Dynamic Behavior Analysis Using Binary Instrumentation

Jonathan Salwan
jsalwan@quarkslab.com

St'Hack
Bordeaux – France

March 27 2015

Keywords: program analysis, DBI, DBA, Pin, concrete execution, symbolic execution, DSE, taint analysis, context snapshot, Z3 theorem prover and behavior analysis.

QUARKSLAB
INNOVATIVE SECURITY

Who Am I?

- I am a junior security researcher at Quarkslab working on tools development for programs analysis
- I have a strong interest in all low level computing
- I like to play with weird things even though it sometimes seems useless

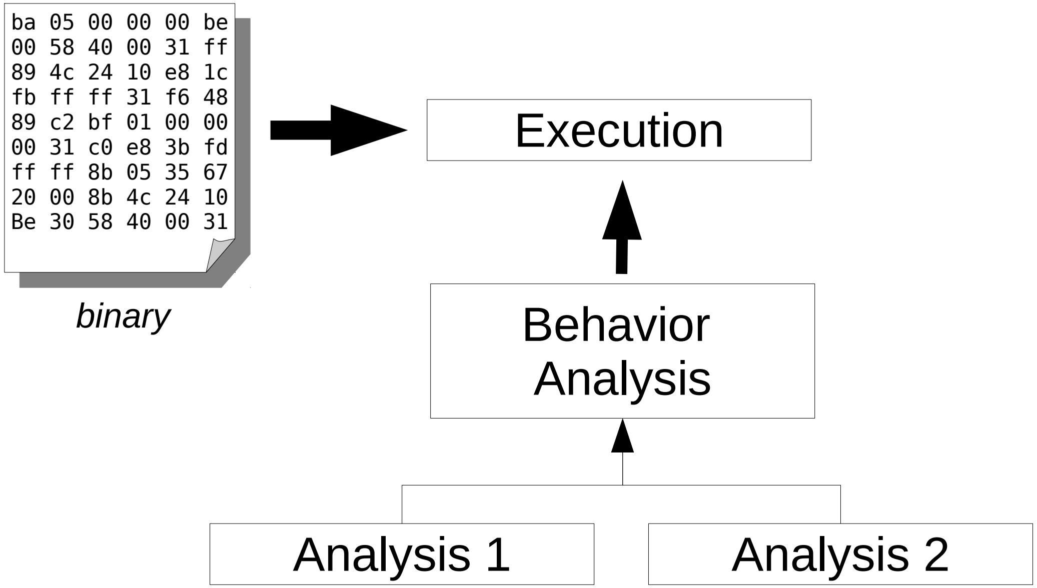
Roadmap Of This Talk

- Few words about the goal of this stuff
- Short review of the Security Day Lille's talk
  - Really short introduction
  - Covering a function using DSE approach
  - Some words about vulnerabilities hunting
- Objectives of this talk
  - Build specific analysis to find specific bugs
    - Analysis for use-after-free detection
    - Analysis for heap overflow detection
    - Analysis for stack overflow detection
    - Analysis for format string detection
    - Analysis for {wrtite, read}-what-where detection
  - Few words about generic analysis
  - Few words about the **Triton** project !!
- Conclusion
- Q&A

First Of All – The Goal Of This Stuff

First Of All – The Goal Of This Stuff

- Main goal:
  - I want to analyze a binary dynamically
  - I want to find unexpected behaviors in its execution to find potential vulnerabilities
    - Even if these bugs do not crash the program



```
ba 05 00 00 00 be
00 58 40 00 31 ff
89 4c 24 10 e8 1c
fb ff ff 31 f6 48
89 c2 bf 01 00 00
00 31 c0 e8 3b fd
ff ff 8b 05 35 67
20 00 8b 4c 24 10
Be 30 58 40 00 31
```

Short Review Of The Security Day Lille's Talk

Short Review Of The Security Day Lille's Talk

- In the last talk [0], we saw how it was possible to cover a function in memory using a dynamic symbolic execution approach
  - We generate and inject concrete values in memory in order to go through all paths

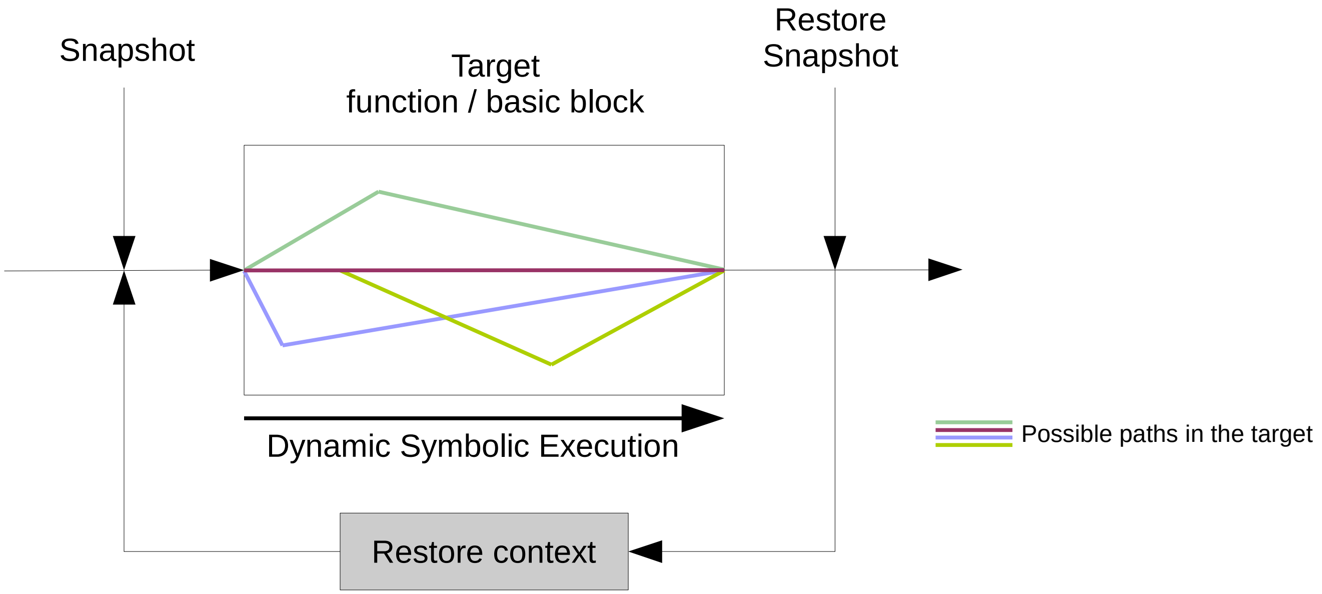
[0] http://shell-storm.org/talks/SecurityDay2015_dynamic_symbolic_execution_Jonathan_Salwan.pdf

Short Review Of The Security Day Lille's Talk

- Basically, we:
  - Target a function
  - Take a context snapshot at the first instruction
  - Switch to a dynamic symbolic execution in order to build the path constraint (*PC*)
  - Restore the snapshot and generate another concrete value to go through another path
  - Repeat this operation until all paths are taken

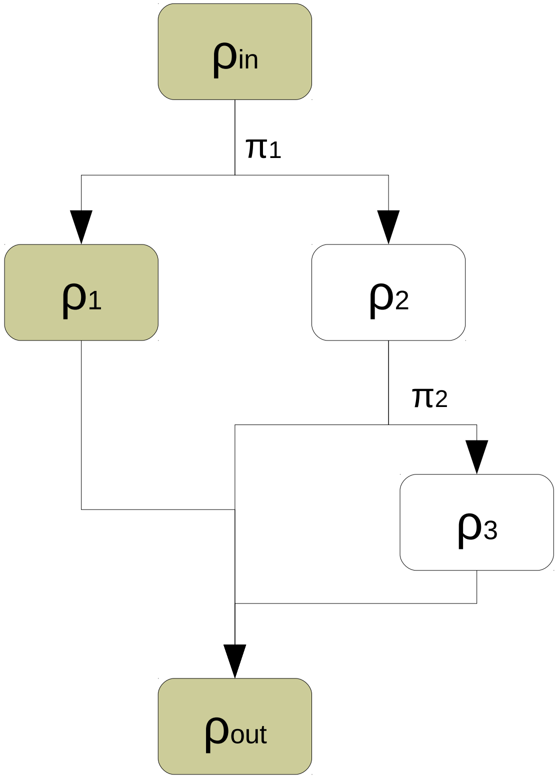
Short Review Of The Security Day Lille's Talk

- In a nutshell, we got something like this:



Dynamic Symbolic Execution

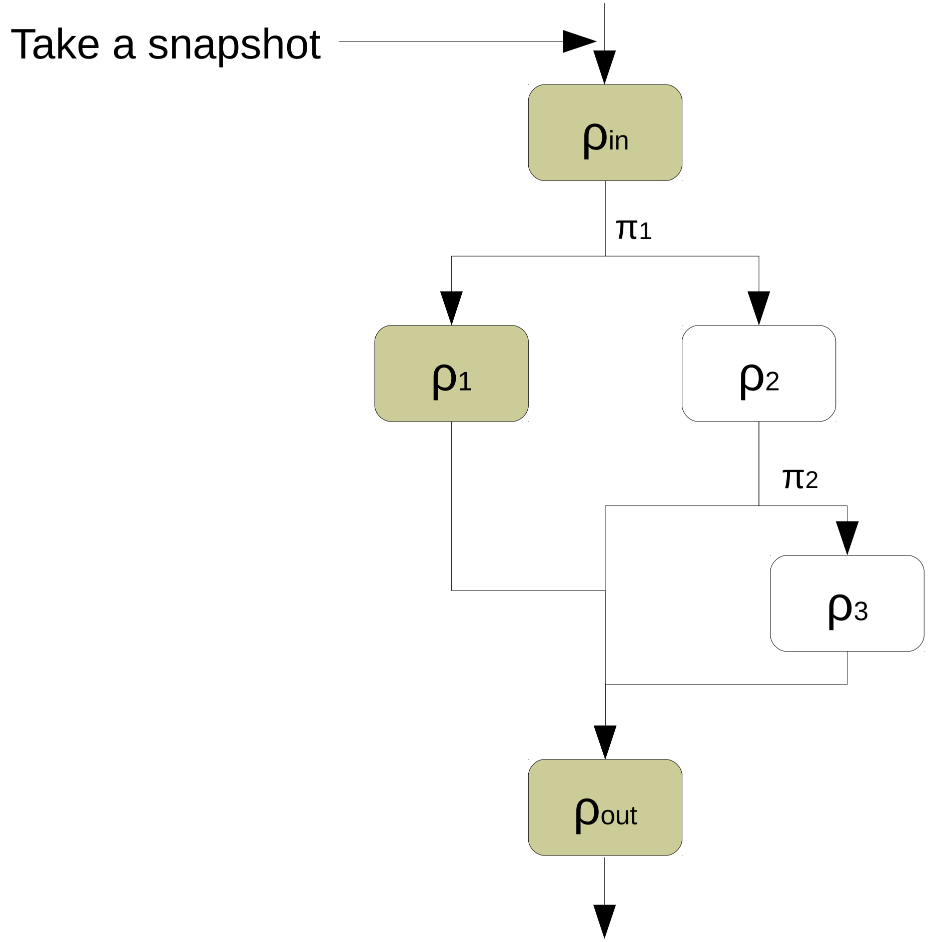
- The Dynamic Symbolic Execution process looks like this:



