Analyzing Smart Contracts: From EVM to a sound Control-Flow Graph

Elvira Albert[1,2], Jesús Correas[2], Pablo Gordillo[2], Alejandro Hernández-Cerezo[2], Guillermo Román-Díez[3], and Albert Rubio[1,2]

[1] Instituto de Tecnología del Conocimiento, Spain
[2] Complutense University of Madrid, Spain
[3] Universidad Politécnica de Madrid, Spain

Abstract. The EVM language is a simple stack-based language with words of 256 bits, with one significant difference between the EVM and other virtual machine languages (like Java Bytecode or CLI for .Net programs): the use of the stack for saving the jump addresses instead of having it explicit in the code of the jumping instructions. Static analyzers need the complete control flow graph (CFG) of the EVM program in order to be able to represent all its execution paths. This report addresses the problem of obtaining a precise and complete stack-sensitive CFG by means of a static analysis, cloning the blocks that might be executed using different states of the execution stack. The soundness of the analysis presented is proved.

1 EVM Language

The EVM language is a simple stack-based language with words of 256 bits with a local volatile memory that behaves as a simple word-addressed array of bytes, and a persistent storage that is part of the blockchain state. A more detailed description of the language and the complete set of operation codes can be found in [6]. In this section, we focus only on the relevant characteristics of the EVM that are needed for describing our work.

Example 1. In order to describe our techniques, we use as running example a simplified version (without calls to the external service Oraclize and the authenticity proof verifier) of the contract [1] that implements a lottery system. During a game, players call a method `joinPot` to buy lottery tickets; each player's address is appended to an array `addresses` of current players, and the number of tickets is appended to an array `slots`, both having variable length. After some time has elapsed, anyone can call `rewardWinner` which calls the `Oraclize` service to obtain a random number for the winning ticket. If all goes according to plan, the `Oraclize` service then responds by calling the `__callback` method with this random number and the authenticity proof as arguments. A new instance of the game is then started, and the winner is allowed to withdraw her balance using a `withdraw` method. Figure 2 shows an excerpt of the Solidity code (including the public function `findWinner`) and a fragment of the EVM code produced by the compiler. The Solidity source code is shown for readability, as our analysis works directly on the EVM code.

To the right of Figure 2 we show a fragment of the EVM code of method `findWinner`. It can be seen that the EVM has instructions for operating with the stack contents, like DUPx or SWAPx; for comparisons, like LT, GT; for accessing the storage (memory) of the contract, like SSTORE, SLOAD (MLOAD, MSTORE); to add/remove elements to/from the stack, like PUSHx/ POP; and many others (we again refer to [6] for details). Some instructions increment the program counter in several units (e.g., PUSHx Y adds a word with the constant Y of x bytes to the stack and increments the program counter by $x + 1$). In what follows, we use $size(b)$ to refer to the number of units that instruction b increments the value of the program counter. For instance $size(\texttt{POP}) = 1$, $size(\texttt{PUSH1}) = 2$ or $size(\texttt{PUSH3}) = 4$. ∎

One significant difference between the EVM and other virtual machine languages (like Java Bytecode or CLI for .Net programs) is the use of the stack for saving the jump addresses instead of having it explicit in the code of the jumping instructions. In EVM, instructions JUMP and JUMPI will jump, unconditionally and conditionally respectively, to the program counter stored in the top of the execution stack. This feature of the EVM requires, in order to obtain the control flow graph of the program, to keep track of the information stored in the stack. Let us illustrate it with an example.

Example 2. In the EVM code to the right of Figure 2 we can see two jump instructions at program points 953 and 660, respectively, and the jump address (64B and 6D0) is stored in the instruction immediately before them: 950 or 65D. It then jumps to this destination by using the instruction JUMPDEST (program points 941, 64B, 653). ∎

```
                                                        ...
  1  contract EthereumPot {                             64B: JUMPDEST
  2    address [] public addresses ;                    64C: PUSH1 0x00
  3    address public winnerAddress;                    64E: DUP1
  4    uint [] public slots ;                           64F: PUSH1 0x00
  5    function __callback (bytes32 _queryId, string _result , bytes _proof){   651: SWAP1
  6      if (msg.sender != oraclize_cbAddress()) throw;  652: POP
  7      random_number = uint(sha3(_result))            653: JUMPDEST
  8      winnerAddress = findWinner(random_number);     654: PUSH1 0x03
  9      amountWon = this.balance ∗ 98 / 100 ;          656: DUP1
 10      winnerAnnounced(winnerAddress, amountWon);     657: SLOAD
 11      if (winnerAddress.send(amountWon)) {           658: SWAP1
 12        if (owner.send(this.balance)) {              659: POP
 13          openPot();                                 65A: DUP2
 14        }                                            65B: LT
 15      }                                              65C: ISZERO
 16    }                                                65D: PUSH2 0x06D0
 17                                                     660: JUMPI
 18    function findWinner (uint random) constant returns (address winner) {  661: PUSH1 0x03
 19      for (uint i = 0; i < slots .length; i++) {     663: DUP2
 20        if (random <= slots[i]) {                    664: DUP2
 21          return addresses [ i ];                    665: SLOAD
 22        }                                            666: DUP2
 23      }                                              ...
 24    }                                                941: JUMPDEST
 25    // Other functions                               942: MOD
 26  }                                                  943: ADD
                                                        944: PUSH1 0x0A
                                                        946: DUP2
                                                        947: SWAP1
                                                        948: SSTORE
                                                        949: POP
                                                        94A: PUSH2 0x0954
                                                        94D: PUSH1 0x0A
                                                        94F: SLOAD
                                                        950: PUSH2 0x064B
                                                        953: JUMP
                                                        ...
```

Fig. 1. Excerpt of Solidity code for `EthereumPot` contract (left), and fragment of EVM code for function `findWinner` (right)

We start our analysis by defining the set $\mathcal{J}$, which contains all possible jump destinations in an EVM program $P \equiv b_0, \ldots, b_p$:

$$\mathcal{J}(P) = \{pc \mid b_{pc} \in P \wedge b_{pc} \equiv \texttt{JUMPDEST}\}.$$

We use $b_{pc} \in P$ for referring to the instruction at program counter pc in the EVM program P. In what follows, we omit P from definitions when it is clear from the context, e.g., we use $\mathcal{J}$ to refer to $\mathcal{J}(P)$.

Example 3. Given the EVM code that corresponds to function `findWinner`, we get the following set:

$$\mathcal{J} = \{123, 142, 954, 64B, 6D0, 66F, 653, 6C3, 691, 6D1, 6BA\}$$

∎

The first step in the computation of the CFG is to define the notion of *block*. In general [2], given a program P, a *block* is a maximal sequence of straight-line consecutive code in the program with the properties that the flow of control can only enter the block through the first instruction in the block, and can only leave the block at the last instruction. Let us define the concept of block in an EVM program:

Definition 1 (blocks). *Given an EVM program* $P \equiv b_0, \ldots, b_p$, *we define*

$$blocks(P) = \left\{ B_i \equiv b_i, \ldots, b_j \ \middle| \ \begin{array}{l} (\forall k.i < k < j, b_k \notin Jump \cup End \cup \{\textit{JUMPDEST}\}) \ \wedge \\ (\ i{=}0 \vee b_i {\equiv} \textit{JUMPDEST} \vee b_{i-1}{=}\textit{JUMPI} \ ) \ \wedge \\ (\ j{=}p \vee b_j \in Jump \vee b_j \in End \vee b_{j+1}{\equiv}\textit{JUMPDEST} \ ) \end{array} \right\}$$

where

$$Jump = \{\textit{JUMP}, \textit{JUMPI}\}$$
$$End = \{\textit{REVERT}, \textit{STOP}, \textit{INVALID}\}$$

