Toward a Sound Construction of EVM Bytecode Control-Flow Graphs

UNIVERSITÀ DI PARMA

Vincenzo Arceri
University of Parma
Italy

Luca Negrini
University of Venice
Italy

Greta Dolcetti
University of Venice
Italy

Saverio Mattia Merenda
University of Parma
Italy

Luca Olivieri
University of Venice
Italy

Enea Zaffanella
University of Parma
Italy

Formal Techniques for Java-like Programs (FTfJP 2024)
Vienna, 20 Sep 2024

Introduction to Ethereum and Smart Contracts

Ethereum

- Public permissionless blockchain
- Supporting smart contracts

Introduction to Ethereum and Smart Contracts

Ethereum

- Public permissionless blockchain
- Supporting smart contracts

Smart Contracts

- Immutable programs stored on the blockchain
- Critical to ensure they are bug-free to avoid irrevocable issues

Introduction to Ethereum and Smart Contracts

Ethereum

- Public permissionless blockchain
- Supporting smart contracts

Smart Contracts

- Immutable programs stored on the blockchain
- Critical to ensure they are bug-free to avoid irrevocable issues

EVM Bytecode

- A low-level, stack-based language, executed in a virtual machine
- Instructions manipulate the stack directly
- Supports arithmetic, logical, and execution control flow operations

EVM Bytecode Overview

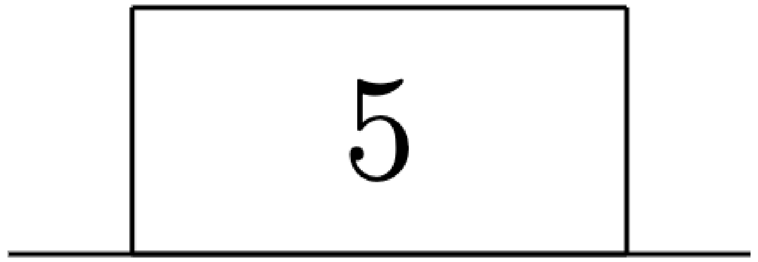
Example

```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
```

EVM Bytecode Overview

Example

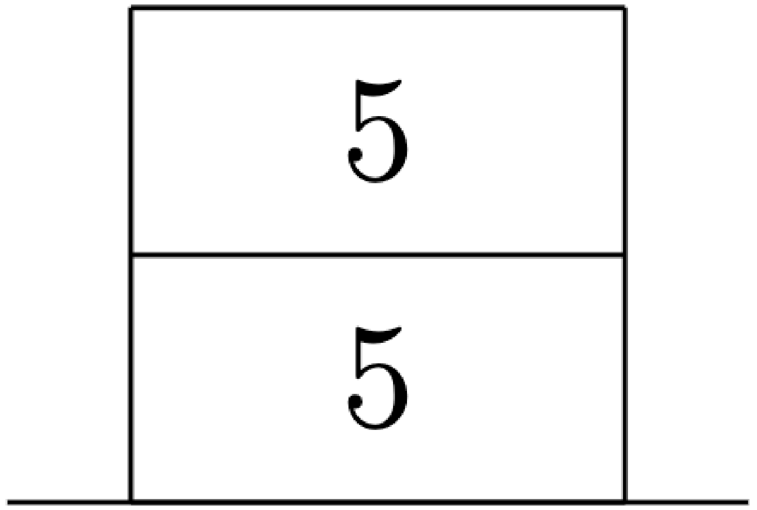
```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
```



EVM Bytecode Overview

Example

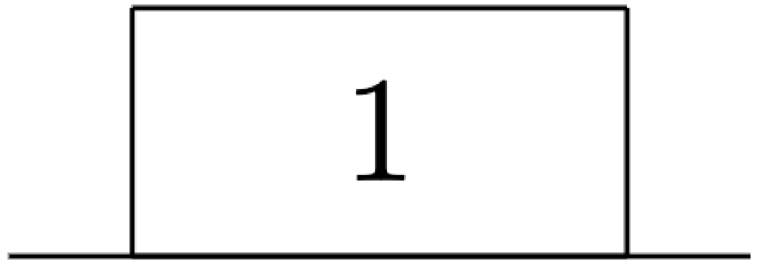
```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
```



EVM Bytecode Overview

Example

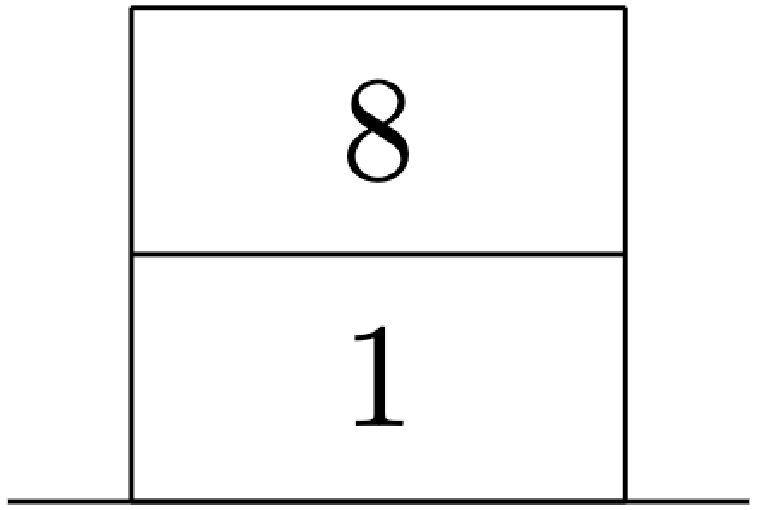
```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
```



EVM Bytecode Overview

Example

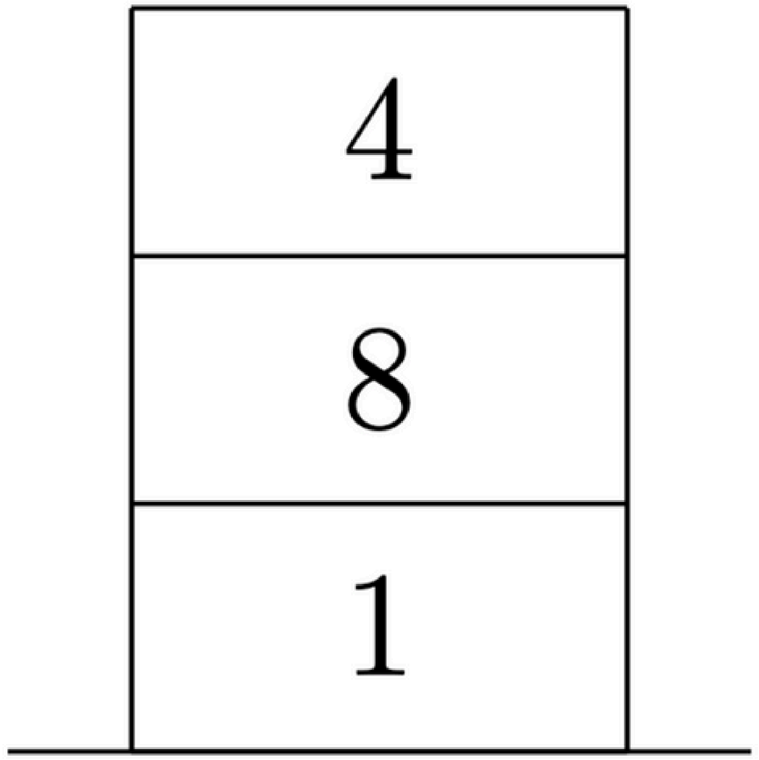
```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
```



