler Construction*, pages 197–212, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

[40] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software*

Engineering, 47(11):2312–2331, 2021. https://doi.org/10.1109/TSE.2019.2946563.

[41] Antoine Miné et al. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages*, 4(3-4):120–372, 2017.

[42] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[43] Luca Negrini, Vincenzo Arceri, Luca Olivieri, Agostino Cortesi, and Pietro Ferrara. Teaching through practice: Advanced static analysis with lisa. In Emil Sekerinski and Leila Ribeiro, editors, *Formal Methods Teaching*, pages 43–57, Cham, 2024. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-71379-8_3.

[44] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. LiSA: A generic framework for multilanguage static analysis. In Vincenzo Arceri, Agostino Cortesi, Pietro Ferrara, and Martina Olliaro, editors, *Challenges of Software Verification*, pages 19–42. Springer Nature Singapore, Singapore, 2023. https://doi.org/10.1007/978-981-19-9601-6_2.

[45] Nomic Foundation. Hardhat Official Website. https://hardhat.org/ Accessed: 04-11-2024.

[46] Luca Olivieri, Thomas Jensen, Luca Negrini, and Fausto Spoto. Michelsonlisa: A static analyzer for tezos. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 80–85, 2023. https://doi.org/10.1109/PerComWorkshops56833.2023.10150247.

[47] Luca Olivieri, Aradhita Mukherjee, Nabendu Chaki, and Agostino Cortesi. Cross-chain smart contracts and dapps verification by static analysis: Limits and challenges. In *CEUR Workshop Proceedings*, volume 3962. CEUR-WS, 2025. Joint National Conference on Cybersecurity (ITASEC & SERICS 2025).

[48] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–25, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2023.23.

[49] Luca Olivieri and Fausto Spoto. Software verification challenges in the blockchain ecosystem. *International Journal on Software Tools for Technology Transfer*, 26(4):431–444, 2024. https://doi.org/10.1007/s10009-024-00758-x.

[50] Michele Pasqua, Andrea Benini, Filippo Contro, Marco Crosara, Mila Dalla Preda, and Mariano Ceccato. Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode. *J. Syst. Softw.*, 200:111653, 2023. https://doi.org/10.1016/J.JSS.2023.111653.

[51] Nathaniel Popper. A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency. *The New York Times*, 2016. June 17th.

[52] Dalila Ressi, Riccardo Romanello, Carla Piazza, and Sabina Rossi. Ai-enhanced blockchain technology: A review of advancements and opportunities. *Journal of Network and Computer Applications*, 225:103858, 2024. https://doi.org/10.1016/j.jnca.2024.103858.

[53] Dalila Ressi, Alvise Spanò, Lorenzo Benetollo, Carla Piazza, Michele Bugliesi, and Sabina Rossi. Vulnerability detection in ethereum smart contracts via machine learning: A qualitative analysis. *arXiv preprint arXiv:2407.18639*, 2024. https://doi.org/10.48550/arXiv.2407.18639.

[54] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. http://www.jstor.org/stable/1990888.

[55] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, Cambridge, MA, USA, 2020.

[56] Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in ethereum smart contracts. *CoRR*, abs/2105.02881, 2021. https://arxiv.org/abs/2105.02881.

[57] Noama Fatima Samreen and Manar H. Alalfi. Smartscan: An approach to detect denial of service vulnerability in ethereum smart contracts. In *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 17–26, 2021. https://doi.org/10.1109/WETSEB52558.2021.00010.

[58] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 621–640, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3372297.3417250.

[59] Clara Schneidewind, Markus Scherer, and Matteo Maffei. The good, the bad and the ugly: Pitfalls and best practices in automated sound static analysis of ethereum smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 212–231, Cham, 2020. Springer International Publishing.

[60] The Solidity Authors. Solidity Documentation | ABI specification, 2024. https://docs.soliditylang.org/en/v0.8.28/abi-spec.html Accessed 04/2025.

[61] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, 2018.

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

[63] Ke Wang, Yue Li, Che Wang, Jianbo Gao, Zhi Guan, and Zhong Chen. Xguard: Detecting inconsistency behaviors of crosschain bridges. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 612–616, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3663529.3663809.

[64] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2021. https://doi.org/10.1109/TNSE.2020.2968505.