Example 4. Figure 3 shows the blocks (nodes) obtained for `findWinner` and their corresponding jump invocations. Solid and dashed edges represent the two possible execution paths depending on the entry block: solid edges represent the path that starts from block 941 and dashed edges the path that starts from 123. Note that most of the blocks start with a JUMPDEST instruction (123, 941, 64B, 653, 66F, 954,

```
1   contract EthereumPot {
2     address [] public addresses;
3     address public winnerAddress;
4     uint [] public slots;
5     function __callback (bytes32 _queryId, string _result, bytes _proof){
6       if (msg.sender != oraclize_cbAddress()) throw;
7       random_number = uint(sha3(_result))
8       winnerAddress = findWinner(random_number);
9       amountWon = this.balance * 98 / 100;
10      winnerAnnounced(winnerAddress, amountWon);
11      if (winnerAddress.send(amountWon)) {
12        if (owner.send(this.balance)) {
13          openPot();
14        }
15      }
16    }
17
18    function findWinner (uint random) constant returns (address winner) {
19      for (uint i = 0; i < slots.length; i++) {
20        if (random <= slots[i]) {
21          return addresses[i];
22        }
23      }
24    }
25    // Other functions
26  }
```

```
...
64B: JUMPDEST
64C: PUSH1 0x00
64E: DUP1
64F: PUSH1 0x00
651: SWAP1
652: POP
653: JUMPDEST
654: PUSH1 0x03
656: DUP1
657: SLOAD
658: SWAP1
659: POP
65A: DUP2
65B: LT
65C: ISZERO
65D: PUSH2 0x06D0
660: JUMPI
661: PUSH1 0x03
663: DUP2
664: DUP2
665: SLOAD
666: DUP2
...
941: JUMPDEST
942: MOD
943: ADD
944: PUSH1 0x0A
946: DUP2
947: SWAP1
948: SSTORE
949: POP
94A: PUSH2 0x0954
94D: PUSH1 0x0A
94F: SLOAD
950: PUSH2 0x064B
953: JUMP
...
```

Fig. 2. Excerpt of Solidity code for `EthereumPot` contract (left), and fragment of EVM code for function `findWinner` (right)

6C3, 691, 142, 6D1, 6D0). The rest of the blocks start with instructions that come right after a JUMPI instruction (661, 683). Analogously, most blocks end in a JUMP (941, 6C3, 123, 691, 6D1), JUMPI (653, 661, 66F, 683) or RETURN (142) instruction or in the instruction that precedes JUMPDEST (64B). ∎

Observing the blocks in Figure 3, we can see that most JUMP instructions use the address introduced in the PUSH instruction executed immediately before the JUMP. However, in general, in EVM code, it is possible to find a JUMP whose address has been stored in a different block. This happens for instance when a public function is invoked privately from other methods of the same contract, the returning program counter is introduced by the invokers at different program points and it will be used in a unique JUMP instruction when the invoked method finishes in order to *return* to the particular caller that invoked that function.

Example 5. In Figure 3, at block 6D1 we have a JUMP (marked with ☆) whose address is not pushed in the same block. This JUMP takes the returned address from function `findWinner`. If `findWinner` is publicly invoked, it jumps to address 142 (pushed at block 123 at ⋆) and if it is invoked from `__callback` it jumps to 954 (pushed at block 941 at ⋆).

2   From EVM to a complete CFG

As we have seen in the previous section, the addresses used by the jumping instructions are stored in the execution stack. In EVM, blocks can be reached with different stack sizes an contents. As it is used in other tools [4,3,5], to precisely infer the possible addresses at jumping program points, we need a *context-sensitive* static analysis that analyze separately all blocks for each possible stack than can reach them (only considering the addresses stored in the stack). This section presents an *address analysis* of EVM programs which allows us to compute a complete CFG of the EVM code. To compute the addresses involved in the jumping instructions, we define a static analysis which soundly infers all possible addresses that a JUMP instruction could use.

In our address analysis we aim at having the stack represented by explicit variables. Given the characteristics of EVM programs, the execution stack of EVM programs produced from Solidity programs
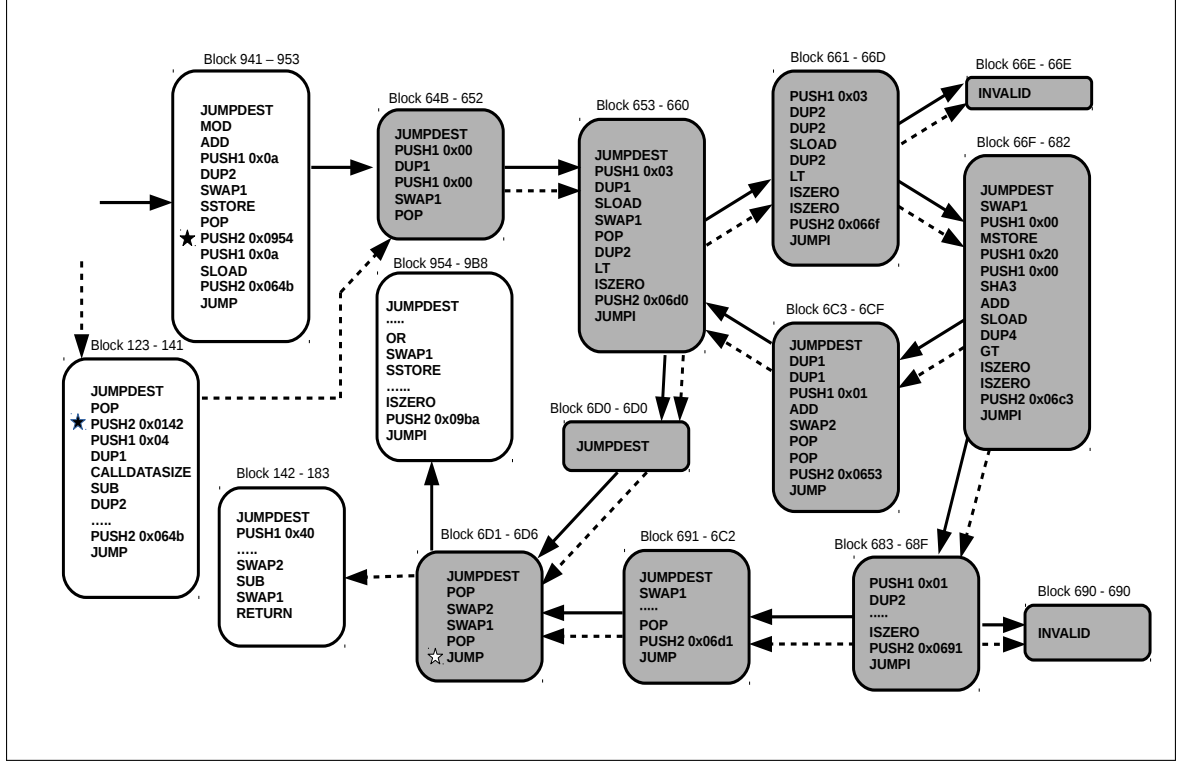
Fig. 3. Fragment of the CFG of `findWinner`

without recursion can be flattened. Besides, as the size of the stack of the Ethereum Virtual Machine is bounded to 1024 elements (see [6]), the number of stack variables is limited. We use $\mathcal{V}$ to represent the set of all possible stack variables that may be used in the program. The first element we define for our analysis is its abstract state:

The abstract state Our analysis uses a partial representation of the execution stack as basic element. To this end, we use the notion of *stack state* as a pair $\langle n, \sigma \rangle$, where n is the number of elements in the stack, and σ is a partial mapping that relates some stack positions with a set of jump destinations. A position in the stack is referred as s_i with $0 \le i < n$, and s_{n-1} is the position at the top of the stack. The *abstract state* of the analysis is defined on the set of all stack states $\mathcal{S} = \{\langle n, \sigma \rangle \mid 0 \le n \le |\mathcal{V}| \wedge \sigma(s) \in \Sigma_n\}$ where Σ_n is the set of all mappings using n stack variables, defined recursively as follows: $\Sigma_i = \Sigma_{i-1} \cup \{\sigma[s_i \mapsto j] \mid \sigma \in \Sigma_{i-1} \wedge j \subseteq \mathcal{J}\}; \Sigma_0 = \{\sigma_\emptyset\}$, where $\sigma_\emptyset$ is the empty mapping.