EVM Bytecode Overview

Example

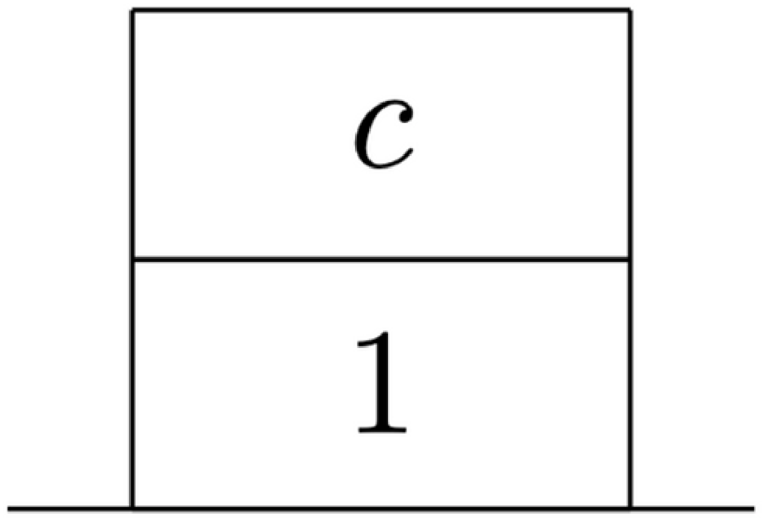
```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
```



EVM Bytecode Overview

Example

```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
```



Challenges in Building CFGs for EVM Bytecode

Control flow instructions

- Execution is sequential: begins with the first opcode and proceeds sequentially
- JUMP and JUMPI alter the execution flow
- JUMPDEST marks valid jump destinations: computed at runtime

Challenges in Building CFGs for EVM Bytecode

Control flow instructions

- Execution is sequential: begins with the first opcode and proceeds sequentially
- JUMP and JUMPI alter the execution flow
- JUMPDEST marks valid jump destinations: computed at runtime

Dynamic jumps

- Jump targets are not always explicitly defined
- We can identify two types of jumps: *pushed jumps* and *orphan jumps*

Dynamic jumps create ***complex situations*** when identifying valid jump targets.

Pushed jumps

Definition

- Jump target is determined by a value pushed onto the stack
- Jump destination is known at compile-time

```
[00]  PUSH1  0x01
[02]  PUSH1  0x02
[04]  JUMP
```

Example of pushed jump.

Orphan jumps

Definition

- Jumps whose targets are not immediately obvious from the code
- Jump target is not known at compile-time and is determined during execution

```
[00] PUSH1  0x01
[02] PUSH1  0x02
[04] ADD
[05] JUMP
```

Example of orphan jump.

Sound CFGs for EVM Bytecode (1/2)

Static Analysis

- Used to identify potential issues without executing the code
- Essential for early detection of bugs and vulnerabilities

Sound CFGs for EVM Bytecode (1/2)

Static Analysis

- Used to identify potential issues without executing the code
- Essential for early detection of bugs and vulnerabilities

Control-flow Graphs (CFGs)

- Data structure representing all paths that may be traversed during program execution
- Nodes represent basic blocks of instructions; edges represent control flow
- Essential for effective static analysis

Building a Sound CFG allows us ***to perform*** a sound Static Analysis.

Sound CFGs for EVM Bytecode (2/2)

Challenges

- Jump destination targets aren't always clear from syntax alone
  - Pushed jumps: targets are clear from the syntax
  - Orphan jumps: targets are computed at runtime

```
[00]  PUSH1  0x01
[02]  PUSH1  0x02
[04]  JUMP
```

Example of pushed jump.

```
[00]  PUSH1  0x01
[02]  PUSH1  0x02
[04]  ADD
[05]  JUMP
```

Example of orphan jump.

Sound CFGs for EVM Bytecode (2/2)

Challenges

- Jump destination targets aren't always clear from syntax alone
  - Pushed jumps: targets are clear from the syntax
  - Orphan jumps: targets are computed at runtime

Goal

- Build a sound CFG for EVM Bytecode
- Over-approximate jump destinations for each jump node

```
[00] PUSH1 0x01
[02] PUSH1 0x02
[04] JUMP
```

Example of pushed jump.

```
[00] PUSH1 0x01
[02] PUSH1 0x02
[04] ADD
[05] JUMP
```

Example of orphan jump.

Contribution of the Paper

Novel approach

- Abstract interpretation-based method to construct CFGs for EVM bytecode
- Abstract domains to evaluate instructions to over-approximate stacks reaching each node

* github.com/lisa-analyzer/evm-lisa

Contribution of the Paper

Novel approach

- Abstract interpretation-based method to construct CFGs for EVM bytecode
- Abstract domains to evaluate instructions to over-approximate stacks reaching each node

Iterative algorithm

- Iteratively builds the CFG until a stable, sound graph is achieved
- Handles *pushed jumps* and *orphan jumps* effectively

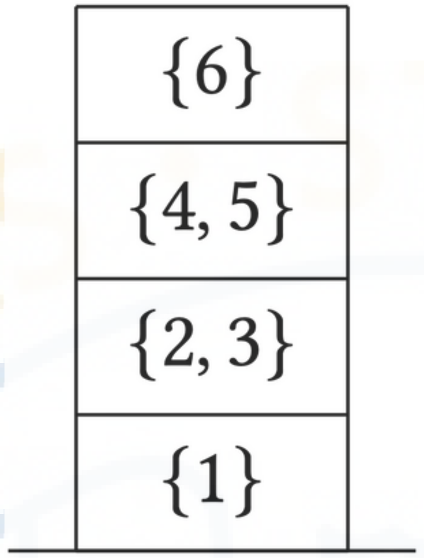
In this paper we present ***EVMLiSA***,* a ***static analyzer*** for EVM bytecode that ***demonstrates*** practical application and effectiveness of the proposed method.

* github.com/lisa-analyzer/evm-lisa

Abstract domain of k-sets of integers (1/2)

Definition

- $\mathbb{Z}^\sharp_k \triangleq \langle \wp_{\leq k}(\mathbb{Z}) \cup \{\top_{\mathbb{Z}}, \top_{\overline{\mathbb{Z}}}, \top_{\mathbb{Z}^\sharp_k}\}, \sqsubseteq_{\mathbb{Z}^\sharp_k}, \sqcup_{\mathbb{Z}^\sharp_k}, \sqcap_{\mathbb{Z}^\sharp_k}, \top_{\mathbb{Z}^\sharp_k}, \varnothing \rangle$
- Where $\wp_{\leq k}(\mathbb{Z})$ are sets of integers having cardinality at most k
- Values of $\mathbb{Z}^\sharp_k$ are the elements of abstract stacks

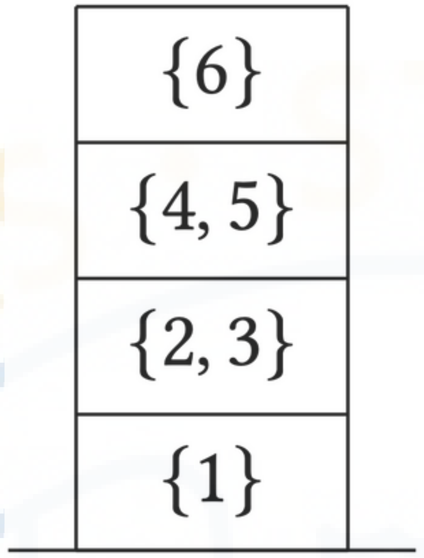


Example of abstract stack with k = 2.

Abstract domain of k-sets of integers (1/2)

Definition

- $\mathbb{Z}_k^\sharp \triangleq \langle \wp_{\leq k}(\mathbb{Z}) \cup \{\top_{\mathbb{Z}}, \top_{\overline{\mathbb{Z}}}, \top_{\mathbb{Z}_k^\sharp}\}, \sqsubseteq_{\mathbb{Z}_k^\sharp}, \sqcup_{\mathbb{Z}_k^\sharp}, \sqcap_{\mathbb{Z}_k^\sharp}, \top_{\mathbb{Z}_k^\sharp}, \varnothing \rangle$
- Where $\wp_{\leq k}(\mathbb{Z})$ are sets of integers having cardinality at most k
- Values of $\mathbb{Z}_k^\sharp$ are the elements of abstract stacks



Example of abstract stack with k = 2.