Control flow graph

Dynamic Symbolic Execution

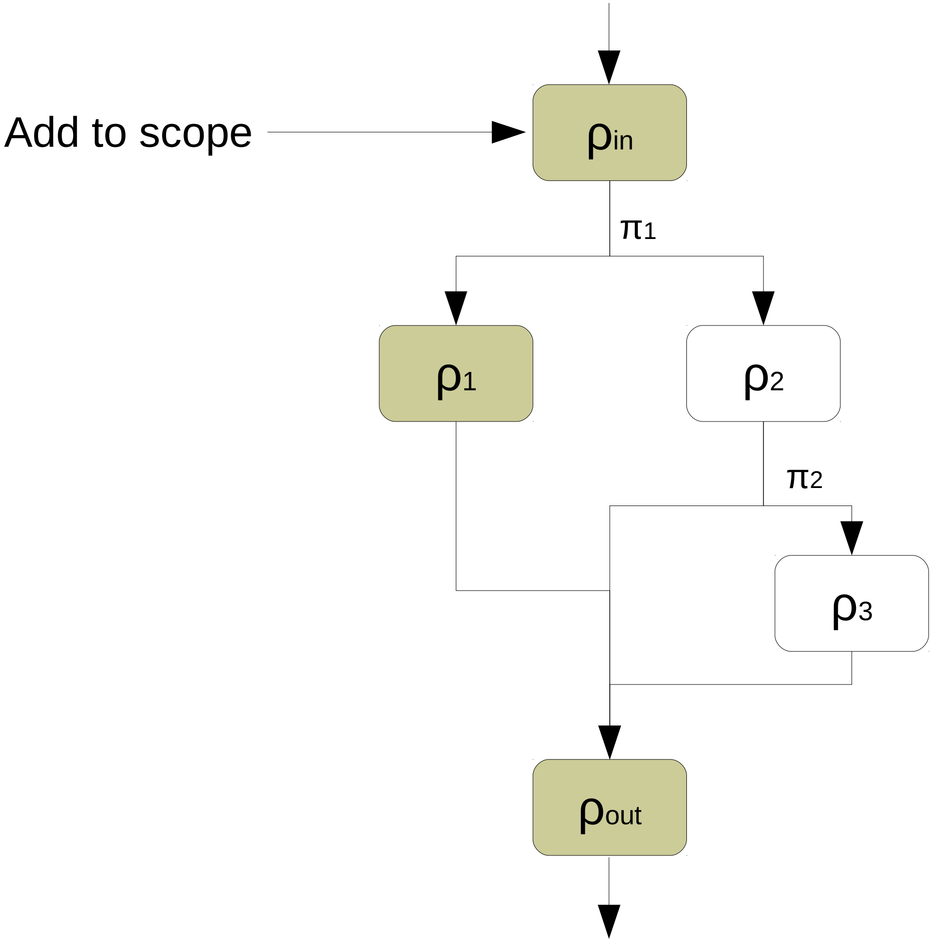
- The Dynamic Symbolic Execution process looks like this:



<u>1st iteration</u>

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:

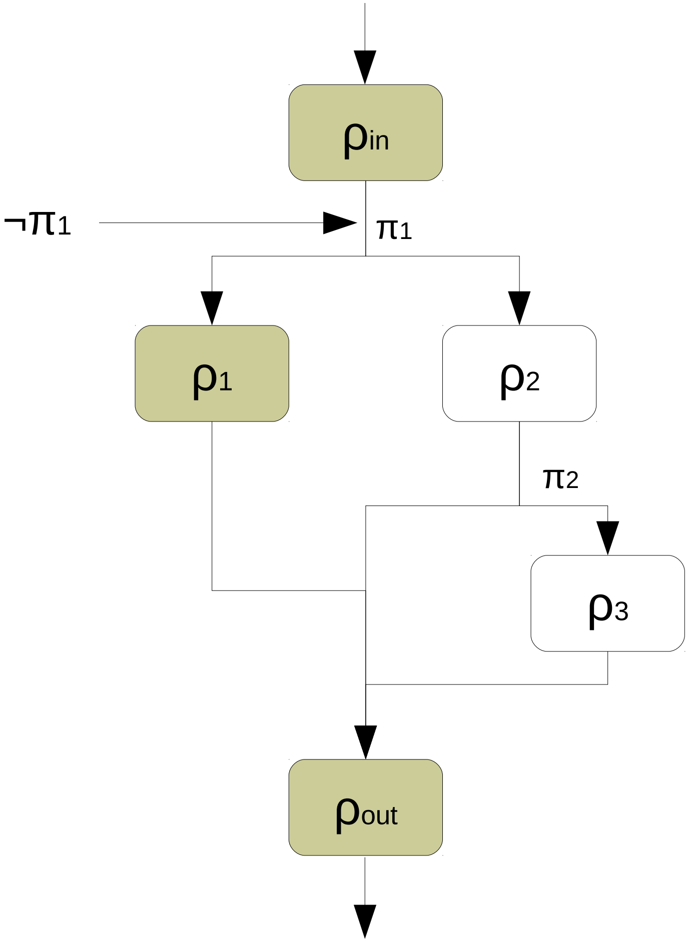


<u>1st iteration</u>

PC = T : {ρ_{in}}

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:

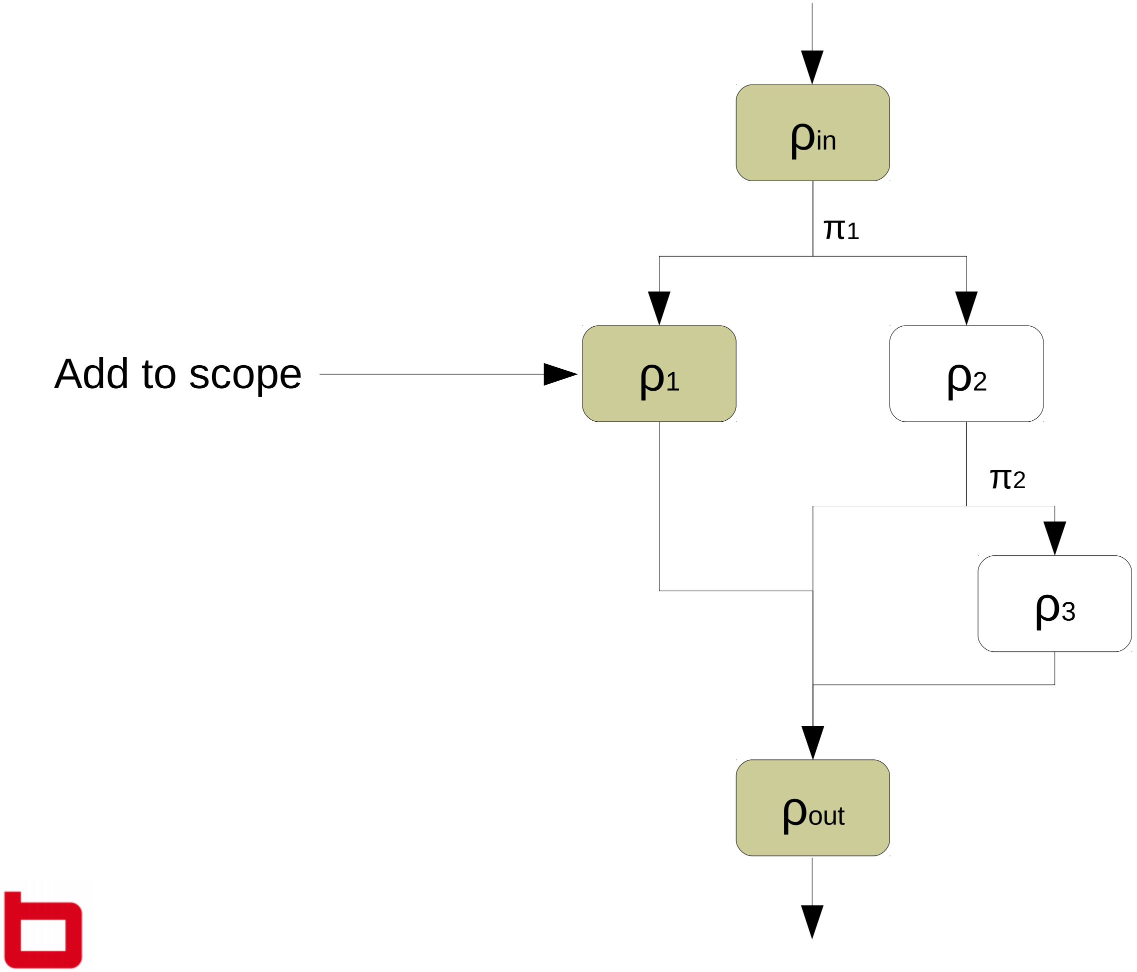


1st iteration

PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:

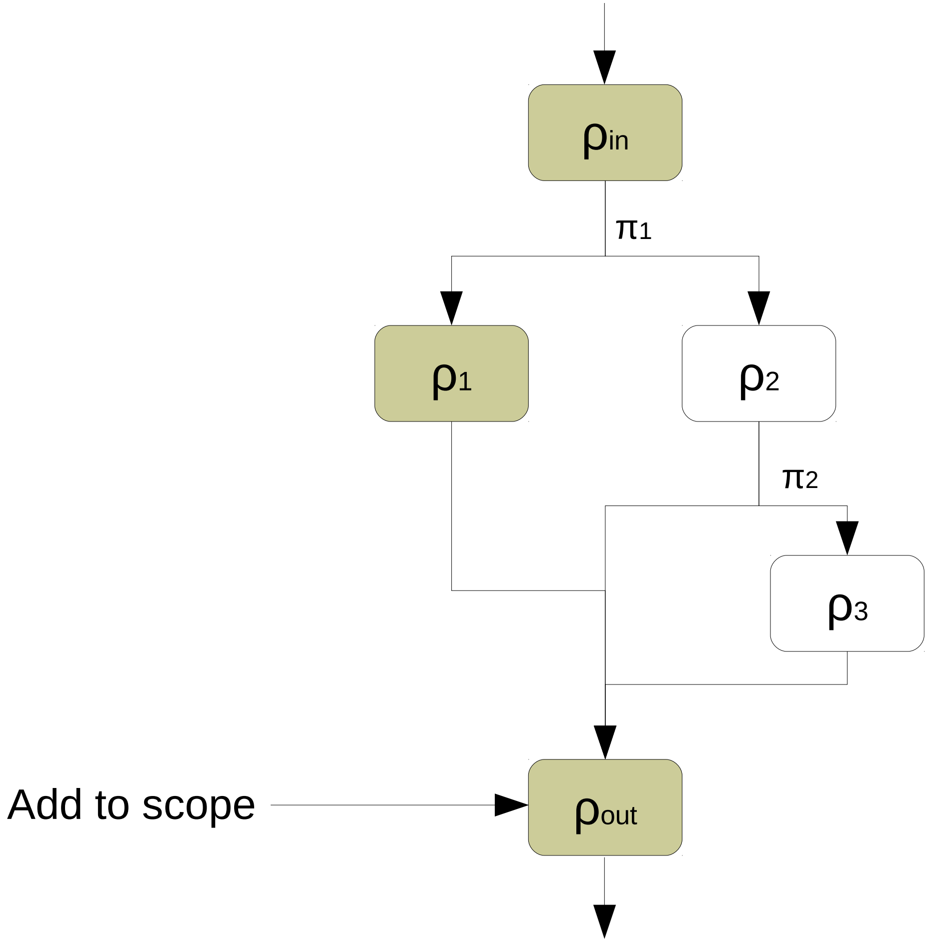


1st iteration

$PC = T : \{\rho_{in}\}$
$PC = \neg\pi_1 : \{\rho_{in}, \rho_1\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:

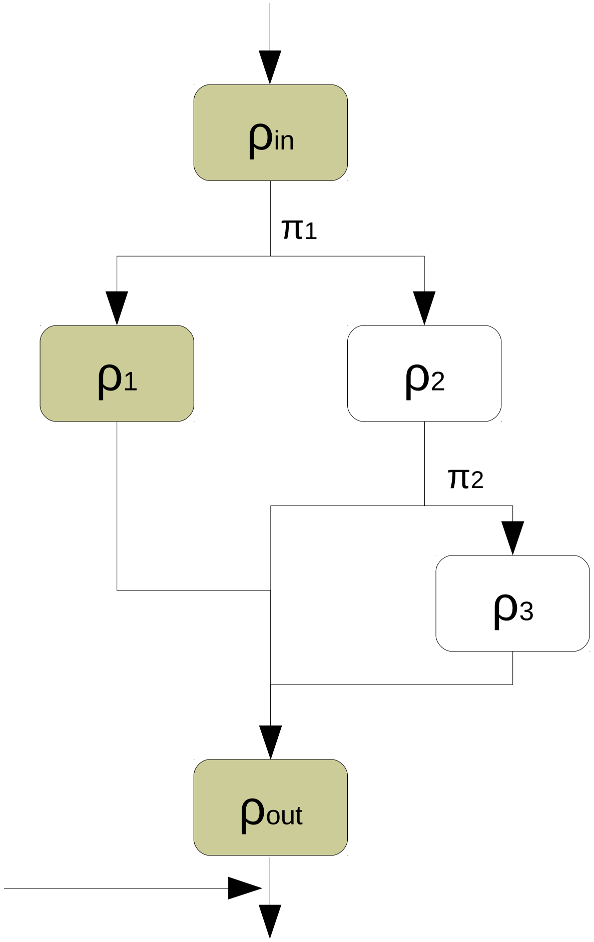


1st iteration

$PC = T : \{\rho_{in}\}$
$PC = \neg\pi_1 : \{\rho_{in}, \rho_1, \rho_{out}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



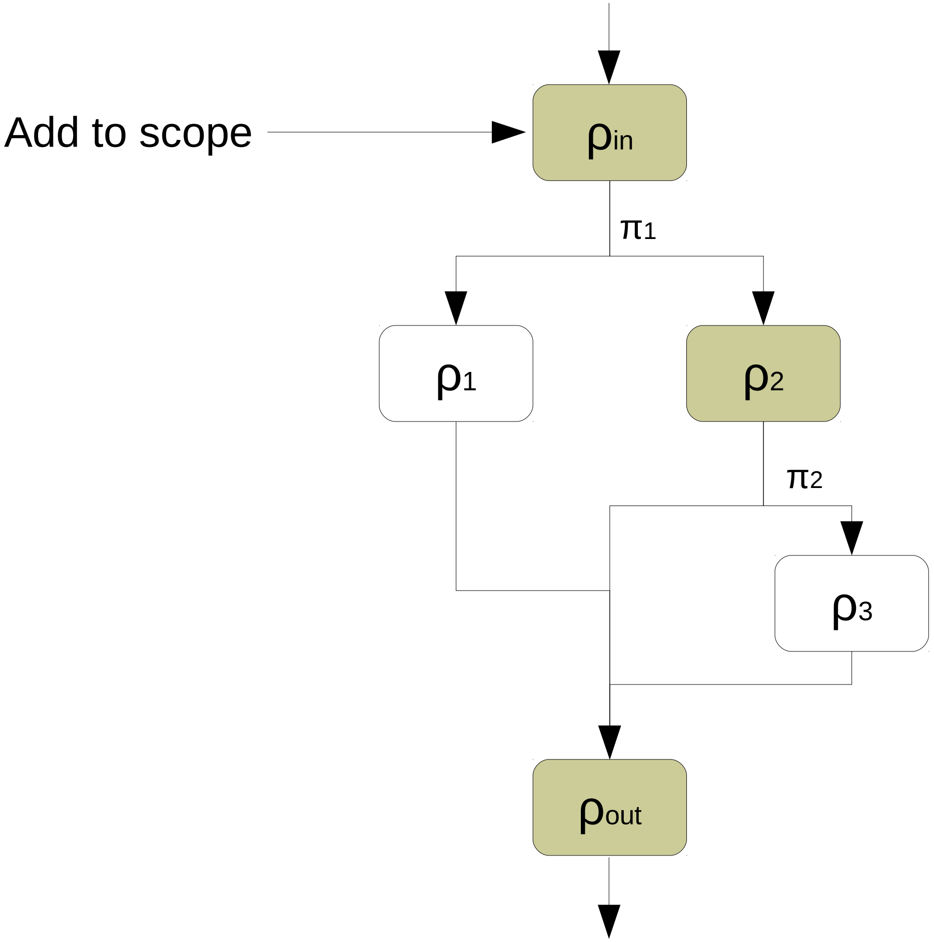
Restore the snapshot

1st iteration

PC = T : {ρ_{in}}
PC = $\neg\pi_1$: {ρ_{in}, ρ_1, ρ_{out}}

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

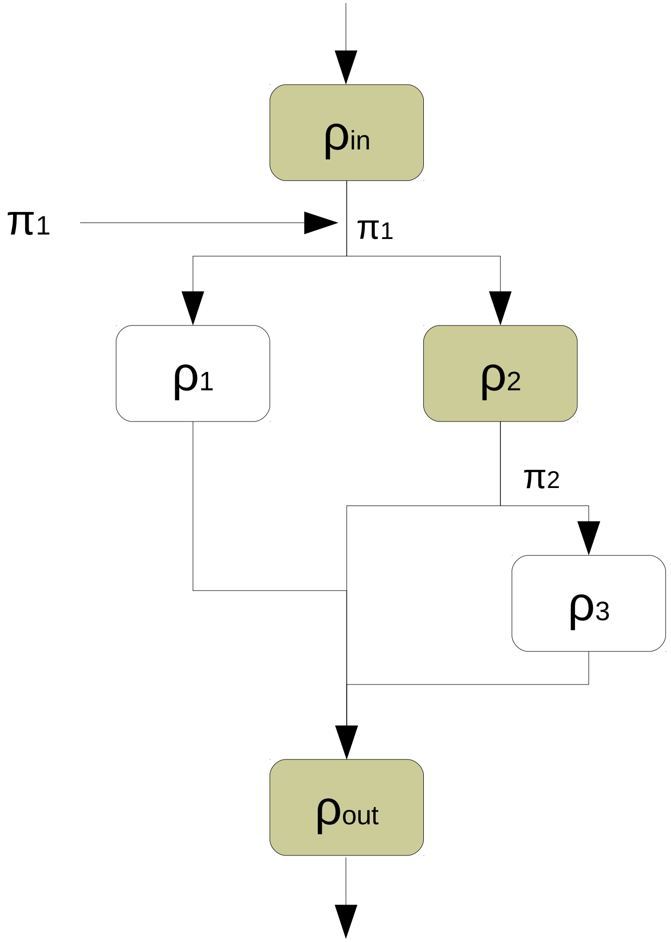
PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

PC = T : $\{\rho_{in}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

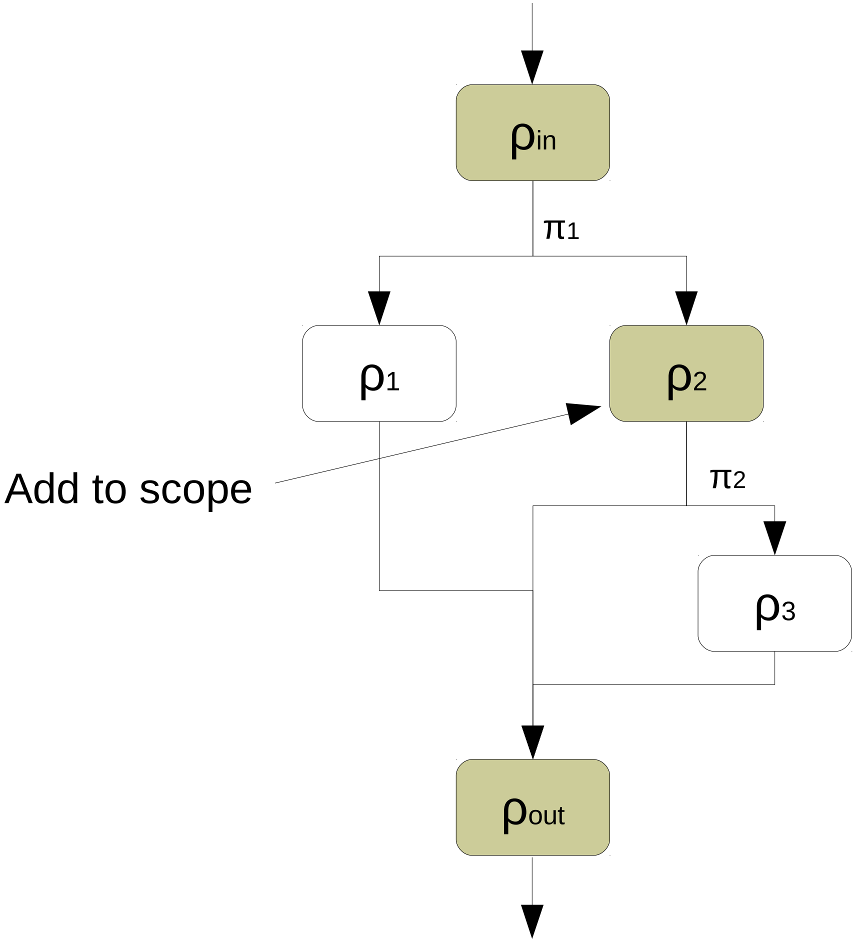
$PC = T : \{\rho_{in}\}$
$PC = \neg\pi_1 : \{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