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

Appendix A. Soundness Proofs

Appendix A.1. Completeness of the $\mathcal{S}_h^l$ lattice

Theorem 8. $\langle \mathcal{S}_h^l, \subseteq, \sqcup_{\mathcal{S}_h^l}, \cap, \emptyset, \mathcal{S}_h \rangle$ is a complete ACC lattice.

Proof. $\mathcal{S}_h^l$ is the subset of the complete powerset lattice $\langle \wp(\mathcal{S}_h), \subseteq, \cup, \cap, \emptyset, \mathcal{S}_h \rangle$ where all elements $X \in \wp(\mathcal{S}_h) \, : \, X \neq \mathcal{S}_h \wedge |X| > l$ are missing. Since $\sqcup_{\mathcal{S}_h^l}$ "skips" the missing elements, $\mathcal{S}_h^l$ is closed under $\sqcup_{\mathcal{S}_h^l}$ and $\cap$, and any $X \subseteq \mathcal{S}_h^l$ trivially has worst-case lub $\mathcal{S}_h$ and glb $\emptyset$. Moreover, since it has a finite height of $l + 2$, it also satisfies ACC: analyses using $\mathcal{S}_h^l$ will converge in a finite number of fixpoint iterations. □

Appendix A.2. Monotonicity of γ

Theorem 9. γ is monotone.

Proof. We prove that $\forall \widehat{S}_1, \widehat{S}_2 \in \mathcal{S}_h^l \, : \, \widehat{S}_1 \subseteq \widehat{S}_2 \implies \gamma(\widehat{S}_1) \subseteq \gamma(\widehat{S}_2)$. Under the assumption that $\widehat{S}_1 \subseteq \widehat{S}_2$, we consider w.l.o.g. that $\widehat{S}_2 = \widehat{S}_1 \cup \widehat{S}'$ for some non-empty $\widehat{S}_1$ and $\widehat{S}'$, and we expand the definition of $\gamma(\widehat{S}_2)$.

$$\begin{aligned}
& \gamma(\widehat{S}_2) \\
&= \gamma(\widehat{S}_1 \cup \widehat{S}') && \wr \text{hypothesis} \wr \\
&= \gamma(\{\hat{s}_0^1, \ldots, \hat{s}_n^1\} \cup \{\hat{s}'_0, \ldots, \hat{s}'_m\}) && \wr \text{set expansion} \wr \\
&= \overline{\gamma}(\hat{s}_0^1) \cup \cdots \cup \overline{\gamma}(\hat{s}_n^1) \cup \overline{\gamma}(\hat{s}'_0) \cup \cdots \cup \overline{\gamma}(\hat{s}'_m) && \wr \text{def. } \gamma \wr \\
&= \gamma(\widehat{S}_1) \cup \gamma(\widehat{S}') && \wr \text{def. } \gamma \wr \quad \square
\end{aligned}$$

Appendix A.3. Soundness of $[\![\mathtt{op}]\!]^\sharp$

Theorem 10 (Soundness of $[\![\mathsf{op}]\!]^\sharp$)**.** $[\![\mathsf{op}]\!]^\sharp$ is a sound over-approximation of $[\![\mathsf{op}]\!]$. Formally:

$$\forall \widehat{S} \in \mathcal{S}_h^l \, : \, [\![{}^\ell\mathsf{op}_\delta^\rho]\!] \gamma(\widehat{S}) \subseteq \gamma([\![{}^\ell\mathsf{op}_\delta^\rho]\!]^\sharp \widehat{S})$$

Proof. Both the concrete and abstract semantics are defined as the additive lift of the semantics on individual stacks. Thus, we first prove the soundness of the latter computation, that is:

$$\forall \hat{s} \in \mathcal{S}_h \, : \, [\![{}^\ell\mathsf{op}_\delta^\rho]\!] \overline{\gamma}(\hat{s}) \subseteq \overline{\gamma}([\![{}^\ell\mathsf{op}_\delta^\rho]\!]^\sharp \hat{s})$$

Let us assume that $\hat{s} \neq \bot_{\mathcal{S}_h}$, $[\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]^{\sharp}\hat{s} \neq \bot_{\mathcal{S}_h}$, and $\forall s \in \gamma(\hat{s}) : [\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]s \neq \bot_{\mathbb{S}}$, since in all three cases computing both in the concrete or in the abstract semantics would trivially result in $\bot_{\mathbb{S}}$. Moreover, to simplify the proof, let us assume that $\delta = \rho < h$. The proof for other combinations of δ, ρ ,and h are analogous.