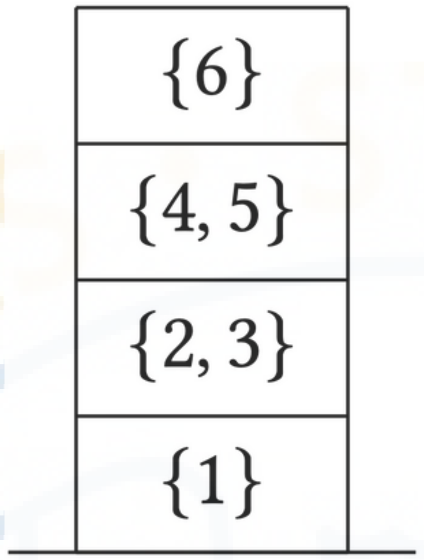
Special elements

- $\top_{\mathbb{Z}_k^\sharp}$ denotes an unknown set of integers
- $\top_{\mathbb{Z}}$ denotes an unknown set of integers that may correspond to valid jump destinations
- $\top_{\overline{\mathbb{Z}}}$ denotes an unknown set of integers that don't correspond to valid jump destinations

Abstract domain of k-sets of integers (1/2)

Definition

- $\mathbb{Z}_k^\sharp \triangleq \langle \wp_{\leq k}(\mathbb{Z}) \cup \{\top_{\mathbb{Z}}, \top_{\overline{\mathbb{Z}}}, \top_{\mathbb{Z}_k^\sharp}\}, \sqsubseteq_{\mathbb{Z}_k^\sharp}, \sqcup_{\mathbb{Z}_k^\sharp}, \sqcap_{\mathbb{Z}_k^\sharp}, \top_{\mathbb{Z}_k^\sharp}, \varnothing \rangle$
- Where $\wp_{\leq k}(\mathbb{Z})$ are sets of integers having cardinality at most k
- Values of $\mathbb{Z}_k^\sharp$ are the elements of abstract stacks



Example of abstract stack with k = 2.

Special elements

- $\top_{\mathbb{Z}_k^\sharp}$ denotes an unknown set of integers
- $\top_{\mathbb{Z}}$ denotes an unknown set of integers that may correspond to valid jump destinations
- $\top_{\overline{\mathbb{Z}}}$ denotes an unknown set of integers that don’t correspond to valid jump destinations

Why did we choose to differentiate $\top_{\mathbb{Z}}$ and $\top_{\overline{\mathbb{Z}}}$?

- Unusual and tricky sequences of opcodes may arise
- EVM bytecode is generated by high-level languages

Abstract domain of k-sets of integers (2/2)

Example

```
[00]  TIMESTAMP
[01]  JUMP
```

- TIMESTAMP pushes the current block's timestamp onto the stack
- The JUMP opcode uses the top stack value to jump in the code

Abstract domain of k-sets of integers (2/2)

Example

```
[00]  TIMESTAMP
[01]  JUMP
```

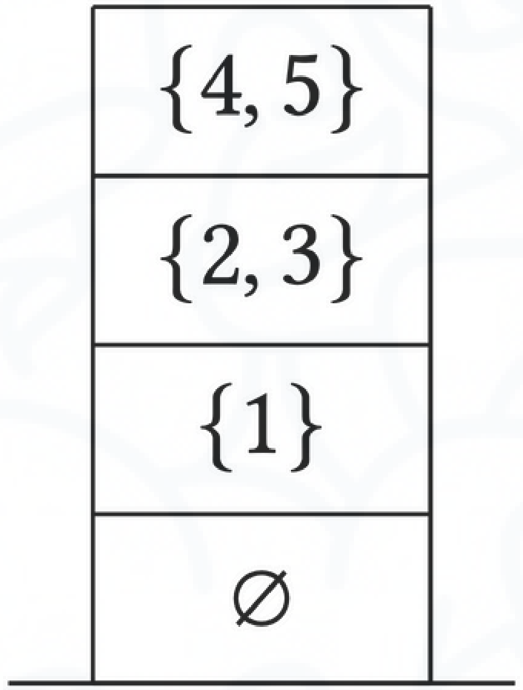
- ◦ TIMESTAMP pushes the current block's timestamp onto the stack
- ◦ The JUMP opcode uses the top stack value to jump in the code
- The semantics of TIMESTAMP returns $\top_{\mathbb{Z}}$

We'll use this ***to assess*** if destination targets of a jump have been resolved or not.

Abstract domain of *h*-sized stack (1/3)

Abstract elements

- $\mathcal{S}_{\mathbb{Z}_k^\sharp, h} \triangleq \{[s_0, s_1, \ldots, s_{h-1}] \mid \forall i \in [0, h-1] \,.\, s_i \in \mathbb{Z}_k^\sharp\}$
- Represents stacks with exactly h elements, where s_{h-1} is the top of the stack
- Stacks with fewer than *h* elements are modeled as stacks with exactly *h*, filling gaps with $\emptyset$



Example of abstract stack with h = 4.

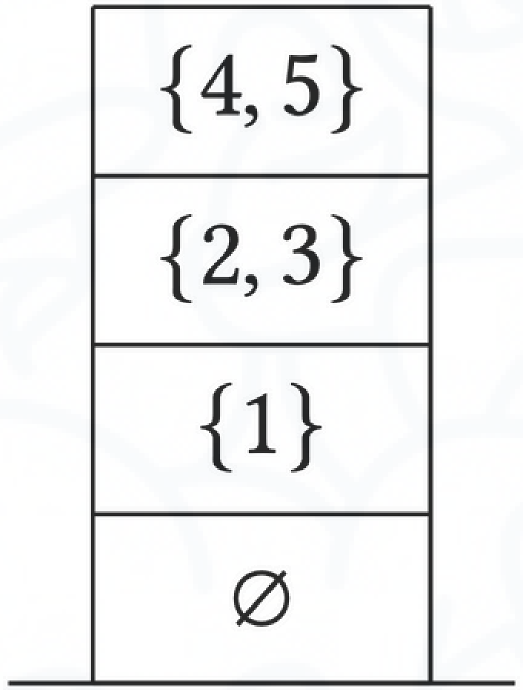
Abstract domain of *h*-sized stack (1/3)

Abstract elements

- $\mathcal{S}_{\mathbb{Z}^\sharp_k,h} \triangleq \{[s_0, s_1, \ldots, s_{h-1}] \mid \forall i \in [0, h-1] \ . \ s_i \in \mathbb{Z}^\sharp_k\}$
- Represents stacks with exactly h elements, where s_{h-1} is the top of the stack
- Stacks with fewer than *h* elements are modeled as stacks with exactly *h*, filling gaps with $\varnothing$

Definition

- $\mathrm{St}^\#_{k,h} \triangleq \langle \mathcal{S}_{\mathbb{Z}^\sharp_k,h} \cup \{\bot_{\mathrm{St}^\#_{k,h}}\}, \sqcup_{\mathrm{St}^\#_{k,h}}, \sqcap_{\mathrm{St}^\#_{k,h}}, \top_{\mathrm{St}^\#_{k,h}}, \bot_{\mathrm{St}^\#_{k,h}} \rangle$
- Lattice operators are element-wise applications of the ones $\mathbb{Z}^\#_k$
- $\top_{\mathrm{St}^\#_{k,h}}, \bot_{\mathrm{St}^\#_{k,h}}$ represents *top* and *bottom* special element, respectively