$PC = T : \{\rho_{in}\}$
$PC = \pi_1 : \{\rho_{in}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

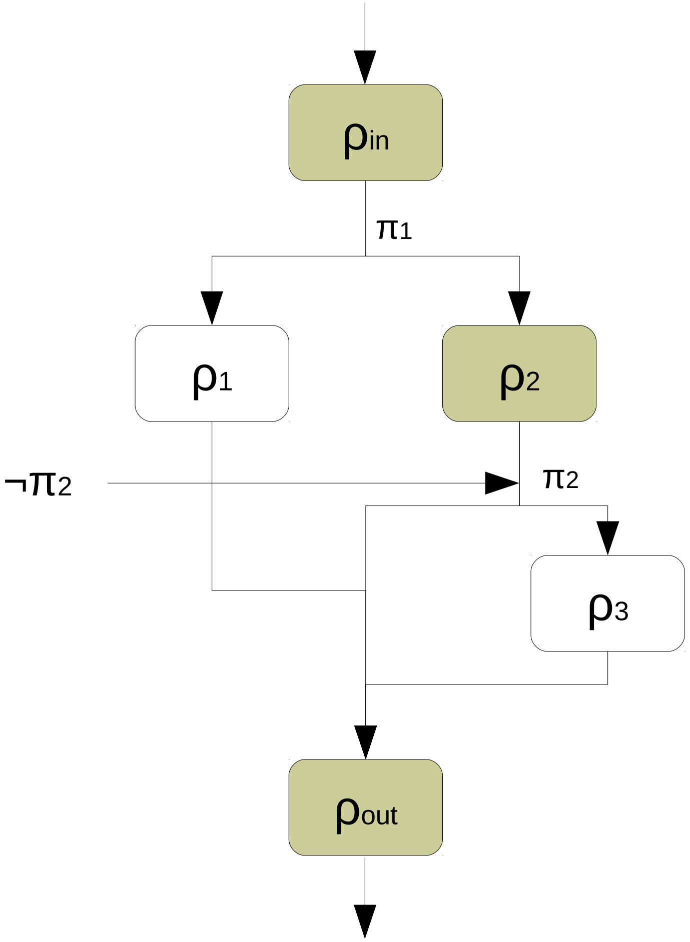
PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

PC = T : $\{\rho_{in}\}$
PC = π_1 : $\{\rho_{in}, \rho_2\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

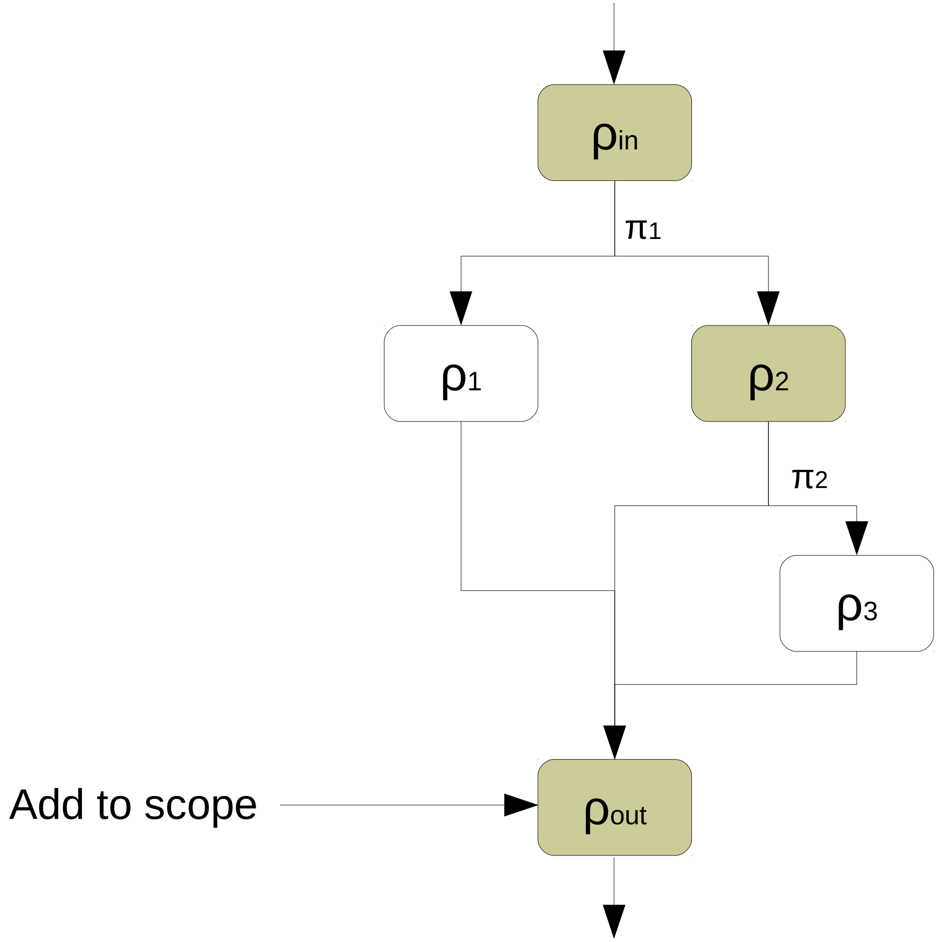
PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \land \neg\pi_2$: $\{\rho_{in}, \rho_2\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

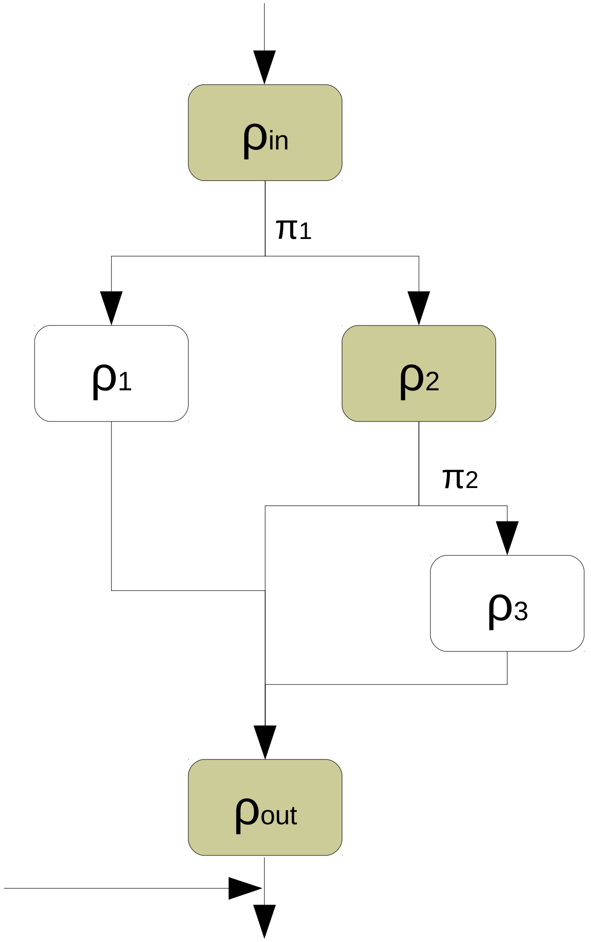
PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \land \neg\pi_2$: $\{\rho_{in}, \rho_2, \rho_{out}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



Restore the snapshot

1st iteration

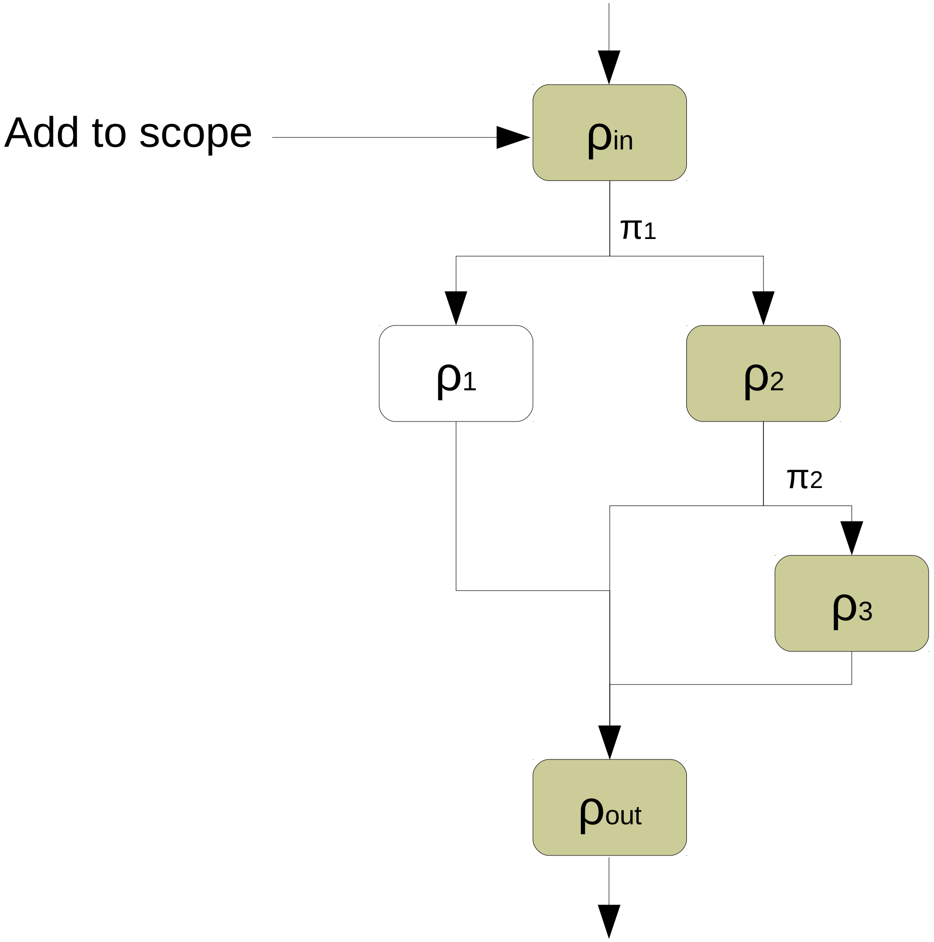
PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \wedge \neg\pi_2$: $\{\rho_{in}, \rho_2, \rho_{out}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