Definition 2 (abstract state). *The abstract state is a partial* mapping π *of the form* $\mathcal{S} \mapsto \mathcal{P}(\mathcal{S})$.

The application of σ to an element s_i, that is, $\sigma(s_i)$, corresponds to the set of jump destinations that a stack variable s_i can contain. The first element of the tuple, that is, n, stores the size of the stack in the different abstract states.

The abstract domain is the lattice $\langle AS, \pi_\top, \pi_\bot, \sqcup, \sqsubseteq \rangle$, where AS is the set of abstract states and $\pi_\top$ is the top of the lattice defined as the mapping $\pi_\top$ such that $\forall s \in \mathcal{S}, \pi_\top(s) = \mathcal{S}$. The bottom element of the lattice $\pi_\bot$ is the empty mapping. Now, to define $\sqcup$ and $\sqsubseteq$, we first define the function $img(\pi, s)$ as $\pi(s)$ if $s \in dom(\pi)$ and $\emptyset$, otherwise. Given two abstract states π_1 and π_2, we use $\pi = \pi_1 \sqcup \pi_2$ to denote that π is the least upper-bound defined as follows $\forall s \in dom(\pi_1) \cup dom(\pi_2), \pi(s) = img(\pi_1, s) \cup img(\pi_2, s)$ At this point, $\pi_1 \sqsubseteq \pi_2$ holds iff $dom(\pi_1) \subseteq dom(\pi_2)$ and $\forall s \in dom(\pi_1), \pi_1(s) \subseteq \pi_2(s)$.

Transfer function One of the ingredients of our analysis is a *transfer function* that models the effect of each EVM instruction on the abtract state for the different instructions. Given a stack state s of the form $\langle n, \sigma \rangle$, Figure 4 defines the updating function $\lambda(b_{pc}^{\delta,\alpha}, s)$ where b corresponds to the EVM instruction to be applied, pc corresponds to the program counter of the instruction and α and δ to the number of elements placed to and removed from the EVM stack when executing b, respectively. Given a map m we will use $m[x \mapsto y]$ to indicate the result of updating m by making $m(x) = y$ while m stays the same for

all locations different from x, and we will use $m\backslash[x]$ to refer to a partial mapping that stays the same for all locations different from x, and $m(x)$ is undefined.

By means of λ, now we can define the *transfer function* of our analysis.

Definition 3 (transfer function). *Given the set of abstract states AS and the set of EVM instructions Ins, the transfer function τ is defined as a mapping of the form*

$$\tau \ : \ Ins \times AS \mapsto AS$$

is defined as follows:

$$\tau(b, \pi) = \pi' \qquad where \quad \forall s \in dom(\pi), \pi'(s) = \lambda(b, \pi(s))$$

	$b^{\delta,\alpha}$	$\lambda(b, \langle n, \sigma \rangle)$	
(1)	PUSHx v	$\langle n+1, \sigma[s_n \mapsto \{v\}]\rangle$	*when* $v \in \mathcal{J}$
		$\langle n+1, \sigma\rangle$	*when* $v \notin \mathcal{J}$
(2)	DUPx	$\langle n+1, \sigma\rangle$	*when* $s_{n-x} \notin dom(\sigma)$
		$\langle n+1, \sigma[s_n \mapsto \sigma(s_{n-x})]\rangle$	*when* $s_{n-x} \in dom(\sigma)$
(3)	SWAPx	$\langle n, \sigma \rangle$	*when* $s_{n-1} \notin dom(\sigma) \wedge s_{n-x-1} \notin dom(\sigma)$
		$\langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1}), s_{n-1} \mapsto \sigma(s_{n-x-1})]\rangle$	*when* $s_{n-1} \in dom(\sigma) \wedge s_{n-x-1} \in dom(\sigma)$
		$\langle n, \sigma[s_{n-1} \mapsto \sigma(s_{n-x-1})]\backslash\sigma[s_{n-x-1}]\rangle$	*when* $s_{n-1} \notin dom(\sigma) \wedge s_{n-x-1} \in dom(\sigma)$
		$\langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1})]\backslash\sigma[s_{n-1}]\rangle$	*when* $s_{n-1} \in dom(\sigma) \wedge s_{n-x-1} \notin dom(\sigma)$
(4)	*otherwise*	$\langle n-\delta+\alpha, \sigma\backslash[s_{n-1}, \ldots, s_{n-\delta}]\rangle$	

Fig. 4. Updating function

Example 6. Given the following initial abstract state $\{\langle 8, \{\}\rangle \mapsto \{\langle 8, \{\}\rangle\}\}$, which corresponds to the initial stack state for executing block 941, the application of the transfer function τ to the block that starts at EVM instruction 941, produces the following results (between parenthesis we show the program point). To the right we show the application of the transfer function to block 123 with its initial abstract state $\{\langle 2, \{\}\rangle \mapsto \{\langle 2, \{\}\rangle\}\}$.

```
(941) JUMPDEST   {⟨8,{}⟩ ↦ {⟨8,{}⟩}}
(942) MOD        {⟨8,{}⟩ ↦ {⟨7,{}⟩}}
(943) ADD        {⟨8,{}⟩ ↦ {⟨6,{}⟩}}
(944) PUSH1 0A   {⟨8,{}⟩ ↦ {⟨7,{}⟩}}
(946) DUP2       {⟨8,{}⟩ ↦ {⟨8,{}⟩}}
(947) SWAP1      {⟨8,{}⟩ ↦ {⟨8,{}⟩}}
(948) SSTORE     {⟨8,{}⟩ ↦ {⟨6,{}⟩}}
(949) POP        {⟨8,{}⟩ ↦ {⟨5,{}⟩}}
(94A) PUSH2 0954 {⟨8,{}⟩ ↦ {⟨6,{s₅ ↦ 954}}⟩}}
(94D) PUSH1 0A   {⟨8,{}⟩ ↦ {⟨7,{s₆ ↦ 954}}⟩}}
(94F) SLOAD      {⟨8,{}⟩ ↦ {⟨7,{s₅ ↦ 954}}⟩}}
(950) PUSH2 064B {⟨8,{}⟩ ↦ {⟨8,{s₅ ↦ 954, s₇ ↦ 64B}⟩}}
(953) JUMP       {⟨8,{}⟩ ↦ {⟨7,{s₅ ↦ 954}}⟩}}
```

```
(123) JUMPDEST    {⟨2,{}⟩ ↦ {⟨2,{}⟩}}
(124) POP         {⟨2,{}⟩ ↦ {⟨1,{}⟩}}
(125) PUSH2 0142  {⟨2,{}⟩ ↦ {⟨2,{s₁ ↦ 142}⟩}}
(128) PUSH1 04    {⟨2,{}⟩ ↦ {⟨3,{s₁ ↦ 142}⟩}}
(12A) DUP1        {⟨2,{}⟩ ↦ {⟨4,{s₁ ↦ 142}⟩}}
(12B) CALLDATASIZE {⟨2,{}⟩ ↦ {⟨5,{s₁ ↦ 142}⟩}}
(12C) SUB         {⟨2,{}⟩ ↦ {⟨4,{s₁ ↦ 142}⟩}}
         ⋮
(13A) SWAP1       {⟨2,{}⟩ ↦ {⟨5,{s₁ ↦ 142}⟩}}
(13B) POP         {⟨2,{}⟩ ↦ {⟨4,{s₁ ↦ 142}⟩}}
(13C) POP         {⟨2,{}⟩ ↦ {⟨3,{s₁ ↦ 142}⟩}}
(13D) POP         {⟨2,{}⟩ ↦ {⟨2,{s₁ ↦ 142}⟩}}
(13E) PUSH2 064B  {⟨2,{}⟩ ↦ {⟨3,{s₁ ↦ 142, s₂ ↦ 64B}⟩}}
(141) JUMP        {⟨2,{}⟩ ↦ {⟨2,{s₁ ↦ 142}⟩}}
```

∎

Addresses equations system The next step consists in defining, by means of the transfer and the updating functions, a constraint equation system to represent all possible jumping addresses that could be valid for executing a *jump* instruction in the program.