$$
\begin{aligned}
&[\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]\overline{\gamma}(\hat{s}) && \\
=&[\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]\overline{\gamma}([v_0,\dots,v_{h-1-\delta},\dots,v_{h-1}]) && \wr\text{def. } \hat{s}\int \\
=&\{[\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!][\dots,z_0,\dots,z_{h-1-\delta},\dots,z_{h-1}]\} && \wr\text{def. } \overline{\gamma}\int \\
=&\{[\dots,z_0,\dots,z_{h-1-\delta},\dot{z}_0,\dots,\dot{z}_{\rho-1}]\} && \wr\text{def. } [\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]\int \\
\subseteq&\overline{\gamma}([v_0,\dots,v_{h-1-\delta},\dot{v}_0,\dots,\dot{v}_{\rho-1}]) && \wr\text{def. } \overline{\gamma}\int \\
=&\overline{\gamma}(\mathsf{push}^{\rho}(\mathsf{pop}^{\delta}(\hat{s}),\dot{v}_0,\dots,\dot{v}_{\rho-1}])) && \wr\text{def. } \mathsf{push}^{\rho},\mathsf{pop}^{\delta}\int \\
=&\overline{\gamma}([\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]^{\sharp}\hat{s}) && \wr\text{def. } [\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]^{\sharp}\int
\end{aligned}
$$

We can now prove the soundness of the overall semantics:

$$
\begin{aligned}
&[\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]\gamma(\widehat{S}) && \\
=&\bigcup_{\hat{s}\in\widehat{S}}[\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]\overline{\gamma}(\hat{s}) && \wr\text{lift, def. } \gamma\int \\
\subseteq&\bigcup_{\hat{s}\in\widehat{S}}\overline{\gamma}([\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]^{\sharp}\hat{s}) && \wr\text{prev. proof}\int \\
=&\gamma([\![{}^{\ell}\mathsf{op}^{\rho}_{\delta}]\!]^{\sharp}\widehat{S}) && \wr\text{def. } \gamma\int \quad \square
\end{aligned}
$$

Appendix A.4. Soundness of BUILD*CFG*

Being our approach based on Abstract Interpretation sound-by-design (provided the necessary proofs), the soundness of the static analysis computing the sets of abstract stacks for each node of a CFG is guaranteed by Theorems 8, 9, and 10 from Appendix A.1, Appendix A.2, and Appendix A.3, respectively. Thus, we prove the soundness of the generated CFG assuming that these hold, and relying on the notion of *partial CFG*.

Definition 11 (CFG for program P)**.** Given an EVM program P, let $N = \{\ell \mid \Pi(\ell) \text{ is an opcode of } \mathtt{P}\}$. Let $G_{\bot} = (N, E_{\bot})$, where

$$E_{\bot} = \{\ell \to \mathsf{next}(\ell) \mid \ell \in N\};$$

Let $G_\top = (N, E_\top)$ where

$$E_\top = E_\bot \cup \{\ell \to \ell' \mid \Pi(\ell) \in \{\texttt{JUMP}, \texttt{JUMPI}\} \wedge \Pi(\ell') = \texttt{JUMPDEST}\}.$$

Then, a CFG $\texttt{G} = (N, E)$ for P is a CFG satisfying $E_\bot \subseteq E \subseteq E_\top$.

Note that the CFG produced by PARTIALCFG(P) is exactly $G_\bot$, as it contains a node for each opcode of P and all syntactically occurring edges having the form $\ell \to \texttt{next}(\ell)$ (these also include the false edges of JUMPI nodes). We now define a lattice structure on CFGs for a program using the subgraph relation as partial order.

Lemma 12. Given an EVM program P, let $\mathbb{G}_\texttt{P}$ be the set of all CFGs satisfying Definition 11. Then $\langle \mathbb{G}_\texttt{P}, \sqsubseteq_E, \sqcup_E, \sqcap_E, G_\bot, G_\top \rangle$, where $(N, E_1) \sqsubseteq_E (N, E_2)$ if and only if $E_1 \subseteq E_2$, is a finite (hence complete and ACC) lattice. The lub anbd glb operators are those induced by $\sqsubseteq_E$.