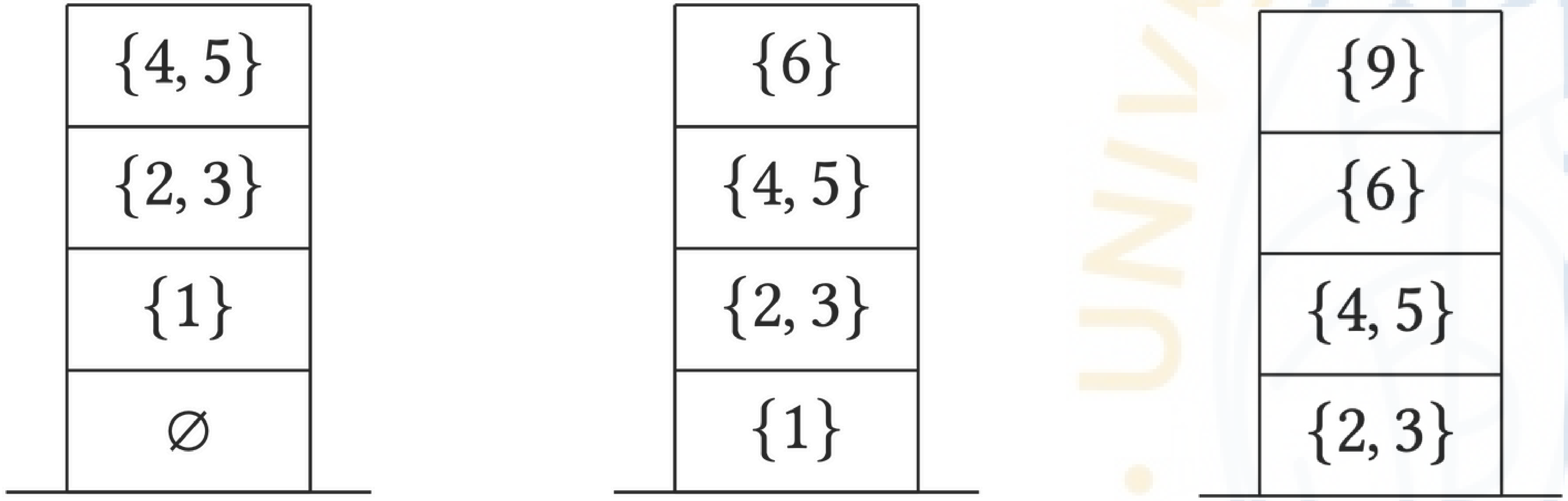


Example of abstract stack with h = 4.

Abstract domain of h-sized stack (*Push function*)

Definition

- The abstract function $\mathsf{push} : \mathrm{St}^{\#}_{k,h} \times \mathbb{Z}^{\sharp}_{k} \rightarrow \mathrm{St}^{\#}_{k,h}$ pushes a $\mathbb{Z}^{\#}_{k}$ into $S_{\mathbb{Z}^{\#}_{k},h}$
- Fig. **a** shows an abstract stack of $\mathcal{S}_{\mathbb{Z}^{\sharp}_{2},4}$ with size 3
- Fig. **b** and Fig. **c** show the result of abstractly executing *PUSH*, starting from the abstract stack in Fig. **a** and Fig. **b**, respectively

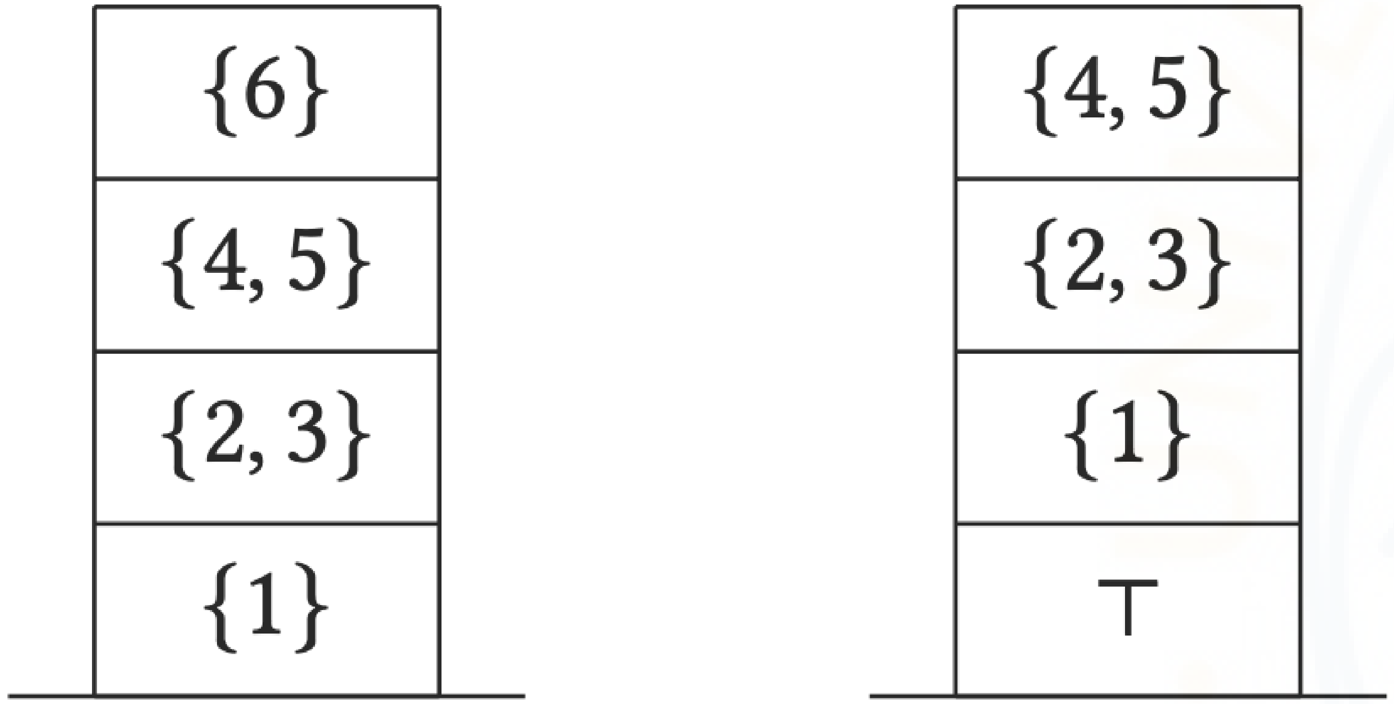


(a) **(b)** **(c)**

Abstract domain of h-sized stack (*Pop function*)

Definition

- The abstract function $\mathsf{pop} : \mathrm{St}^{\#}_{k,h} \to \mathrm{St}^{\#}_{k,h}$ pops an element from $S_{\mathbb{Z}^{\#}_k,h}$
- Shifts elements up and fills the bottom with $\varnothing$ if $s_0 = \varnothing$, or with $\top_{\mathbb{Z}^{\sharp}_k}$ if $s_0 \neq \varnothing$
- Fig. b is obtained by popping an element from the abstract stack of Fig. a



(a) **(b)**

Static Analysis Algorithm

Definition

- The described approach defines a static analysis that over-approximates concrete stacks for each node in the CFG

Static Analysis Algorithm

Definition

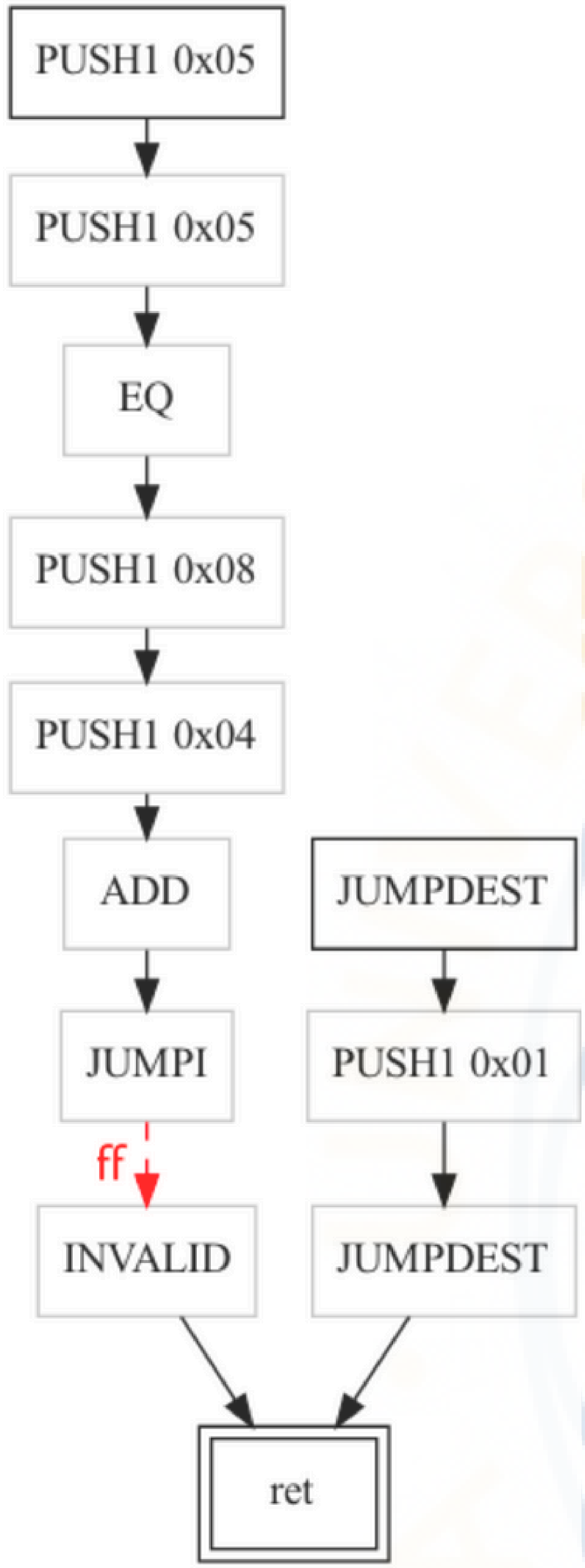
- The described approach defines a static analysis that over-approximates concrete stacks for each node in the CFG