Definition 4 (addresses equations system). *Given an EVM program P of the form $b_0, \ldots, b_p$, its addresses equation system, $\mathcal{E}(P)$ includes the following equations according to all EVM bytecode instruction $b_{pc} \in P$:*

b_{pc}	C_{pc}		
(1) JUMP	$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma \rangle))$		$\forall s \in dom(\mathcal{X}_{pc}), \langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$
(2) JUMPI	$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma \rangle))$		$\forall s \in dom(\mathcal{X}_{pc}), \langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$
	$\mathcal{X}_{pc+1} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma \rangle))$		$\forall s \in dom(\mathcal{X}_{pc}), \langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$
(3) $b_{pc} \notin End \wedge$ $b_{pc+size(b_{pc})} =$ JUMPDEST	$\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n, \sigma \rangle))$		$\forall s \in dom(\mathcal{X}_{pc}), \langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$
(4) $b_{pc} \notin End$	$\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq \tau(b_{pc}, \mathcal{X}_{pc})$		
(5) otherwise	$\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq \tau(b_{pc}, \mathcal{X}_{pc})$		

where $idmap(s)$ returns a map π such that $dom(\pi) = \{s\}$ and $\pi(s) = \{s\}$ and $size(b_{pc})$ returns the number of bytes of the instruction b_{pc}.

Observe that the addresses equations system will have one equation for all program points of the program. Concretely, variables of the form $\mathcal{X}_{pc}$ store the *jumping addresses* saved in the stack after executing b_{pc} for all possible entry stacks. This information will be used for computing all possible jump destinations when executing JUMP or JUMPI instructions. For computing the system, most instructions, cases (4) and (5), just apply the transfer function τ to compute the possible stack states of the subsequent instruction. Note that the expression $pc + size(b_{pc})$ at (3) just computes the position of the next instruction in the EVM program. Jumping instructions, points (1) and (2), compute the initial state of the invoked blocks, thus they produce a map with all possible input stack states that can reach one block. JUMP and JUMPI instructions produce, for each stack state, one equation by taking the element from the previous stack state $\mathcal{X}_{\sigma(s_{n-1})}$. JUMPI, point (2) produces an extra equation $\mathcal{X}_{pc+1}$ to capture the possibility of continuing to the next instruction instead of jumping to the destination address. Additionally, those instructions before JUMPDEST, point (3), produce initial states for the block that starts in the JUMPDEST. When the constraint equation system is solved, constraint variables over-approximate the jumping information for the program.

Example 7. As it can be seen if Figure 3, we can jump to block 64B from two different blocks, 941 and 123. The computation of the jump equations systems will produce the following equations for the entry program points of these two blocks:

$$\mathcal{X}_{941} \sqsupseteq \{\langle 8, \{\} \rangle \mapsto \{\langle 8, \{\} \rangle\}\} \qquad\qquad \mathcal{X}_{123} \sqsupseteq \{\langle 2, \{\} \rangle \mapsto \{\langle 2, \{\} \rangle\}\}$$
$$\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots$$
$$\mathcal{X}_{950} \sqsupseteq \{\langle 8, \{\} \rangle \mapsto \{\langle 8, \{s_5 \mapsto 954, s_8 \mapsto 64B\} \rangle\}\} \qquad \mathcal{X}_{318} \sqsupseteq \{\langle 2, \{\} \rangle \mapsto \{\langle 4, \{s_1 \mapsto 142, s_3 \mapsto 64B\} \rangle\}\}$$
$$\mathcal{X}_{64B}^{①} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \{\langle 7, \{s_5 \mapsto 954\} \rangle\}\} \qquad \mathcal{X}_{64B}^{②} \sqsupseteq \{\langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \{\langle 3, \{s_1 \mapsto 142\} \rangle\}\}$$

Observe that we have two different stack contents reaching the same program point, e.g. two equations for $\mathcal{X}_{64B}$ are produced by two different blocks, the JUMP at the end of block 941, identified by $\mathcal{X}_{64B}^{①}$, and the JUMP at the end of block 123, identified by $\mathcal{X}_{64B}^{②}$. Thus the equation that must hold for $\mathcal{X}_{64B}$ is produced by the application of the operation $\mathcal{X}_{64B}^{①} \sqcup \mathcal{X}_{64B}^{②}$, as follows:

$$\mathcal{X}_{64B} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \langle 7, \{s_5 \mapsto 954\} \rangle, \langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \langle 3, \{s_1 \mapsto 142\} \rangle\}$$

Note that the application of the transfer function τ for all instructions of block 64B applies function λ to all elements in the abstract state and updates the stack state accordingly

$$(\text{JUMPDEST})\ \mathcal{X}_{64B} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \langle 7, \{s_5 \mapsto 954\} \rangle,\ \langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \langle 3, \{s_1 \mapsto 142\} \rangle\}$$
$$(\text{PUSH1 00})\ \mathcal{X}_{64C} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \langle 8, \{s_5 \mapsto 954\} \rangle,\ \langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \langle 4, \{s_1 \mapsto 142\} \rangle\}$$
$$(\text{DUP1})\ \mathcal{X}_{64E} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \langle 9, \{s_5 \mapsto 954\} \rangle,\ \langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \langle 5, \{s_1 \mapsto 142\} \rangle\}$$
$$(\text{PUSH1 00})\ \mathcal{X}_{64F} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \langle 10, \{s_5 \mapsto 954\} \rangle,\ \langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \langle 6, \{s_1 \mapsto 142\} \rangle\}$$
$$(\text{SWAP1})\ \mathcal{X}_{651} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \langle 10, \{s_5 \mapsto 954\} \rangle,\ \langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \langle 6, \{s_1 \mapsto 142\} \rangle\}$$
$$(\text{POP})\ \mathcal{X}_{652} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\} \rangle \mapsto \langle 9, \{s_5 \mapsto 954\} \rangle,\ \langle 3, \{s_1 \mapsto 142\} \rangle \mapsto \langle 5, \{s_1 \mapsto 142\} \rangle\}$$

∎

Solving the addresses equations system of a program P can be done iteratively. A naïve algorithm consists in first creating one constraint variable $\mathcal{X}_0 \sqsupseteq \pi_\emptyset[\langle 0, \sigma_\emptyset \rangle \mapsto \{\langle 0, \sigma_\emptyset \rangle\}]$, where $\pi_\emptyset$ and $\sigma_\emptyset$ are empty mappings, and $\mathcal{X}_{pc} \sqsupseteq \pi_\perp$ for all $pc \in P, pc \neq 0$, and then iteratively refining the values of these variables as follows:

$\text{Ⓐ}\mathcal{X}_{64B} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\}\rangle \mapsto \langle 7, \{s_5 \mapsto 954\}\rangle, \qquad \langle 3, \{s_1 \mapsto 142\}\rangle \mapsto \langle 3, \{s_1 \mapsto 142\}\rangle \qquad \}$

$\vdots$

$\mathcal{X}_{652} \sqsupseteq \{\langle 7, \{s_5 \mapsto 954\}\rangle \mapsto \langle 9, \{s_5 \mapsto 954\}\rangle, \qquad \langle 3, \{s_1 \mapsto 142\}\rangle \mapsto \langle 5, \{s_1 \mapsto 142\}\rangle \qquad \}$
$\text{Ⓐ}\mathcal{X}_{653} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 9, \{s_5 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 5, \{s_1 \mapsto 142\}\rangle \qquad \}$

$\vdots$

$\mathcal{X}_{660} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 11, \{s_5 \mapsto 954, s_{10} \mapsto 6D0\}\rangle, \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 7, \{s_1 \mapsto 142, s_6 \mapsto 6D0\}\rangle \quad \}$
$\text{Ⓐ}\mathcal{X}_{661} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 9, \{s_5 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 5, \{s_1 \mapsto 142\}\rangle \qquad \}$
$\mathcal{X}_{6D0} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 9, \{s_5 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 5, \{s_1 \mapsto 142\}\rangle \qquad \}$