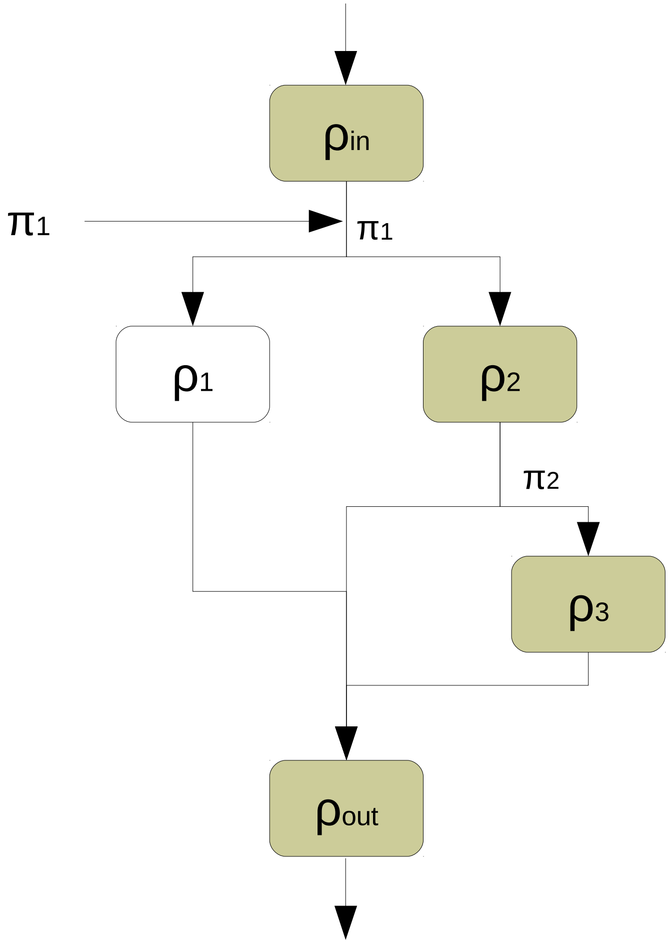
PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \wedge \neg\pi_2$: $\{\rho_{in}, \rho_2, \rho_{out}\}$

3th iteration

PC = T : $\{\rho_{in}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

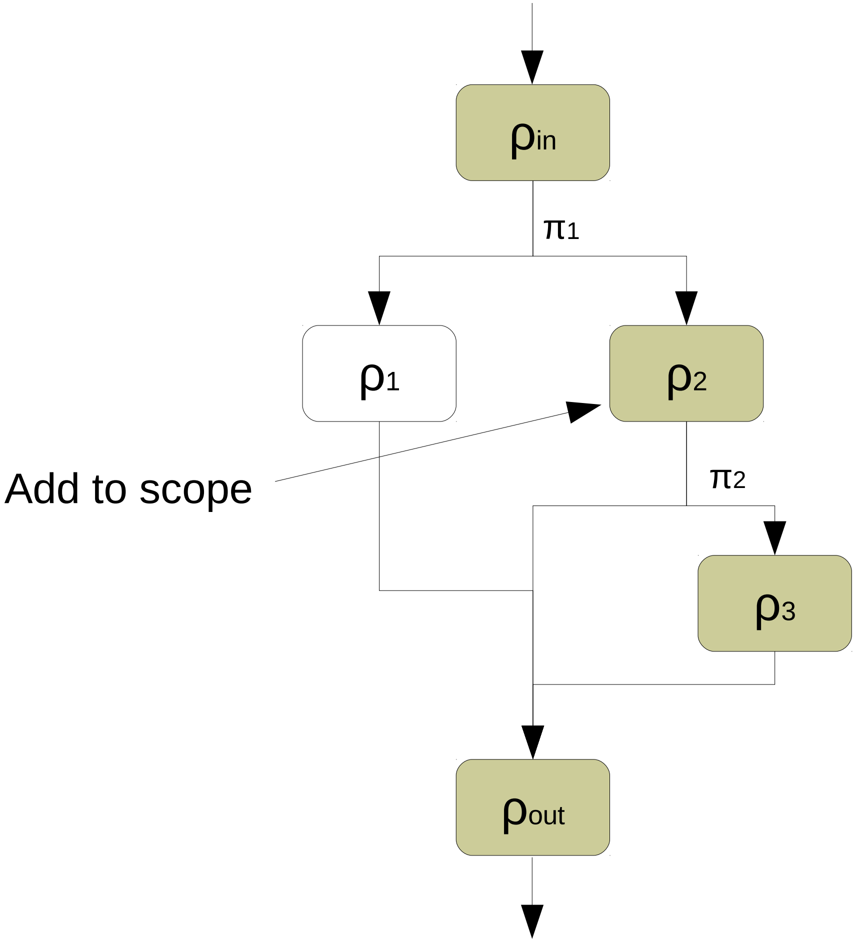
PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \wedge \neg\pi_2$: $\{\rho_{in}, \rho_2, \rho_{out}\}$

3th iteration

PC = T : $\{\rho_{in}\}$
PC = π_1 : $\{\rho_{in}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

$PC = T : \{\rho_{in}\}$
$PC = \neg\pi_1 : \{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

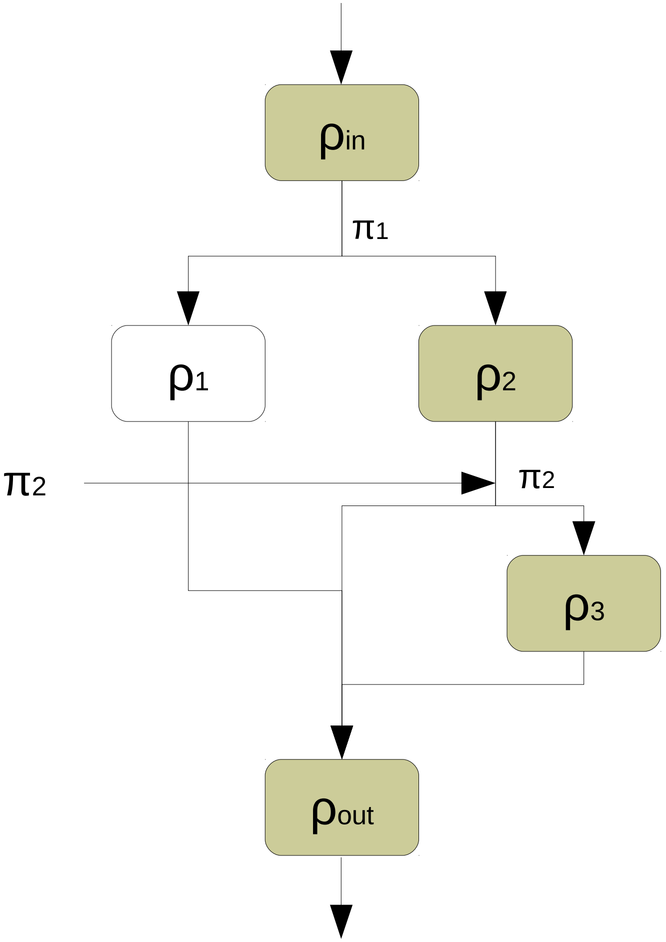
$PC = T : \{\rho_{in}\}$
$PC = \pi_1 \land \neg\pi_2 : \{\rho_{in}, \rho_2, \rho_{out}\}$

3th iteration

$PC = T : \{\rho_{in}\}$
$PC = \pi_1 : \{\rho_{in}, \rho_2\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



<u>1st iteration</u>

PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

<u>2nd iteration</u>

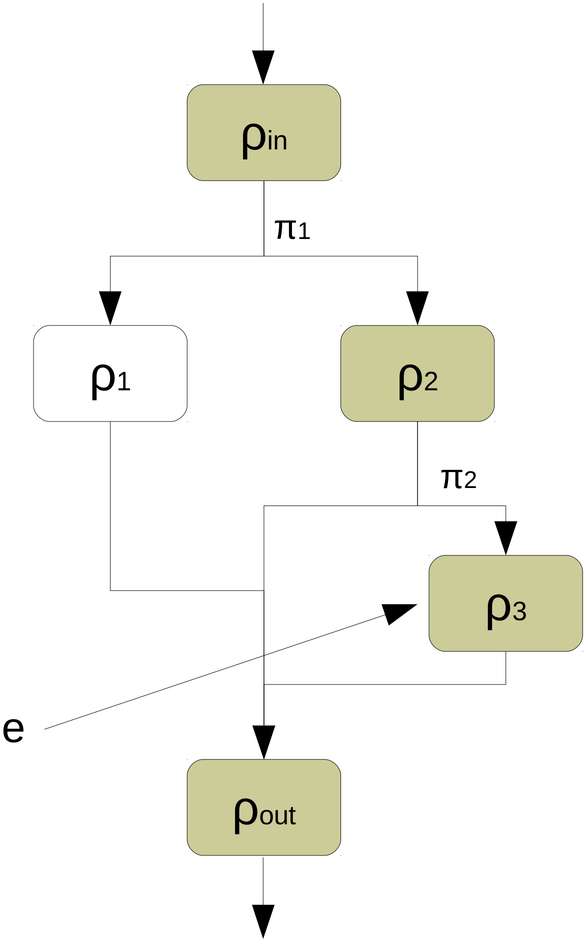
PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \land \neg\pi_2$: $\{\rho_{in}, \rho_2, \rho_{out}\}$

<u>3th iteration</u>

PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \land \pi_2$: $\{\rho_{in}, \rho_2\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



Add to scope

1st iteration

$PC = T : \{\rho_{in}\}$
$PC = \neg\pi_1 : \{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