Algorithm

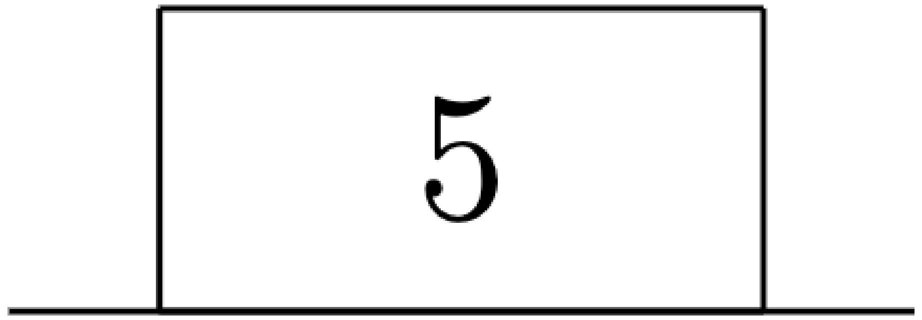
1. *Create* an initial, partial CFG with only sequential edges
2. *Run* static analysis to compute the abstract stack for each node
3. *Use* the analysis to try to resolve jump destinations
4. *Re-run* the analysis each time a new edge is added (back to point 2)
5. *Stop* when no more edges can be added to the CFG

Jump Resolution

```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
[0a] JUMPI // orphan jump
[0b] INVALID
[0c] JUMPDEST
[0d] PUSH1 0x01
[0f] JUMPDEST
```

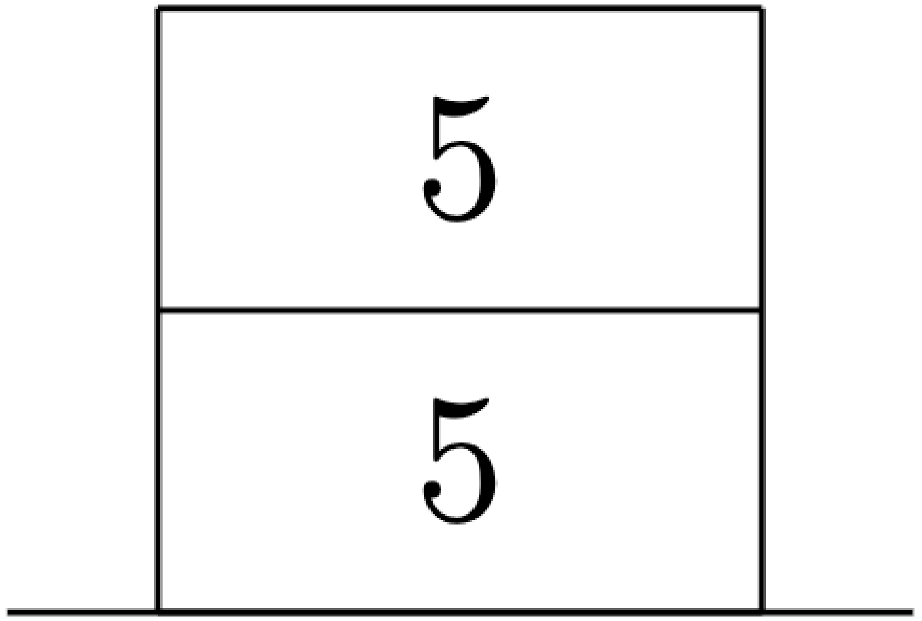


Jump Resolution



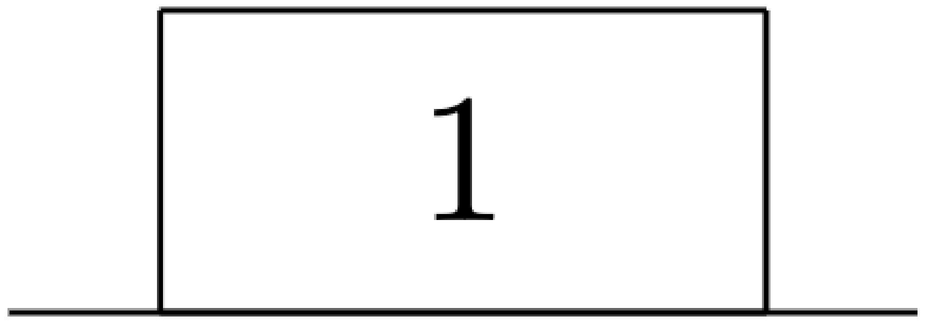

```
[00]  PUSH1  0x05
[02]  PUSH1  0x05
[04]  EQ
[05]  PUSH1  0x08
[07]  PUSH1  0x04
[09]  ADD
[0a]  JUMPI  // orphan jump
[0b]  INVALID
[0c]  JUMPDEST
[0d]  PUSH1  0x01
[0f]  JUMPDEST
```

Jump Resolution



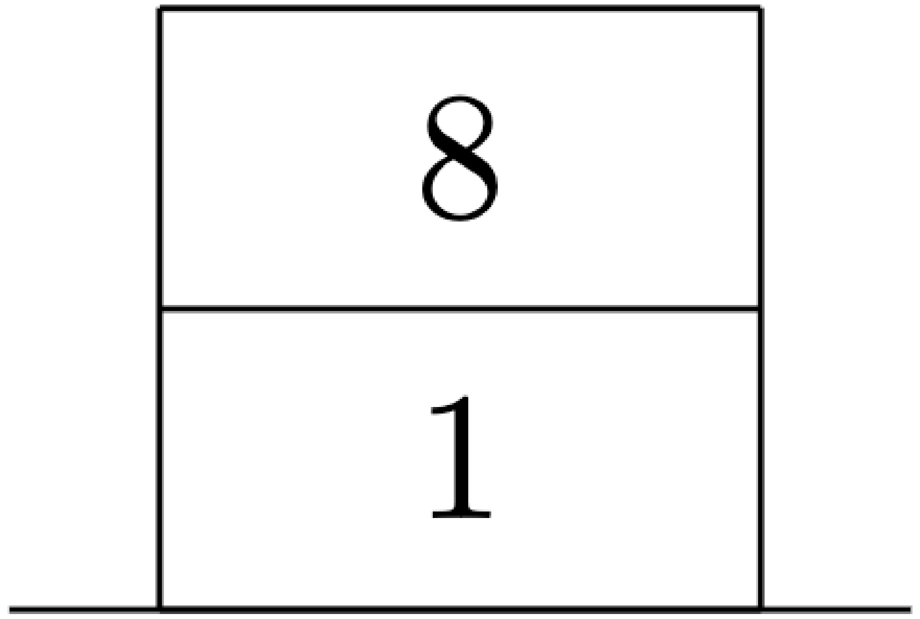

```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
[0a] JUMPI // orphan jump
[0b] INVALID
[0c] JUMPDEST
[0d] PUSH1 0x01
[0f] JUMPDEST
```