$\vdots$

$\mathcal{X}_{66D} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 13, \{s_5 \mapsto 954, s_{12} \mapsto 66F\}\rangle, \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 9, \{s_1 \mapsto 142, s_{10} \mapsto 66F\}\rangle \quad \}$
$\mathcal{X}_{66E} \sqsupseteq \{\langle 11, \{s_5 \mapsto 954\}\rangle \mapsto \langle 11, \{s_5 \mapsto 954\}\rangle, \qquad \langle 7, \{s_1 \mapsto 142\}\rangle \mapsto \langle 7, \{s_1 \mapsto 142\}\rangle \qquad \}$
$\text{Ⓐ}\mathcal{X}_{66F} \sqsupseteq \{\langle 11, \{s_5 \mapsto 954\}\rangle \mapsto \langle 11, \{s_5 \mapsto 954\}\rangle, \qquad \langle 7, \{s_1 \mapsto 142\}\rangle \mapsto \langle 7, \{s_1 \mapsto 142\}\rangle \qquad \}$

$\vdots$

$\mathcal{X}_{682} \sqsupseteq \{\langle 11, \{s_5 \mapsto 954\}\rangle \mapsto \langle 11, \{s_5 \mapsto 954, s_{10} \mapsto 6C3\}\rangle, \langle 7, \{s_1 \mapsto 142\}\rangle \mapsto \langle 7, \{s_1 \mapsto 142, s_6 \mapsto 6C3\}\rangle \quad \}$
$\text{Ⓐ}\mathcal{X}_{6C3} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 9, \{s_5 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 5, \{s_1 \mapsto 142\}\rangle \qquad \}$

$\vdots$

$\mathcal{X}_{6CF} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 10, \{s_5 \mapsto 954, s_9 \mapsto 653\}\rangle, \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 6, \{s_1 \mapsto 142, s_5 \mapsto 653\}\rangle \quad \}$
$\text{Ⓐ}\mathcal{X}_{683} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 9, \{s_5 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 5, \{s_1 \mapsto 142\}\rangle \qquad \}$

$\vdots$

$\mathcal{X}_{68F} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 13, \{s_5 \mapsto 954, s_{12} \mapsto 691\}\rangle, \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 9, \{s_1 \mapsto 142, s_8 \mapsto 691\}\rangle \quad \}$
$\text{Ⓐ}\mathcal{X}_{690} \sqsupseteq \{\langle 11, \{s_5 \mapsto 954\}\rangle \mapsto \langle 11, \{s_5 \mapsto 954\}, \qquad \langle 7, \{s_1 \mapsto 142\}\rangle \mapsto \langle 7, \{s_1 \mapsto 142\}\rangle \qquad \}$
$\text{Ⓐ}\mathcal{X}_{691} \sqsupseteq \{\langle 11, \{s_5 \mapsto 954\}\rangle \mapsto \langle 11, \{s_5 \mapsto 954\}, \qquad \langle 7, \{s_1 \mapsto 142\}\rangle \mapsto \langle 7, \{s_1 \mapsto 142\}\rangle \qquad \}$

$\vdots$

$\mathcal{X}_{6C2} \sqsupseteq \{\langle 11, \{s_5 \mapsto 954\}\rangle \mapsto \langle 10, \{s_5 \mapsto 954, s_9 \mapsto 6D1\}\rangle, \langle 7, \{s_1 \mapsto 142\}\rangle \mapsto \langle 6, \{s_1 \mapsto 142, s_5 \mapsto 6D1\}\rangle \quad \}$
$\text{Ⓐ}\mathcal{X}_{6D1} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 9, \{s_5 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 5, \{s_1 \mapsto 142\}\rangle \qquad \}$
$\mathcal{X}_{6D2} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 8, \{s_5 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 4, \{s_1 \mapsto 142\}\rangle \qquad \}$
$\mathcal{X}_{6D3} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 8, \{s_7 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 4, \{s_3 \mapsto 142\}\rangle \qquad \}$
$\mathcal{X}_{6D4} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 8, \{s_6 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 4, \{s_2 \mapsto 142\}\rangle \qquad \}$
$\mathcal{X}_{6D5} \sqsupseteq \{\langle 9, \{s_5 \mapsto 954\}\rangle \mapsto \langle 7, \{s_6 \mapsto 954\}\rangle, \qquad \langle 5, \{s_1 \mapsto 142\}\rangle \mapsto \langle 3, \{s_2 \mapsto 142\}\rangle \qquad \}$
$\text{Ⓑ}\mathcal{X}_{954} \sqsupseteq \{\langle 6, \{\}\rangle \mapsto \langle 6, \{\}\rangle \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \}$

$\vdots$

$\text{Ⓑ}\mathcal{X}_{142} \sqsupseteq \{\langle 2, \{\}\rangle \mapsto \langle 2, \{\}\rangle \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \}$

$\vdots$

Fig. 5. Jumps equations system of `__callback` function

1. substitute the current values of the constraint variables in the right-hand side of each constraint, and then evaluate the right-hand side if needed;
2. if each constraint $\mathcal{X} \sqsupseteq E$ holds, where E is the value of the evaluation of the right-hand side of the previous step, then the process finishes; otherwise
3. for each $\mathcal{X} \sqsupseteq E$ which does not hold, let E' be the current value of $\mathcal{X}$. Then update the current value of $\mathcal{X}$ to $E \sqcup E'$. Once all these updates are (iteratively) applied we repeat the process at step 1.

Termination is guaranteed since the abstract domain does not have infinitely ascending chains as the number of jump destinations and the stack size is finite.

Example 8. Figure 5 shows the equations produced by Definition 4 of the first and the last instruction of all blocks shown in Figure 3. The first instruction shown in the system is $\mathcal{X}_{64B}$, computed in Example 7. Observe that the application of τ stores the jumping addresses in the corresponding abstract states after `PUSH` instructions (see $\mathcal{X}_{660}$, $\mathcal{X}_{66D}$, $\mathcal{X}_{6CF}$, $\mathcal{X}_{68C}$, ...). Such addresses will be used to produce the equations at the `JUMP` or `JUMPI` instructions. In the case of `JUMP`, as the jump is unconditional, it only produces one equation, e.g. $\mathcal{X}_{66E}$ consumes address 66F to produce the input state of $\mathcal{X}_{66F}$, or $\mathcal{X}_{6C2}$ produces the input abstract state for $\mathcal{X}_{6D1}$. `JUMPI` instructions produce two different equations: (1) one equation which corresponds to the jumping address stored in the stack, e.g. equations $\mathcal{X}_{6D0}$ and $\mathcal{X}_{66F}$ produced by the jumps of the equations $\mathcal{X}_{660}$ and $\mathcal{X}_{66D}$ respectively; and (2) one equation which corresponds to the next

instruction, e.g. $\mathcal{X}_{661}$ and $\mathcal{X}_{66E}$ produced by $\mathcal{X}_{660}$ and $\mathcal{X}_{66D}$, respectively. Finally, another point to highlight occurs at equation $\mathcal{X}_{6D5}$: as we have two possible jumping addresses in the stack of and both can be used by the JUMP at the end of the block, we produce two inputs for the two possible jumping addresses, $\mathcal{X}_{954}$ and $\mathcal{X}_{142}$, for capturing the two possible branches from block 6D1 (see Figure 3). ∎

Theorem 1 (Soundness). *Let $P \equiv b_0, \ldots, b_p$ be a program, $\mathcal{X}_1, \ldots, \mathcal{X}_n$ the solution of the jumps equations system of P, and pc the program counter of a jump instruction. Then for any execution of P, there exists $s \in dom(\mathcal{X}_{pc})$ such that $\langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$ and $\sigma(s_{n-1})$ contains all jump addresses that instruction b_{pc} might jump to during the execution of P.*

Control Flow Graph. At this point, by means of the addresses equation system solution, we compute the control flow graph of the program. In order to simplify the notation, given a block B_i, we define the function $getId(i, \langle n, \sigma \rangle)$, which receives the block identifier i and an abstract stack $\langle n, \sigma \rangle$ and returns a unique identifier for the abstract stack $\langle n, \sigma \rangle \in dom(\mathcal{X}_i)$. Similarly, $getStack(i, id)$ returns the abstract state $\langle n, \sigma \rangle$ that corresponds to the identifier id of block B_i. Besides, we define the function $getSize(pc, id)$ that, given a program point $pc \in B_i$ and a unique identifier id for B_i, returns the value n' s.t. $\langle n, \sigma \rangle = getStack(i, id)$, and $\mathcal{X}_{pc}(\langle n, \sigma \rangle) = \langle n', \sigma' \rangle$.