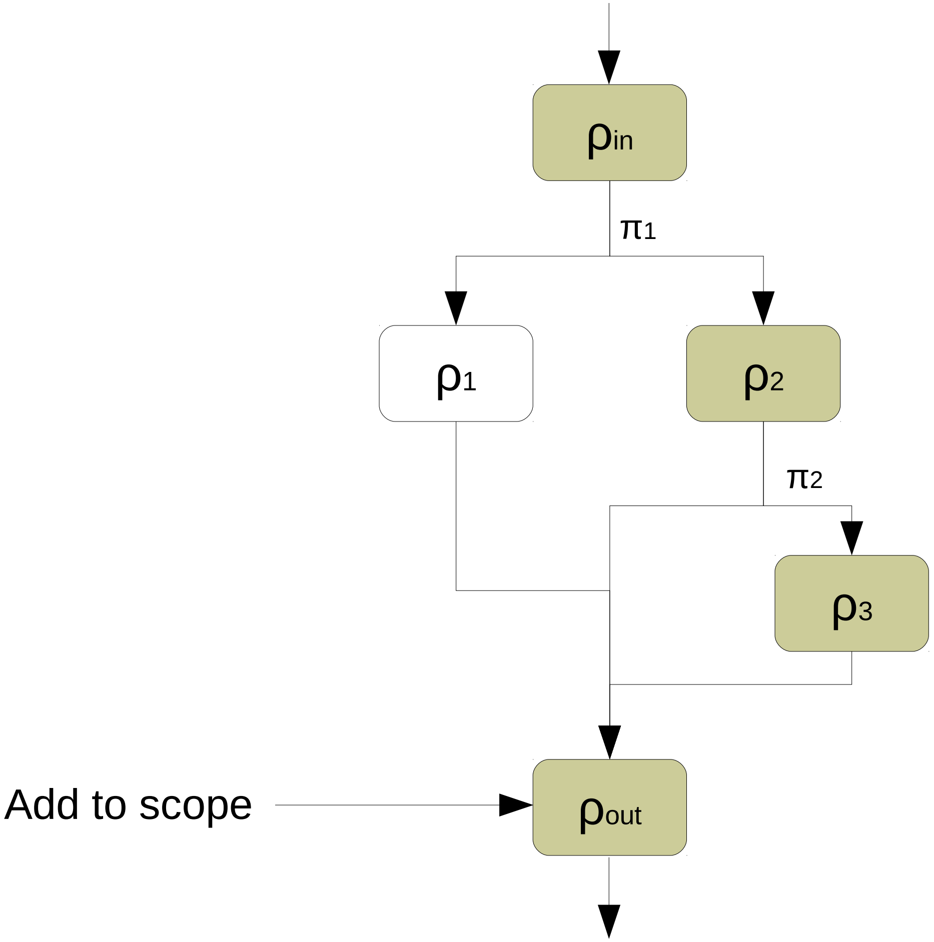
$PC = T : \{\rho_{in}\}$
$PC = \pi_1 \wedge \neg\pi_2 : \{\rho_{in}, \rho_2, \rho_{out}\}$

3th iteration

$PC = T : \{\rho_{in}\}$
$PC = \pi_1 \wedge \pi_2 : \{\rho_{in}, \rho_2, \rho_3\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:



1st iteration

PC = T : $\{\rho_{in}\}$
PC = $\neg\pi_1$: $\{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \wedge \neg\pi_2$: $\{\rho_{in}, \rho_2, \rho_{out}\}$

3th iteration

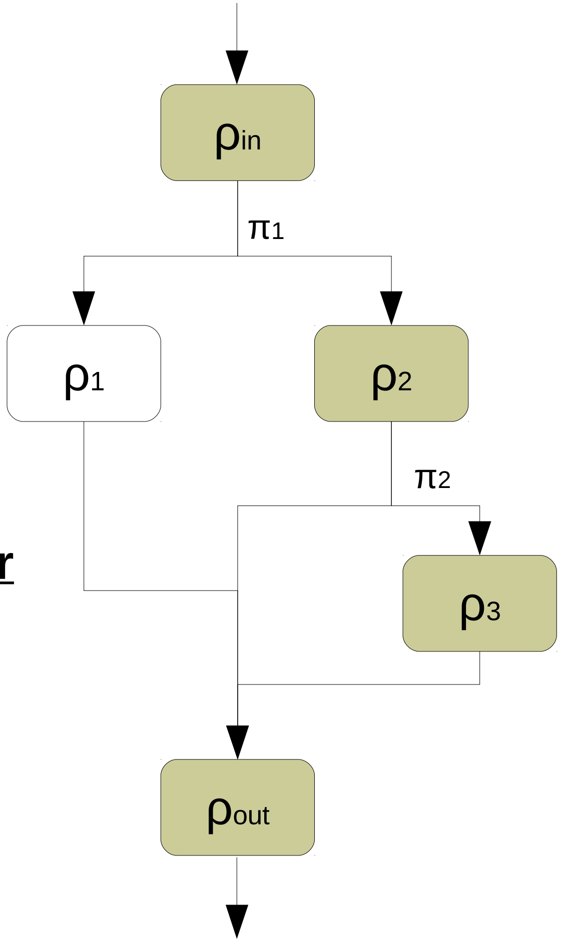
PC = T : $\{\rho_{in}\}$
PC = $\pi_1 \wedge \pi_2$: $\{\rho_{in}, \rho_2, \rho_3, \rho_{out}\}$

Dynamic Symbolic Execution

- The Dynamic Symbolic Execution process looks like this:

There is no more branches, so the restore process is over

$PC_1 = T : \{\rho_{in}\}$
$PC_2 = \neg\pi_1 : \{\rho_{in}, \rho_1, \rho_{out}\}$
$PC_3 = \pi_1 \land \neg\pi_2 : \{\rho_{in}, \rho_2, \rho_{out}\}$
$PC_4 = \pi_1 \land \pi_2 : \{\rho_{in}, \rho_2, \rho_3, \rho_{out}\}$



1st iteration

$PC = T : \{\rho_{in}\}$
$PC = \neg\pi_1 : \{\rho_{in}, \rho_1, \rho_{out}\}$

2nd iteration

$PC = T : \{\rho_{in}\}$
$PC = \pi_1 \land \neg\pi_2 : \{\rho_{in}, \rho_2, \rho_{out}\}$

3th iteration

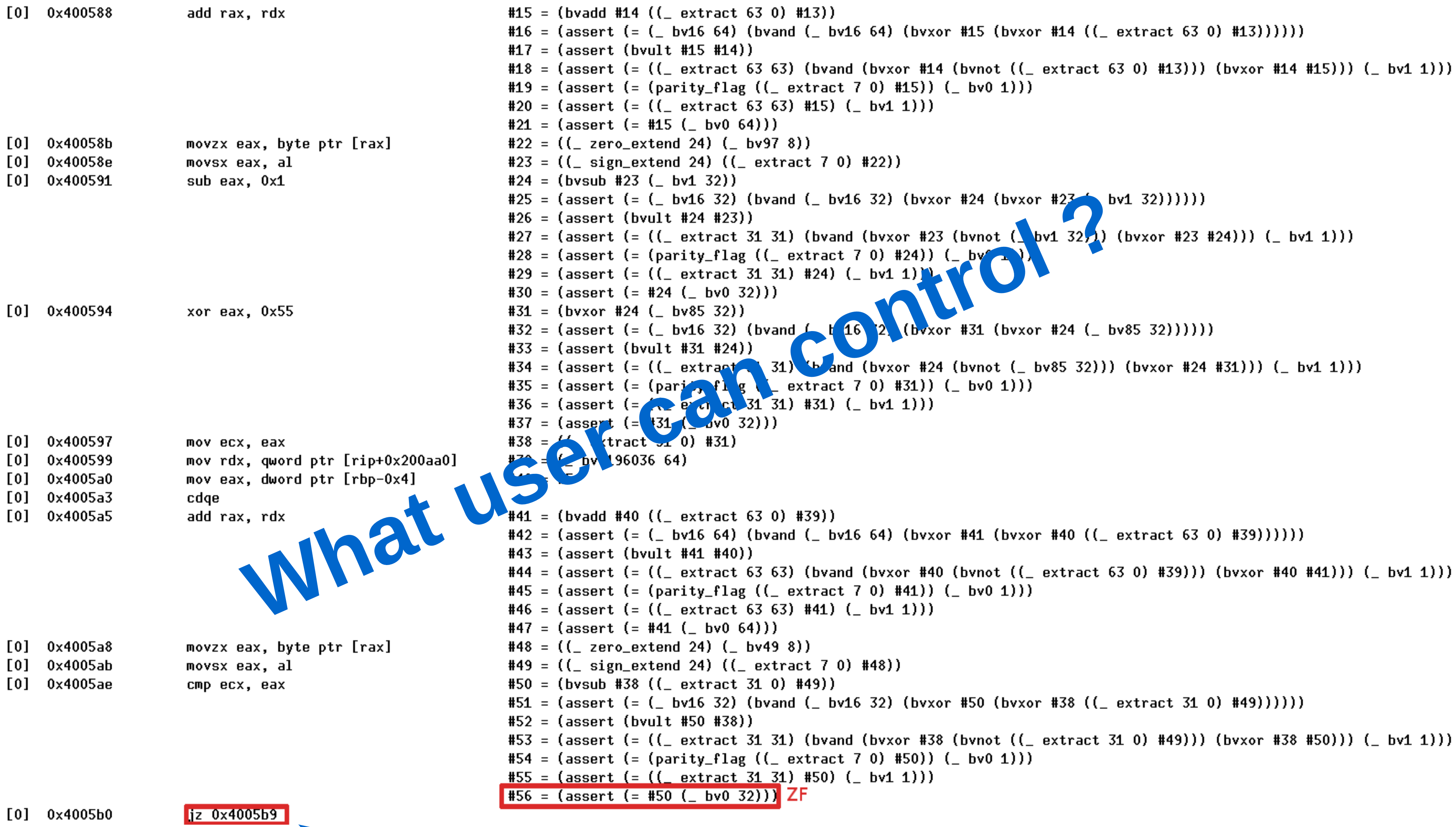
$PC = T : \{\rho_{in}\}$
$PC = \pi_1 \land \pi_2 : \{\rho_{in}, \rho_2, \rho_3, \rho_{out}\}$

Symbolic Execution Guided By The Taint Analysis

Symbolic Execution Guided By The Taint Analysis

- Taint analysis provides information about which registers and memory addresses are controllable by the user at each program point:
  - Assists the symbolic engine to setup the symbolic variables (a symbolic variable is a memory area that the user can control)
  - May assist the symbolic engine to perform some symbolic optimizations
  - At each branch instruction, we directly know if the user can go through both branches (this is mainly used for code coverage)