Jump Resolution



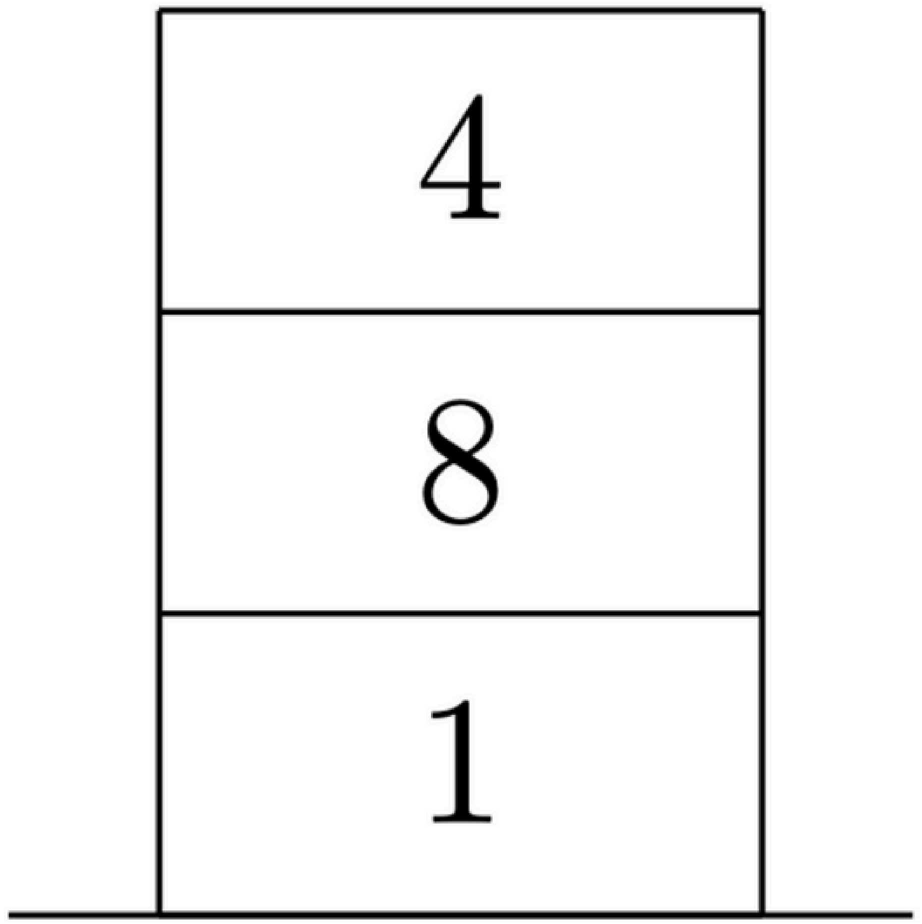

```
[00]  PUSH1  0x05
[02]  PUSH1  0x05
[04]  EQ
[05]  PUSH1  0x08
[07]  PUSH1  0x04
[09]  ADD
[0a]  JUMPI  // orphan jump
[0b]  INVALID
[0c]  JUMPDEST
[0d]  PUSH1  0x01
[0f]  JUMPDEST
```

Jump Resolution



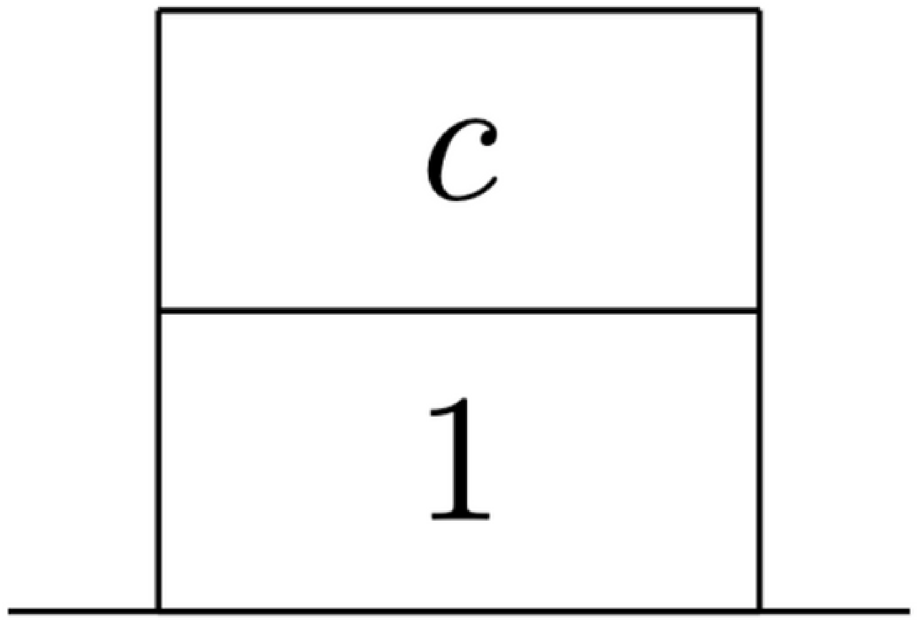

```
[00] PUSH1 0x05
[02] PUSH1 0x05
[04] EQ
[05] PUSH1 0x08
[07] PUSH1 0x04
[09] ADD
[0a] JUMPI // orphan jump
[0b] INVALID
[0c] JUMPDEST
[0d] PUSH1 0x01
[0f] JUMPDEST
```

Jump Resolution



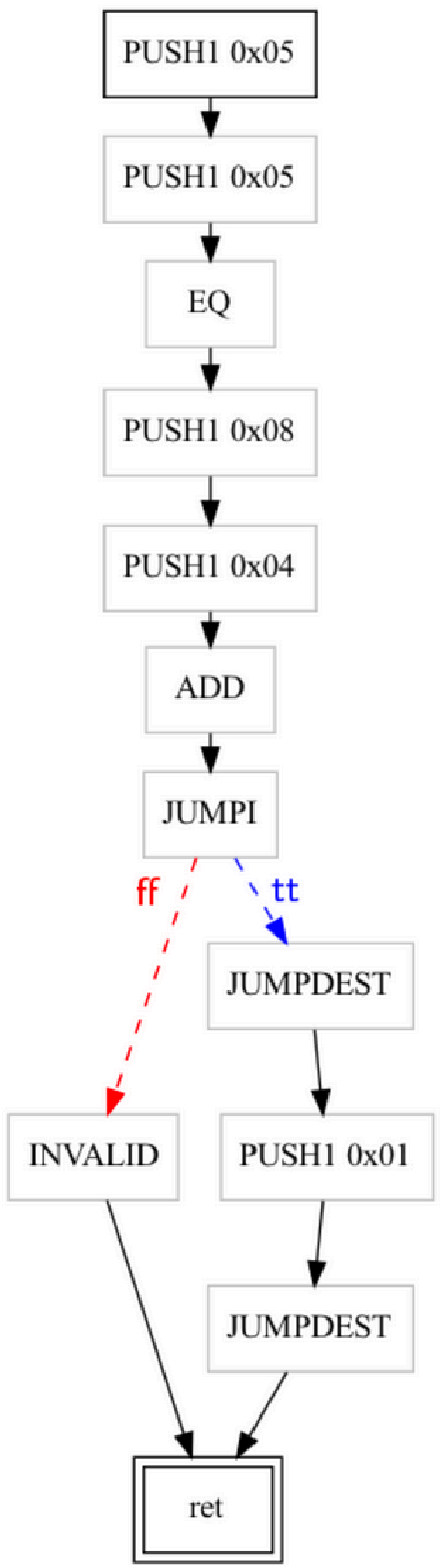

```
[00]  PUSH1  0x05
[02]  PUSH1  0x05
[04]  EQ
[05]  PUSH1  0x08
[07]  PUSH1  0x04
[09]  ADD
[0a]  JUMPI  // orphan jump
[0b]  INVALID
[0c]  JUMPDEST
[0d]  PUSH1  0x01
[0f]  JUMPDEST
```

Jump Resolution




```
[00]  PUSH1  0x05
[02]  PUSH1  0x05
[04]  EQ
[05]  PUSH1  0x08
[07]  PUSH1  0x04
[09]  ADD
[0a]  JUMPI  // orphan jump
[0b]  INVALID
[0c]  JUMPDEST
[0d]  PUSH1  0x01
[0f]  JUMPDEST
```