Example 9. Given the equation:

$$\mathcal{X}_{64B} \sqsupseteq \{\underbrace{\langle 7, \{s_5 \mapsto 954\} \rangle}_{1} \mapsto \langle 7, \{s_5 \mapsto 954\} \rangle, \underbrace{\langle 3, \{s_1 \mapsto 142\} \rangle}_{2} \mapsto \langle 3, \{s_1 \mapsto 142\} \rangle \},$$

if we compute the functions $getId$ and $getSize$, we have that $getId(64B, \langle 7, \{s_5 \mapsto 954\} \rangle) = 1$ and $getId(64B, \langle 3, \{s_1 \mapsto 142\} \rangle) = 2$. Analogously, $getSize(64B, 1) = 7$ and $getSize(64B, 2) = 3$. ∎

Definition 5 (control flow graph). *Given an EVM program P, its blocks $B_i \equiv b_i \ldots b_j \in blocks(P)$ and its flow analysis results provided by a set of variables of the form $\mathcal{X}_{pc}$ for all $pc \in P$, we define the control flow graph of P as a directed graph $CFG = \langle V, E \rangle$ with a set of vertices*

$$V = \{B_{i:id} \mid B_i \in blocks(P) \wedge \langle n, \sigma \rangle \in dom(\mathcal{X}_i) \wedge id = getId(i, \langle n, \sigma \rangle)\}$$

and a set of edges $E = E_{jump} \cup E_{next}$ such that:

$$
\begin{aligned}
E_{jump} = \{B_{i:id} \to B_{d:id_2} \mid \ & b_j \in Jump \ \wedge \\
& \langle n, \sigma \rangle \in dom(\mathcal{X}_j) \wedge id = getId(i, \langle n, \sigma \rangle) \ \wedge \\
& \langle n', \sigma' \rangle \in \mathcal{X}_j(\langle n, \sigma \rangle) \ \wedge \ d = \sigma'(s_{n'-1}) \ \wedge \\
& \langle n'', \sigma'' \rangle = \lambda(b_j, \langle n', \sigma' \rangle) \wedge id_2 = getId(d, \langle n'', \sigma'' \rangle) \ \} \\
E_{next} = \{B_{i:id} \to B_{d:id_2} \mid \ & b_j \neq \text{JUMP} \ \wedge b_j \notin End \ \wedge \\
& \langle n, \sigma \rangle \in dom(\mathcal{X}_j) \wedge id = getId(i, \langle n, \sigma \rangle) \ \wedge \\
& \langle n', \sigma' \rangle \in \mathcal{X}_j(\langle n, \sigma \rangle) \wedge d = j + size(b_j) \ \wedge \\
& \langle n'', \sigma'' \rangle = \lambda(b_j, \langle n', \sigma' \rangle) \wedge id_2 = getId(d, \langle n'', \sigma'' \rangle)
\end{aligned}
$$

The first relevant point of the control flow graph (CFG) we produce is that, for producing the set of vertices V, we replicate each block for each different stack state that could be used for invoking it (gray nodes in Figure 3 are replicated in the CFG). Analogously, the different entry stack states are also used to produce different edges depending on its corresponding replicated blocks. Note that the definition distinguishes between two kinds of edges. (1) edges produced by JUMP or JUMPI instructions at the end of the blocks, whose destination is taken from the values stored in the stack states of the instruction before the jump with $d = \sigma'(s_{n'-1})$; and (2) edges produced by continuations to the next instruction, whose destination is computed with $d = j + size(b_j)$. In both kinds of edges, as we could have replicated blocks, we apply function λ and get the id of the resulting state to compute the id of the destination: $\langle n'', \sigma'' \rangle = \lambda(b_j, \langle n', \sigma' \rangle) \wedge id_2 = getId(d, \langle n'', \sigma'' \rangle)$.

Example 10. Considering the blocks shown in Figure 3 and the equations shown at Figure 5, the CFG of the program includes non-replicated nodes for those blocks that only receive one possible stack state (white nodes in Figure 3). However, the nodes that could be reached by two different stack states (gray nodes in Figure 3) will be replicated in the CFG:

$$
\begin{aligned}
V = \{ & B_{941}, B_{123}, B_{954}, B_{142}, B_{64B:1}, B_{653:1}, B_{661:1}, B_{66F:1}, B_{6C3:1}, B_{66E:1}, B_{690:1}, B_{683:1}, B_{691:1}, B_{6D0:1}, B_{6D1:1}, \\
& B_{64B:2}, B_{653:2}, B_{661:2}, B_{66F:2}, B_{6C3:2}, B_{66E:2}, B_{690:2}, B_{683:2}, B_{691:2}, B_{6D0:2}, B_{6D1:2} \}
\end{aligned}
$$

Analogously, our CFG replicates the edges according to the nodes replicated (solid and dashed edges in Figure 3):

$$E = \{ \; B_{941} \; \rightarrow \; B_{64B:1}, \; B_{64B:1} \; \rightarrow \; B_{653:1}, \; B_{653:1} \; \rightarrow \; B_{661:1}, \; B_{661:1} \; \rightarrow \; B_{66F:1}, \; B_{66F:1} \; \rightarrow \; B_{6C3:1},$$
$$B_{6C3:1} \; \rightarrow \; B_{653:1}, \; B_{66D:1} \; \rightarrow \; B_{66E:1}, \; B_{66F:1} \; \rightarrow \; B_{690:1}, \; B_{66F:1} \; \rightarrow \; B_{683:1}, \; B_{683:1} \; \rightarrow \; B_{691:1},$$
$$B_{691:1} \; \rightarrow \; B_{6D1:1}, \; B_{6D1:1} \; \rightarrow \; B_{954}, \; B_{123} \; \dashrightarrow \; B_{64B:2}, \; B_{64B:2} \; \dashrightarrow \; B_{653:2}, \; B_{653:2} \; \dashrightarrow \; B_{661:2},$$
$$B_{661:2} \; \dashrightarrow \; B_{66F:2}, \; B_{66F:2} \; \dashrightarrow \; B_{6C3:2}, \; B_{6C3:2} \; \dashrightarrow \; B_{653:2}, \; B_{66D:2} \; \dashrightarrow \; B_{66E:2}, \; B_{66F:2} \; \dashrightarrow \; B_{690:2},$$
$$B_{66F:2} \; \dashrightarrow \; B_{683:2}, \; B_{683:2} \; \dashrightarrow \; B_{691:2}, \; B_{691:2} \; \dashrightarrow \; B_{6D1:2}, \; B_{6D1:2} \; \dashrightarrow \; B_{142} \; \}$$

Note that, in Figure 3, we distinguish dashed and solid edges just to remark that we could have two possible execution paths, that is, if the call to `findWinner` comes from block B_{941}, it will return to block B_{954} and, if the execution comes from a public invocation, *i.e.* block B_{123}, it will return to block B_{142}. ∎

3 Proof for Theorem 1

The proof sketch follows the next steps:

1. We first define an EVM operational semantics that describes how EVM programs handle jump addresses on the stack.
2. Then we define an EVM collecting semantics for the operational semantics. Such collecting semantics gathers all transitions that can be produced by the execution of a program P;
3. We continue by defining the jumps-to property as a property of this collecting semantics; and
4. Then we prove a lemma that states that the least solution of the set of constraints generated as described in Definition 4 is a safe approximation of the EVM collecting semantics w.r.t. the jumps-to property.
5. Finally, we rewrite Theorem 1 in terms of the operational semantics and prove it.