Symbolic Execution Guided By The Taint Analysis

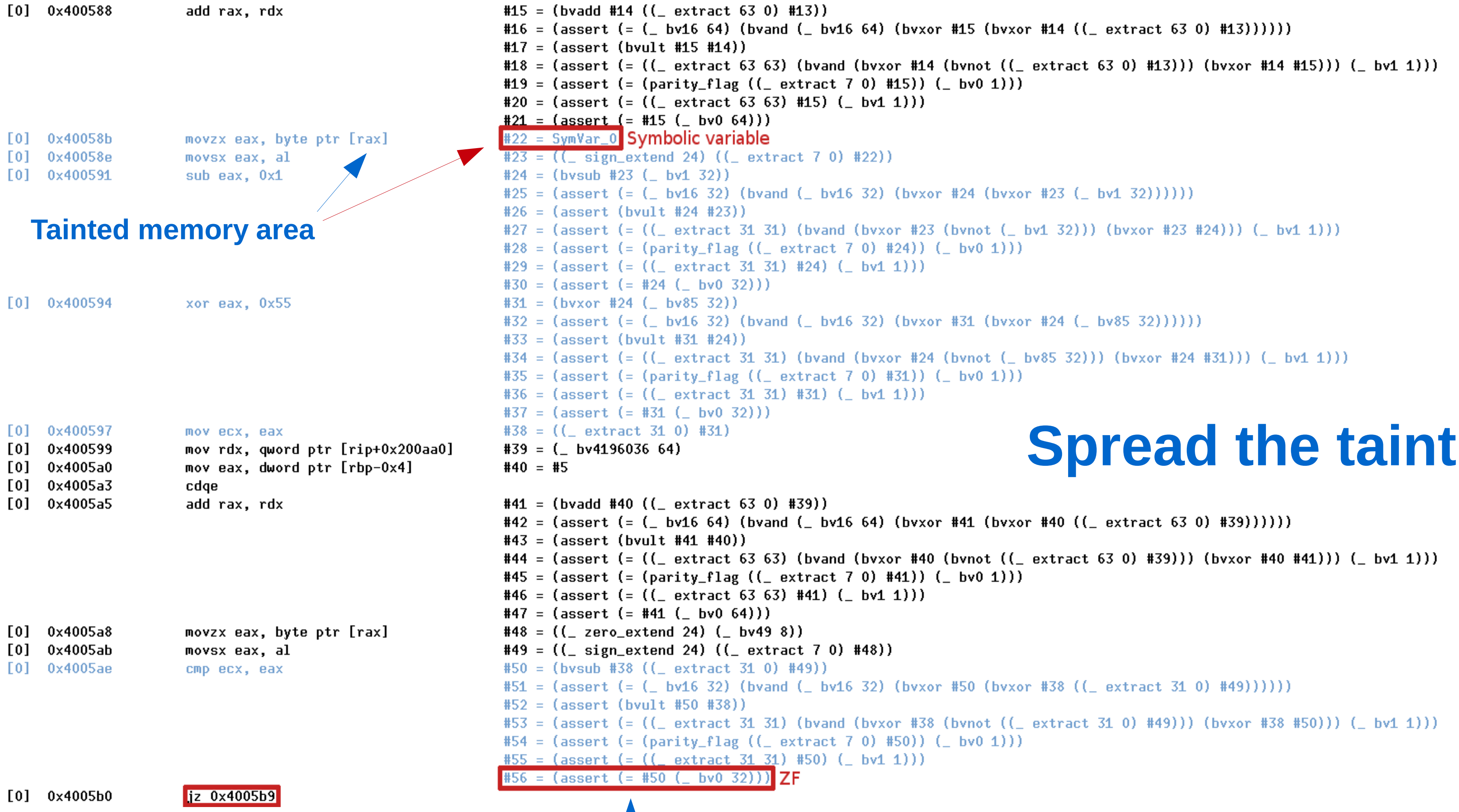


```
[0] 0x400588    add rax, rdx                        #15 = (bvadd #14 ((_ extract 63 0) #13))
                                                    #16 = (assert (= (_ bv16 64) (bvand (_ bv16 64) (bvxor #15 (bvxor #14 ((_ extract 63 0) #13))))))
                                                    #17 = (assert (bvult #15 #14))
                                                    #18 = (assert (= ((_ extract 63 63) (bvand (bvxor #14 (bvnot ((_ extract 63 0) #13))) (bvxor #14 #15))) (_ bv1 1)))
                                                    #19 = (assert (= (parity_flag ((_ extract 7 0) #15)) (_ bv0 1)))
                                                    #20 = (assert (= ((_ extract 63 63) #15) (_ bv1 1)))
                                                    #21 = (assert (= #15 (_ bv0 64)))
[0] 0x40058b    movzx eax, byte ptr [rax]           #22 = ((_ zero_extend 24) (_ bv97 8))
[0] 0x40058e    movsx eax, al                       #23 = ((_ sign_extend 24) ((_ extract 7 0) #22))
[0] 0x400591    sub eax, 0x1                        #24 = (bvsub #23 (_ bv1 32))
                                                    #25 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor #24 (bvxor #23 (_ bv1 32))))))
                                                    #26 = (assert (bvult #24 #23))
                                                    #27 = (assert (= ((_ extract 31 31) (bvand (bvxor #23 (bvnot (_ bv1 32))) (bvxor #23 #24))) (_ bv1 1)))
                                                    #28 = (assert (= (parity_flag ((_ extract 7 0) #24)) (_ bv0 1)))
                                                    #29 = (assert (= ((_ extract 31 31) #24) (_ bv1 1)))
                                                    #30 = (assert (= #24 (_ bv0 32)))
[0] 0x400594    xor eax, 0x55                       #31 = (bvxor #24 (_ bv85 32))
                                                    #32 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor #31 (bvxor #24 (_ bv85 32))))))
                                                    #33 = (assert (bvult #31 #24))
                                                    #34 = (assert (= ((_ extract 31 31) (bvand (bvxor #24 (bvnot (_ bv85 32))) (bvxor #24 #31))) (_ bv1 1)))
                                                    #35 = (assert (= (parity_flag ((_ extract 7 0) #31)) (_ bv0 1)))
                                                    #36 = (assert (= ((_ extract 31 31) #31) (_ bv1 1)))
                                                    #37 = (assert (= #31 (_ bv0 32)))
[0] 0x400597    mov ecx, eax                        #38 = ((_ extract 31 0) #31)
[0] 0x400599    mov rdx, qword ptr [rip+0x200aa0]   #39 = (_ bv[illegible]196036 64)
[0] 0x4005a0    mov eax, dword ptr [rbp-0x4]        #40 = [illegible]
[0] 0x4005a3    cdqe
[0] 0x4005a5    add rax, rdx                        #41 = (bvadd #40 ((_ extract 63 0) #39))
                                                    #42 = (assert (= (_ bv16 64) (bvand (_ bv16 64) (bvxor #41 (bvxor #40 ((_ extract 63 0) #39))))))
                                                    #43 = (assert (bvult #41 #40))
                                                    #44 = (assert (= ((_ extract 63 63) (bvand (bvxor #40 (bvnot ((_ extract 63 0) #39))) (bvxor #40 #41))) (_ bv1 1)))
                                                    #45 = (assert (= (parity_flag ((_ extract 7 0) #41)) (_ bv0 1)))
                                                    #46 = (assert (= ((_ extract 63 63) #41) (_ bv1 1)))
                                                    #47 = (assert (= #41 (_ bv0 64)))
[0] 0x4005a8    movzx eax, byte ptr [rax]           #48 = ((_ zero_extend 24) (_ bv49 8))
[0] 0x4005ab    movsx eax, al                       #49 = ((_ sign_extend 24) ((_ extract 7 0) #48))
[0] 0x4005ae    cmp ecx, eax                        #50 = (bvsub #38 ((_ extract 31 0) #49))
                                                    #51 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor #50 (bvxor #38 ((_ extract 31 0) #49))))))
                                                    #52 = (assert (bvult #50 #38))
                                                    #53 = (assert (= ((_ extract 31 31) (bvand (bvxor #38 (bvnot ((_ extract 31 0) #49))) (bvxor #38 #50))) (_ bv1 1)))
                                                    #54 = (assert (= (parity_flag ((_ extract 7 0) #50)) (_ bv0 1)))
                                                    #55 = (assert (= ((_ extract 31 31) #50) (_ bv1 1)))
                                                    #56 = (assert (= #50 (_ bv0 32)))  ZF
[0] 0x4005b0    jz 0x4005b9
```

What user can control ?

Can I take both branches ?

Symbolic Execution Guided By The Taint Analysis



```
[0]  0x400588      add rax, rdx                        #15 = (bvadd #14 ((_ extract 63 0) #13))
                                                       #16 = (assert (= (_ bv16 64) (bvand (_ bv16 64) (bvxor #15 (bvxor #14 ((_ extract 63 0) #13))))))
                                                       #17 = (assert (bvult #15 #14))
                                                       #18 = (assert (= ((_ extract 63 63) (bvand (bvxor #14 (bvnot ((_ extract 63 0) #13))) (bvxor #14 #15))) (_ bv1 1)))
                                                       #19 = (assert (= (parity_flag ((_ extract 7 0) #15)) (_ bv0 1)))
                                                       #20 = (assert (= ((_ extract 63 63) #15) (_ bv1 1)))
                                                       #21 = (assert (= #15 (_ bv0 64)))
[0]  0x40058b      movzx eax, byte ptr [rax]           #22 = SymVar_0 Symbolic variable
[0]  0x40058e      movsx eax, al                       #23 = ((_ sign_extend 24) ((_ extract 7 0) #22))
[0]  0x400591      sub eax, 0x1                        #24 = (bvsub #23 (_ bv1 32))
                                                       #25 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor #24 (bvxor #23 (_ bv1 32))))))
                                                       #26 = (assert (bvult #24 #23))
                                                       #27 = (assert (= ((_ extract 31 31) (bvand (bvxor #23 (bvnot (_ bv1 32))) (bvxor #23 #24))) (_ bv1 1)))
                                                       #28 = (assert (= (parity_flag ((_ extract 7 0) #24)) (_ bv0 1)))
                                                       #29 = (assert (= ((_ extract 31 31) #24) (_ bv1 1)))
                                                       #30 = (assert (= #24 (_ bv0 32)))
[0]  0x400594      xor eax, 0x55                       #31 = (bvxor #24 (_ bv85 32))
                                                       #32 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor #31 (bvxor #24 (_ bv85 32))))))
                                                       #33 = (assert (bvult #31 #24))
                                                       #34 = (assert (= ((_ extract 31 31) (bvand (bvxor #24 (bvnot (_ bv85 32))) (bvxor #24 #31))) (_ bv1 1)))
                                                       #35 = (assert (= (parity_flag ((_ extract 7 0) #31)) (_ bv0 1)))
                                                       #36 = (assert (= ((_ extract 31 31) #31) (_ bv1 1)))
                                                       #37 = (assert (= #31 (_ bv0 32)))
[0]  0x400597      mov ecx, eax                        #38 = ((_ extract 31 0) #31)
[0]  0x400599      mov rdx, qword ptr [rip+0x200aa0]   #39 = (_ bv4196036 64)
[0]  0x4005a0      mov eax, dword ptr [rbp-0x4]        #40 = #5
[0]  0x4005a3      cdqe
[0]  0x4005a5      add rax, rdx                        #41 = (bvadd #40 ((_ extract 63 0) #39))
                                                       #42 = (assert (= (_ bv16 64) (bvand (_ bv16 64) (bvxor #41 (bvxor #40 ((_ extract 63 0) #39))))))
                                                       #43 = (assert (bvult #41 #40))
                                                       #44 = (assert (= ((_ extract 63 63) (bvand (bvxor #40 (bvnot ((_ extract 63 0) #39))) (bvxor #40 #41))) (_ bv1 1)))
                                                       #45 = (assert (= (parity_flag ((_ extract 7 0) #41)) (_ bv0 1)))
                                                       #46 = (assert (= ((_ extract 63 63) #41) (_ bv1 1)))
                                                       #47 = (assert (= #41 (_ bv0 64)))
[0]  0x4005a8      movzx eax, byte ptr [rax]           #48 = ((_ zero_extend 24) (_ bv49 8))
[0]  0x4005ab      movsx eax, al                       #49 = ((_ sign_extend 24) ((_ extract 7 0) #48))
[0]  0x4005ae      cmp ecx, eax                        #50 = (bvsub #38 ((_ extract 31 0) #49))
                                                       #51 = (assert (= (_ bv16 32) (bvand (_ bv16 32) (bvxor #50 (bvxor #38 ((_ extract 31 0) #49))))))
                                                       #52 = (assert (bvult #50 #38))
                                                       #53 = (assert (= ((_ extract 31 31) (bvand (bvxor #38 (bvnot ((_ extract 31 0) #49))) (bvxor #38 #50))) (_ bv1 1)))
                                                       #54 = (assert (= (parity_flag ((_ extract 7 0) #50)) (_ bv0 1)))
                                                       #55 = (assert (= ((_ extract 31 31) #50) (_ bv1 1)))
                                                       #56 = (assert (= #50 (_ bv0 32))) ZF
[0]  0x4005b0      jz 0x4005b9
```