Jump Resolution

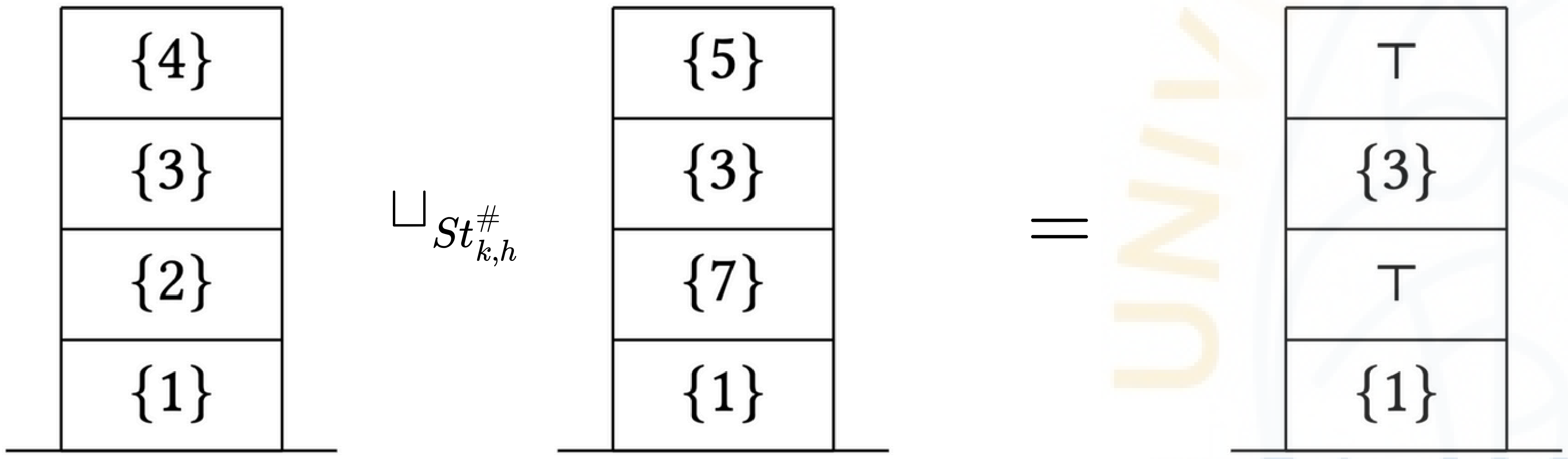



```
[00]  PUSH1  0x05
[02]  PUSH1  0x05
[04]  EQ
[05]  PUSH1  0x08
[07]  PUSH1  0x04
[09]  ADD
[0a]  JUMPI  // orphan jump
[0b]  INVALID
[0c]  JUMPDEST
[0d]  PUSH1  0x01
[0f]  JUMPDEST
```

From Abstract Stacks to Sets of Abstract Stacks (1/2)

Problem

- While loops occur, the analysis merges abstract stacks into one using the least upper bound (lub) operator
- May lose precision when merging elements via lub of the $\mathbb{Z}_k^{\#}$ domain if k is exceeded
- The result would be $\top_{\mathbb{Z}_k^{\sharp}}$, losing all information

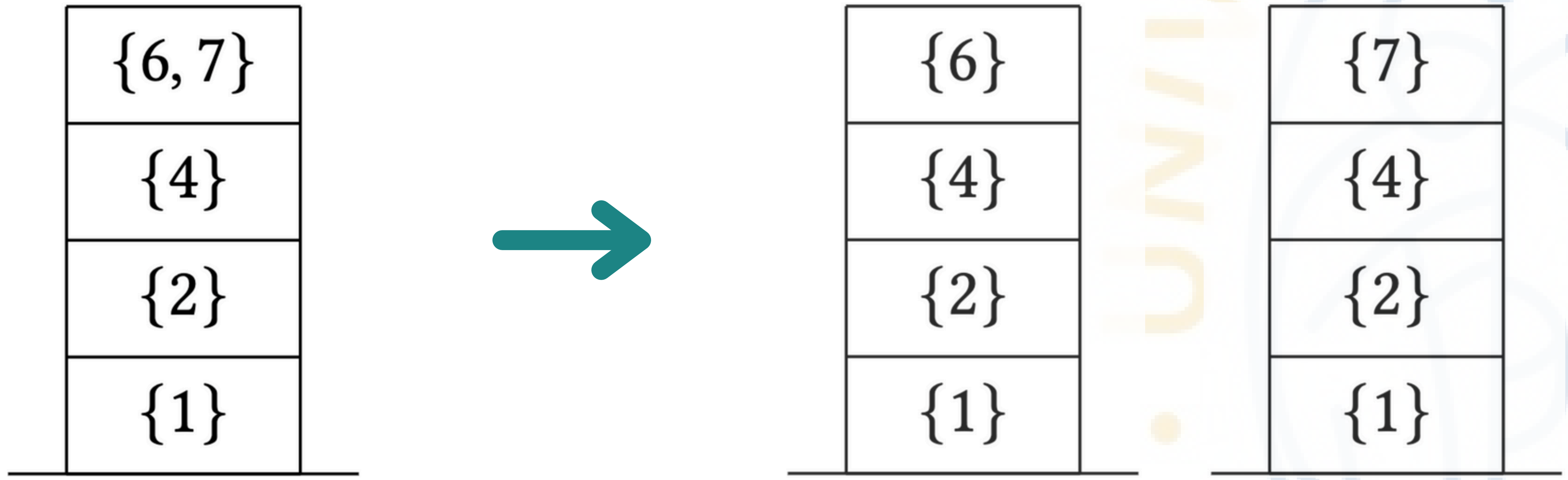


Example of lub operation with k = 1.

From Abstract Stacks to Sets of Abstract Stacks (2/2)

Solution

- Define an abstract stacks set domain with sets of abstract stacks, with at most l elements
- $\mathsf{SetSt}^{\#}_{k,h,l} \triangleq \langle \wp_{\leq l}(\mathcal{S}_{\mathsf{Ints}_k,h}) \cup \{\top_{\mathsf{SetSt}^{\#}_{k,h,l}}\}, \sqcup_{\mathsf{SetSt}^{\#}_{k,h,l}}, \sqcap_{\mathsf{SetSt}^{\#}_{k,h,l}}, \top_{\mathsf{SetSt}^{\#}_{k,h,l}}, \varnothing \rangle$
- $\top_{\mathrm{SetSt}^{\#}_{k,h,l}}$ is returned when the size of the abstract stacks set exceeds l
- No longer need to compute the lub on abstract stacks
- Each element of an abstract stack can now be an integer value (k = 1)



Before ***After***

Experimental Evaluation (1/5)

EVMLiSA*

- Static analyzer for EVM bytecode built on LiSA (**Li**brary for **S**tatic **A**nalysis)
- Generates CFGs from EVM bytecodes based on the approach described in this paper

Evaluation

- Dataset: ~1700 smart contracts from a set of 5000 Ethereum contracts
- Contracts have fewer than 3000 opcodes each (to allow manual inspection)
- Benchmark suite: ~3M opcodes in total, including ~240K jumps

* github.com/lisa-analyzer/evm-lisa

Experimental Evaluation (2/5)

$\top_{\mathbb{Z}_k^\sharp}$: top
$\top_{\mathbb{Z}}$: top may jump target
$\top_{\overline{\mathbb{Z}}}$: top don't jump target

Jump classification

Our evaluation measures resolved jumps, classified as follows:

- ***Resolved***: if all the top values of $\texttt{SetSt}^{\#}_{k,h,l}$ are integer values or $\top_{\overline{\mathbb{Z}}}$
- ***Unresolved***: if any stack reaching the jump has an unknown value that could be a valid destination
- ***Maybe unreachable***: if a jump node is not reached in the CFG by a path from its entry node
- ***Definitely unreachable***: if no stack reaches the jump node
- ***Maybe unresolved***: if the stack set exceeded the maximal stack size l

(Maybe) Unresolved jumps ***can be reduced*** by fine-tuning the parameters l and *h*.

Experimental Evaluation (3/5)

Results

- We run EVMLiSA on the ~1700 smart contracts with h = 128 and l = 32, corresponding to the *maximal height* of abstract stacks and the *maximal size* of abstract stack sets, respectively

Classification	% Jumps
Resolved	96.73
Maybe unreachable	2.41
Definitely unreachable	0.69
Unresolved	0.16
Maybe unresolved	0.01

Experimental Evaluation (4/5)

SLOAD problem

- Jumps marked as (maybe) unresolved are caused by the SLOAD opcode
- SLOAD pops a stack element to fetch a value from blockchain memory, which is statically unknown
- EVMLiSA models SLOAD by popping and pushing $\top_{\mathbb{Z}}$ onto the stack

Specific observation

- The retrieved value was used as a jump destination, leading to an unresolved jump label
- This may result from static analysis over-approximation

We leave the handling of this specific precision problem ***as future work***.