Figure 6 shows the semantics of some instructions involved in the computation of the values stored in the stack for handling jumps. The state of the program S is a tuple $\langle pc, \langle n, \sigma \rangle \rangle$ where pc is the value of the program counter with the index of the next instruction to be executed, and $\langle n, \sigma \rangle$ is a *stack state* as defined in Section 2. Interesting rules are the ones that deal with jump destination addresses on the stack: Rule (4) adds a new address on the stack, and Rules (6) and (8-10) copy or exchange existing addresses on top of the stack, respectively. Rules (1) to (3) perform a jump in the program and therefore consume the address placed on top of the stack, plus an additional word in the case of JUMPI. If the instructions considered in this simplified semantics do not handle jump addresses, the corresponding rules just remove some values from the stack in the program state S (Rules (5), (7) and (11)). The remaining EVM instructions not explicitly considered in this simplified semantics are generically represented by Rule (12) with $b_{pc}^{\delta,\alpha}$, where δ is the number of items removed from stack when b_{pc} is executed, and α is the number of additional items placed on the stack. Complete executions are traces of the form $S_0 \Rightarrow S_1 \Rightarrow \ldots \Rightarrow S_n$ where $S_0 \equiv \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle$ is the initial state, σ is the empty mapping, and S_n corresponds to the last state. There are no infinite traces, as any transaction that executes EVM code has a finite gas limit and every instruction executed consumes some amount of gas. When the gas limit is exceeded, an out-of-gas exception occurs and the program halts immediately.

Definition 6 (EVM collecting semantics). *Given an EVM program P, the EVM collecting semantics operator $\mathcal{C}_P$ is defined as follows:*

$$\mathcal{C}_P(X) = \{ \langle S, S' \rangle \mid \langle \_, S \rangle \in X \wedge S \Rightarrow S' \}$$

The EVM semantics is defined as $\xi_P = \bigcup_{n>0} \mathcal{C}_P^n(X_0)$, where $X_0 \equiv \{ \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle \}$ is the initial configuration.

Definition 7 (jumps-to property). *Let P be an IR program, $\xi_P = \bigcup_{n>0} \mathcal{C}_P^n(X_0)$, and b an instruction at program point pc, then we say that $\xi_P \vDash_{pc} T$ if $T = \{ \langle n, \sigma \rangle \mid \langle S, S' \rangle \in \xi_P \wedge \langle n, \sigma \rangle \in S' \}$.*

The following lemma states that the least solution of the constraint equation system defined in Definition 2 is a safe approximation of ξ_P:

Lemma 1 (soundness). *Let $P \equiv b_0, \ldots, b_p$ be a program, pc a program point and $\mathcal{X}_0, \ldots, \mathcal{X}_p$ the least solution of the constraints equation system as defined in Section 2. The following holds:*

If $\xi_P \vDash_{pc} T$, then for all $\langle n, \sigma \rangle \in T$, exists $s \in dom(\mathcal{X}_{pc})$ such that $\langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$.

$$\frac{b_{pc} = \text{JUMP}}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle \sigma(s_{n-1}), \langle n-1, \sigma \backslash [s_{n-1}] \rangle \rangle}$$

(1)

$$\frac{b_{pc} = \text{JUMPI}}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle \sigma(s_{n-1}), \langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle \rangle}$$

(2)

$$\frac{b_{pc} = \text{JUMPI}}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle \rangle}$$

(3)

$$\frac{b_{pc} = \text{PUSH}x \ v, \text{v} \in \mathcal{J}}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma[s_n \mapsto \{v\}] \rangle \rangle}$$

(4)

$$\frac{b_{pc} = \text{PUSH}x \ v, v \notin \mathcal{J}}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma \rangle \rangle}$$

(5)

$$\frac{b_{pc} = \text{DUP}x, s_{n-x} \in dom(\sigma)}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma[s_n \mapsto \sigma(s_{n-x})] \rangle \rangle}$$

(6)

$$\frac{b_{pc} = \text{DUP}x, s_{n-x} \notin dom(\sigma)}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n+1, \sigma \rangle \rangle}$$

(7)

$$\frac{b_{pc} = \text{SWAP}x, s_{n-1} \in dom(\sigma), s_{n-x-1} \in dom(\sigma)}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1}), s_{n-1} \mapsto \sigma(s_{n-x-1})] \rangle \rangle}$$

(8)

$$\frac{b_{pc} = \text{SWAP}x, s_{n-1} \in dom(\sigma), s_{n-x-1} \notin dom(\sigma)}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma[s_{n-x-1} \mapsto \sigma(s_{n-1})] \backslash [s_{n-1}] \rangle \rangle}$$

(9)

$$\frac{b_{pc} = \text{SWAP}x, s_{n-1} \notin dom(\sigma), s_{n-x-1} \in dom(\sigma)}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma[s_{n-1} \mapsto \sigma(s_{n-x-1})] \backslash [s_{n-x-1}] \rangle \rangle}$$

(10)

$$\frac{b_{pc} = \text{SWAP}x, s_{n-1} \notin dom(\sigma), s_{n-x-1} \notin dom(\sigma)}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n, \sigma \backslash [s_{n-1}, s_{n-x-1}] \rangle \rangle}$$

(11)

$$\frac{b_{pc}^{\delta,\alpha} \in otherwise, b_{pc}^{\delta,\alpha} \notin End}{\langle pc, \langle n, \sigma \rangle \rangle \Rightarrow \langle pc+size(b_{pc}), \langle n-\delta+\alpha, \sigma \backslash [s_{n-1}, ..., s_{n-\delta}] \rangle \rangle}$$

(12)

Fig. 6. Simplified EVM semantics for handling jumps

Proof. We use $\mathcal{X}_{pc}^m$ to refer to the value obtained for $\mathcal{X}_{pc}$ after m iterations of the algorithm for solving the equation system depicted in Section 2. We say that $\mathcal{X}_{pc}$ *covers* $\langle n, \sigma \rangle$ in $\mathcal{C}_P^m(X_0)$ at program point pc when this lemma holds for the result of computing $\mathcal{C}_P^m(X_0)$. In order to prove this lemma, we can reason by induction on the value of m, the length of the traces $S_0 \Rightarrow^m S_m$ considered in $\mathcal{C}_P^m(X_0)$.

Case base: if $m = 0$, $S_0 = \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle$ and the Lemma trivially holds as $\langle 0, \sigma_\emptyset \rangle \in \mathcal{X}_0^0(\langle 0, \sigma_\emptyset \rangle)$.

Induction Hypothesis: we assume Lemma 1 holds for all traces of length $m \geq 0$.

Inductive Case: Let us consider traces of length $m + 1$, which are of the form $S_0 \Rightarrow^m S_m \Rightarrow S_{m+1}$. S_m is a program state of the form $S_m = \langle pc, \langle n, \sigma \rangle \rangle$. We can apply the induction hypothesis to S_m: there exists some $s \in dom(\mathcal{X}_{pc}^m)$ such that $\langle n, \sigma \rangle \in \mathcal{X}_{pc}^m(s)$. For extending the Lemma, we reason for all possible rules in the simplified EVM semantics (Fig. 6) we may apply from S_m to S_{m+1}:

– Rule (1): After executing a JUMP instruction S_{m+1} is of the form $\langle \sigma(s_{n-1}), \langle n-1, \sigma \backslash [s_{n-1}] \rangle \rangle$. In iteration $m + 1$, the following set of equations corresponding to b_{pc} is evaluated:

$$\mathcal{X}_{\sigma(s_{n-1})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n', \sigma' \rangle)) \quad \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma' \rangle \in \mathcal{X}_{pc}(s')$$

where $idmap(\lambda(b_{pc}, \langle n', \sigma' \rangle)) = \pi_\perp[\langle n'-1, \sigma' \backslash [s_{n-1}] \rangle \mapsto \{\langle n'-1, \sigma' \backslash [s_{n-1}] \rangle\}]$ (Case (4) in Fig. 4). The induction hypothesis guarantees that there exists some $s'' \in \mathcal{X}_{pc}^m$ such that $\langle n, \sigma \rangle \in \mathcal{X}_{pc}^m(s'')$,

where $S_m = \langle pc, \langle n, \sigma \rangle \rangle$. Therefore, at Iteration $m+1$, the following must hold:

$$\mathcal{X}^{m+1}_{\sigma(s_{n-1})} \sqsupseteq \pi_\perp[\langle n-1, \sigma \backslash [s_{n-1}] \rangle \mapsto \{\langle n-1, \sigma \backslash [s_{n-1}] \rangle\}]$$

so $\langle n-1, \sigma \backslash [s_{n-1}] \rangle \in \mathcal{X}^{m+1}_{\sigma(s_{n-1})}(\langle n-1, \sigma \backslash [s_{n-1}] \rangle)$ and thus Lemma 1 holds.