Proof. The proof is straightforward, since $\mathbb{G}_\texttt{P}$ is finite. □

We now show that procedure JUMPSOLVER implements an extensive and monotone operator on the CFG lattice for program P.

Lemma 13 (JUMPSOLVER is extensive)**.** Given an EVM program P and $G \in \mathbb{G}_\texttt{P}$ JUMPSOLVER produces a CFG $G' \in \mathbb{G}_\texttt{P}$ such that $G \sqsubseteq_E G'$.

Proof. Straightforward, since the procedure can only add new edges. □

Lemma 14 (JUMPSOLVER is monotone)**.** Given an EVM program P and $G_1, G_2 \in \mathbb{G}_\texttt{P}$, let G'_i be the CFG computed by JUMPSOLVER(G_i, h, l, c), for $i = 1, 2$. Then $G_1 \sqsubseteq_E G_2 \implies G'_1 \sqsubseteq_E G'_2$.

Proof. The static analysis at line 2 of Pseudocode 2 implements a monotonic function (in particular, since the abstract domain satisfies ACC, we do not use widening operators). Hence, since all the edges of G_1 also occur in G_2, all new edges added to G_1 will also be added to G_2, thereby obtaining $G'_1 \sqsubseteq_E G'_2$. □

Theorem 15. BUILDCFG always terminates and computes the least fixpoint of JUMPSOLVER on the lattice $\mathbb{G}_\texttt{P}$.

Proof. Immediate from Lemmas 12, 13 and 14. □

Finally, we can prove the soundness of BUILDCFG.

Theorem 16 (buildCFG is sound)**.** Let P be an EVM program whose first label is ℓ_0. Then buildCFG$(\mathtt{P}, h, l, \mathtt{true})$ returns a sound CFG $G = (N, E)$ for P. Namely, for all ${}^{\ell}\mathtt{op}^{\rho}_{\delta} \in \mathtt{P}$:

$$\{\ell \to \ell' \mid \exists s, s' \in \mathbb{S}\,.\,\Xi^{\ell}(\mathtt{P}, \langle [], \ell_0 \rangle) = \langle s, \ell \rangle \wedge [\![{}^{\ell}\mathtt{op}^{\rho}_{\delta}]\!]\langle s, \ell \rangle = \langle s', \ell' \rangle\} \subseteq E.$$

Proof. By Theorem 15, procedure buildCFG always terminates producing a CFG $G = (N, E)$ for P.

From Theorems 8, 9, and 10, we known that the abstract stacks reaching each jump are a sound over-approximation of those occurring in the concrete executions.

Suppose first that the parameter l is sufficiently large to never cause $\widehat{S} = \mathcal{S}_h$. jumpSolver will thus process all jumps with lines 10–17. These will have a finite number of stacks reaching them, whose top element v will be either:

1. $v \neq \top_{\mathbb{Z}^\sharp}$ is a valid jump destination $v \in \mathtt{JD}_{\mathtt{P}}$: in this case, the edge representing the concrete jump is added, ensuring soundness;

2. $v \neq \top_{\mathbb{Z}^\sharp}$ is an invalid jump destination ($v \notin \mathtt{JD}_{\mathtt{P}}$ or $v = \top_{\overline{\mathbb{Z}}}$) or an uninitialized stack element $\varnothing$: in this case, the concrete execution would always raise an error; hence, soundness is preserved even if we no edge is added;

3. an unknown value $\top_{\mathbb{Z}^\sharp}$: since *conservative* $= \mathtt{true}$, jumpSolver adds edges to all possible destinations to ensure soundness.

Instead, if the parameter l is too small and it causes $\widehat{S} = \mathcal{S}_h$ at any jump opcode, since *conservative* $= \mathtt{true}$, jumpSolver will add edges to all possible destinations in this case too, once again ensuring soundness. □

While we proved soundness for the case where *conservative* $= \mathtt{true}$, it is worth noting that if case (3) does not occur, soundness is ensured even when *conservative* $= \mathtt{false}$, as every possible jump target in the concrete semantics is still accounted for in the CFG returned by buildCFG.

Conflict of Interest

The authors have declared no conflict of interest

Corresponding author: Vincenzo Arceri (vincenzo.arceri@unipr.it)