Experimental Evaluation (5/5)

Further experiment

- Refined benchmark: selected contracts where SLOAD value doesn't affect jump destination
- ~550 smart contracts, ~837K opcodes, ~59K jumps, h = 128, l = 32

Classification	% Jumps (all tests)	% Jumps (refined)
Resolved	96.73	**97.83**
Maybe unreachable	2.41	**0**
Definitely unreachable	0.69	**2.17**
Unresolved	0.16	**0**
Maybe unresolved	0.01	**0**

Future works

Resolving SLOAD problem

- Introducing the ability to read external information from the persistent storage
- Hybrid beta-version already implemented, *resolving 100% of jumps* in the original benchmark of 5000 smart contracts
- Hybrid approach is effective but strays from static analysis principles due to reliance on external data.

Checker

- Developing Reentrancy & Buffer Overflow checker
- Implementing a Gas Estimator

Conclusions

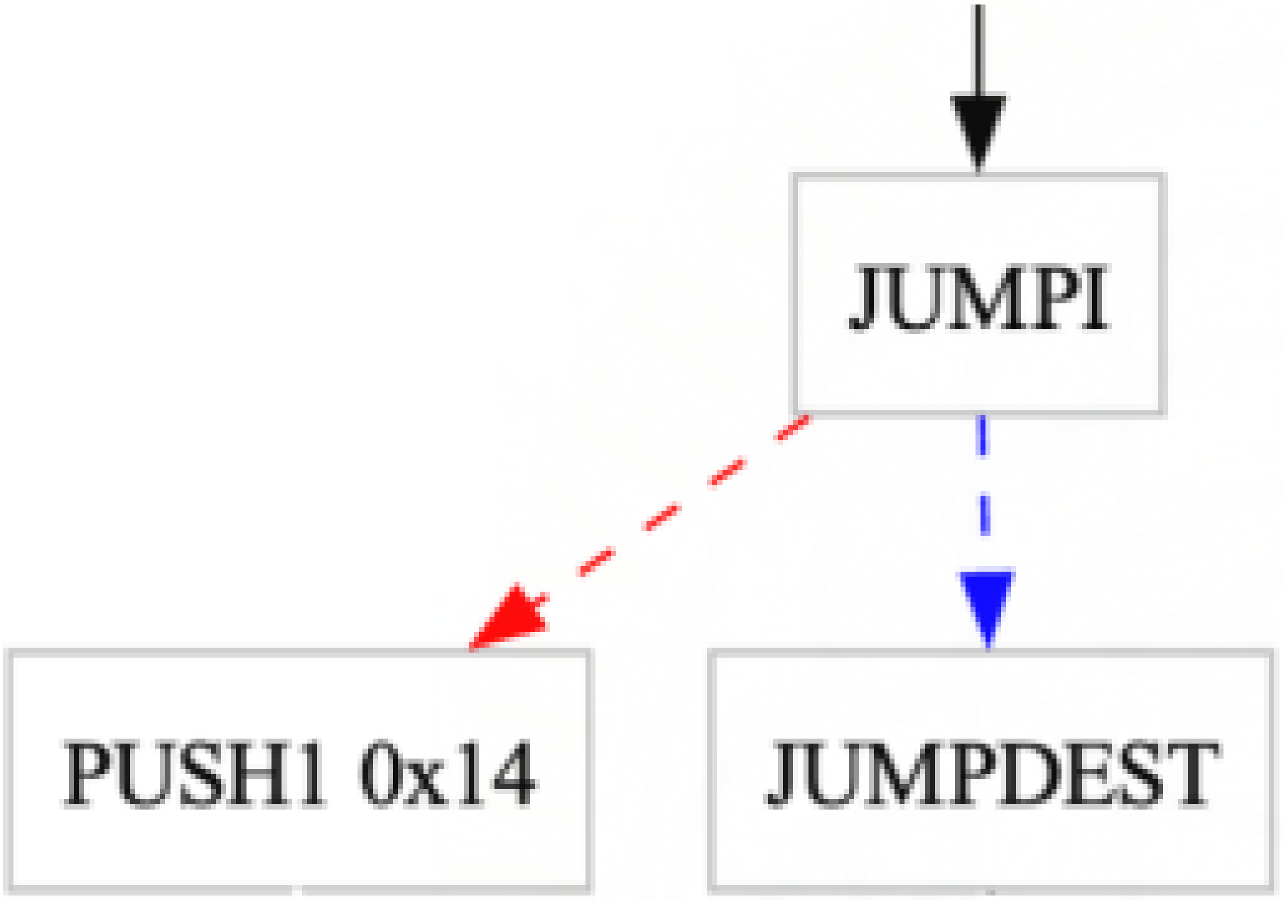
- Introduced a ***new approach*** to constructing sound CFGs for EVM bytecode
- Used ***abstract interpretation*** to over-approximate behavior and identify dynamic jump destinations
- Refined the CFG iteratively, using ***domains tailored*** to EVM's characteristics
- Implemented ***EVMLiSA***, showing practical effectiveness
- Tested on real smart contracts, ***proving*** it handles real-world EVM bytecode

EVMLiSA

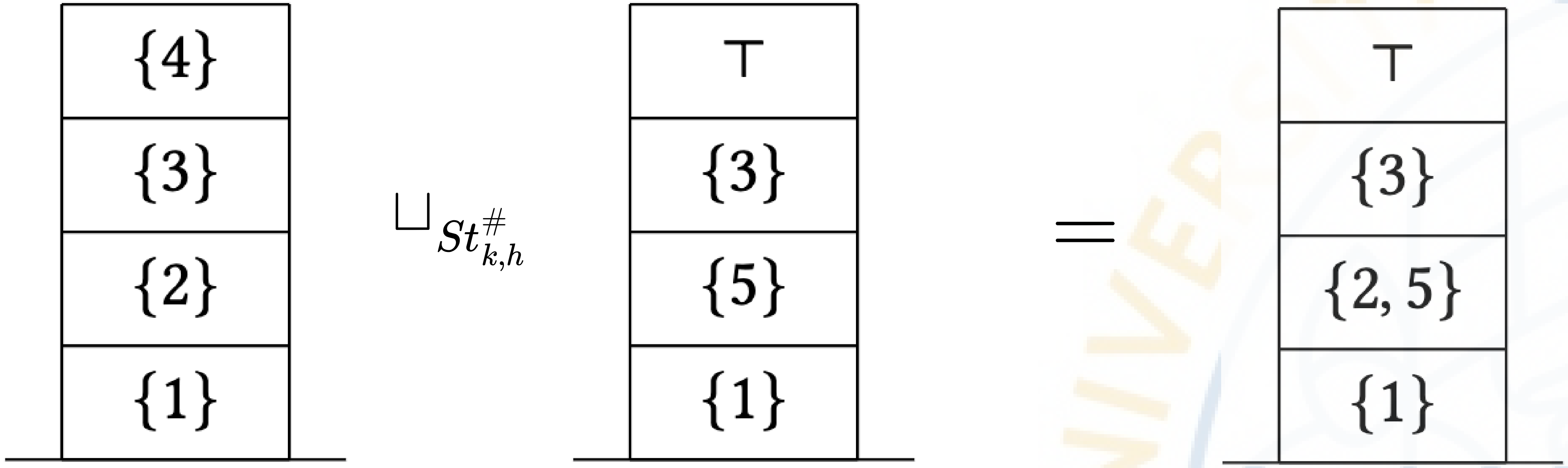
github.com/lisa-analyzer/evm-lisa

Bonus (How JUMPI works)

[10] PUSH1 0x32
[12] JUMPI
[13] PUSH1 0x14
...
[32] JUMPDEST
[33] EQ



Bonus (A benefit of the Abstract Stacks Set)



Example of lub operation with k = 2.