– Rules (2) and (3): After executing a `JUMPI` instruction, S_{m+1} is either $\langle \sigma(s_{n-1}), \langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle \rangle$ or $\langle pc + size(b_{pc}), \langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle \rangle$, respectively. In any of those cases the following sets of equations are evaluated:

$$\mathcal{X}_{\sigma(s_{n-2})} \sqsupseteq idmap(\lambda(\texttt{JUMPI}, \langle n', \sigma' \rangle)) \quad \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma' \rangle \in \mathcal{X}_{pc}(s')$$
$$\mathcal{X}_{pc+1} \sqsupseteq idmap(\lambda(\texttt{JUMPI}, \langle n', \sigma' \rangle)) \quad \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma' \rangle \in \mathcal{X}_{pc}(s')$$

where $idmap(\lambda(b_{pc}, \langle n', \sigma' \rangle)) = \pi_\perp[\langle n'-2, \sigma' \backslash [s_{n-1}, s_{n-2}] \rangle \mapsto \{\langle n'-2, \sigma' \backslash [s_{n-1}, s_{n-2}] \rangle\}]$ (Case (4) of the definition of the update function λ in Fig. 4). As in the previous case, the induction hypothesis guarantees that at Iteration m there exists $s'' \in \mathcal{X}^m_{pc}$ such that $\langle n, \sigma \rangle \in \mathcal{X}^m_{pc}(s'')$. Therefore, in Iteration $m+1$, the following must hold:

$$\mathcal{X}^{m+1}_{\sigma(s_{n-1})} \sqsupseteq \pi_\perp[\langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle \mapsto \{\langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle\}]$$
$$\mathcal{X}^{m+1}_{pc+1} \sqsupseteq \pi_\perp[\langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle \mapsto \{\langle n-2, \sigma \backslash [s_{n-1}, s_{n-2}] \rangle\}]$$

and thus Lemma 1 holds for these cases as well.

– Rules (4) - (12): We will first consider the case in which any of these rules corresponds to an `EVM` instruction followed by an instruction different from `JUMPDEST`. All rules are similar, as they all use the set of equations generated by Case (4) in Definition 4. We will see Rule (4) in detail.

After executing a `PUSH`x v instruction, S_{m+1} is $\langle pc + size(b_{pc}), \langle n+1, \sigma[s_n \mapsto \{v\}] \rangle \rangle$. We have to prove that exists some $s \in dom(\mathcal{X}_{pc+size(b_{pc})})$ such that $\langle n+1, \sigma[s_n \mapsto \{v\}] \rangle \in \mathcal{X}_{pc+size(b_{pc})}(s)$. The following set of equations is evaluated:

$$\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq \tau(\texttt{PUSH}x, \mathcal{X}_{pc}) \tag{1}$$

By Definition 3, $\tau(\texttt{PUSH}x, \mathcal{X}_{pc}) = \pi'$, where $\forall s' \in dom(\pi), \pi'(s') = \lambda(\texttt{PUSH}x, \mathcal{X}_{pc}(s'))$. By the case (1) of the definition of the update function λ, we have that:

$$\forall \langle n'', \sigma'' \rangle \in dom(\mathcal{X}_{pc}), \pi'(\langle n'', \sigma'' \rangle) = \langle n''+1, \sigma''[s_n \mapsto \{v\}] \rangle \tag{2}$$

By the induction hypothesis, at Iteration m there exists some $s \in dom(\mathcal{X}^m_{pc})$ such that $\langle n, \sigma \rangle \in \mathcal{X}^m_{pc}(s)$. Therefore, by 1 and 2, at Iteration $m+1$ we have that the following holds:

$$s \in dom(\mathcal{X}^{m+1}_{pc+size(b_{pc})}) \text{ and } \langle n+1, \sigma[s_n \mapsto \{v\}] \rangle \in \mathcal{X}_{pc+size(b_{pc})}(s)$$

and thus Lemma 1 holds for Rule (4).

– Rules (4) - (12), followed by a `JUMPDEST` instruction. After executing any of these instructions, S_{m+1} is $\langle pc + size(b_{pc}), \langle n''', \sigma''' \rangle \rangle$, where $\langle n''', \sigma''' \rangle$ is obtained according to the rule from Figure 6. We have to prove that exists some $s \in dom(\mathcal{X}_{pc+size(b_{pc})})$ such that $\langle n''', \sigma''' \rangle \in \mathcal{X}_{pc+size(b_{pc})}(s)$. The following set of equations is evaluated:

$$\mathcal{X}_{pc+size(b_{pc})} \sqsupseteq idmap(\lambda(b_{pc}, \langle n', \sigma' \rangle)) \quad \text{for all } s' \in dom(\mathcal{X}_{pc}), \langle n', \sigma' \rangle \in \mathcal{X}_{pc}(s') \tag{3}$$

where $idmap(\lambda(b_{pc}, \langle n', \sigma' \rangle)) = \pi_\perp[\langle n'', \sigma'' \rangle \mapsto \{\langle n'', \sigma'' \rangle\}]$, where n'' and $\sigma`$ are obtained according to the cases of the updating function detailed in Figure 4. We can see that $\{\langle n'', \sigma'' \rangle\}]$ match the modification made to the state S_{m+1} by the corresponding rule of the semantics. Therefore, at Iteration there exists an $s = \{\langle n'', \sigma'' \rangle\}]$ such that $\{\langle n'', \sigma'' \rangle\}] \in \mathcal{X}^{m+1}_{pc+size(b_{pc})}$, and Lemma 1 also holds.

When the algorithm stops Lemma 1 holds, as for any pc $\mathcal{X}^{m+1}_{pc} \sqsupseteq \mathcal{X}^m_{pc}$ for each iteration of the algorithm for solving the equations system of Section 2.

Now we rewrite Theorem 1 in terms of the operational semantics of Figure 6. This rewriting actually is stronger than Theorem 1, as guarantees the correctness of the stack states obtained from the jumps equations system at any step of the execution.

Theorem 2 (Soundness). *Let $P \equiv b_0, \ldots, b_p$ be a program, $S_0 = \langle 0, \langle 0, \sigma_\emptyset \rangle \rangle$ the initial program state, and $\mathcal{X}_1, \ldots, \mathcal{X}_n$ the solution of the jumps equations system of P. Then for any trace $S_0 \rightsquigarrow^* S_m$, where $S_m = \langle pc, \langle n, \sigma \rangle \rangle$, there exists $s \in dom(\mathcal{X}_{pc})$ such that $\langle n, \sigma \rangle \in \mathcal{X}_{pc}(s)$.*

Proof. Straightforward from Lemma 1, as the `EVM` collecting semantics takes into account all possible traces of the operational semantics.

References

1. The EthereumPot contract, 2017. https://etherscan.io/address/0x5a13caa82851342e14cd2ad0257707cddb8a31b7.

2. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.

3. Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A Scalable Security Analysis Framework for Smart Contracts, 2018. arXiv:1809.03981.

4. Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1176–1186. IEEE / ACM, 2019.

5. Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27, 2018.

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