Tainted memory area

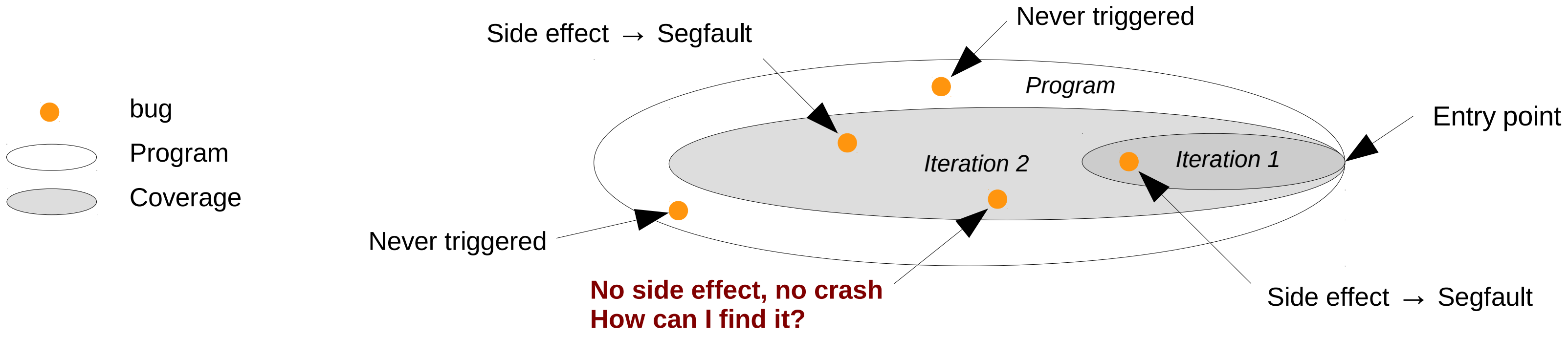
Spread the taint

ZF is controllable, so I can choose the branch

Few Words About Fuzzing

Few Words About Fuzzing

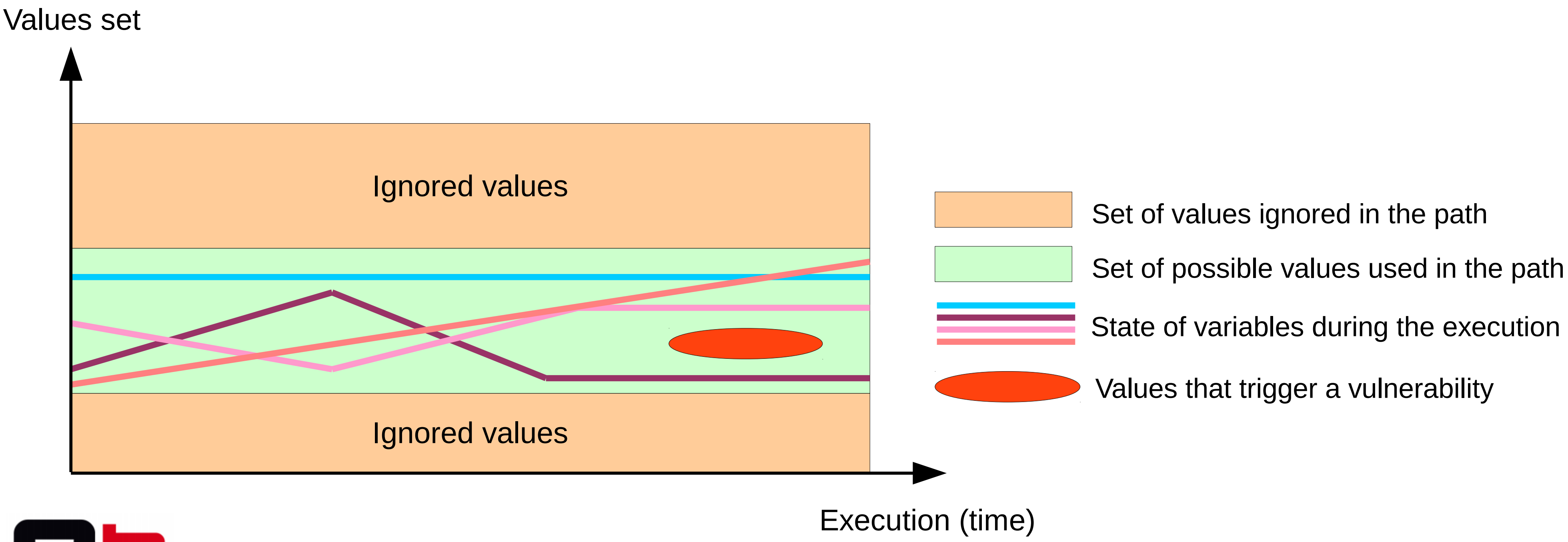
- Generally, the main objective is to cover a maximum of code by injecting different input samples and wait for a side effect like a segmentation fault
  - When a segmentation fault occurs, it means that we probably found a bug
  - The main issue is that some bugs do not make the program crash



Is Covering All The Paths Enough To Find Vulnerabilities?

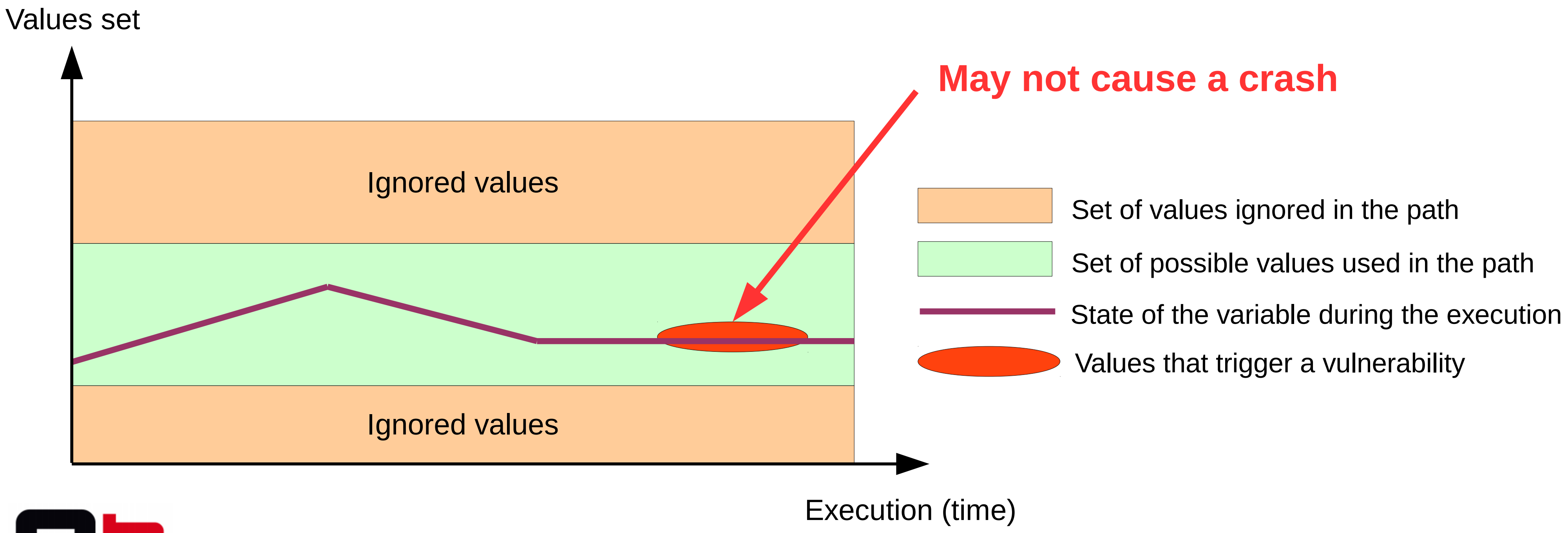
Is Covering All The Paths Enough To Find Vulnerabilities?

- **No!** Code coverage != State coverage. A variable can hold several values during the execution and some of these may not trigger any bug.
- We must generate all concrete values that a path can hold to cover all the possible states.
  - Imply a lot of overload in the worst case
- Below, a Cousot style graph which represents some possible states of a variable during the execution in a path.


Values set
Ignored values
Ignored values
Execution (time)
Set of values ignored in the path
Set of possible values used in the path
State of variables during the execution
Values that trigger a vulnerability

A Bug May Not Make The Program Crash

- Another important point is that a bug may not make the program crash
- Lots of fuzzers are based on the fact that a bug may have side effect like a SIGSEGV
  - That's why we must implement some behavior analysis to find bugs which does not make the program crash



Values set

May not cause a crash

Ignored values

Ignored values

Execution (time)

🟧 Set of values ignored in the path

🟩 Set of possible values used in the path

━ State of the variable during the execution

⬬ Values that trigger a vulnerability

OK, now that the introduction is over, let's start the talk!

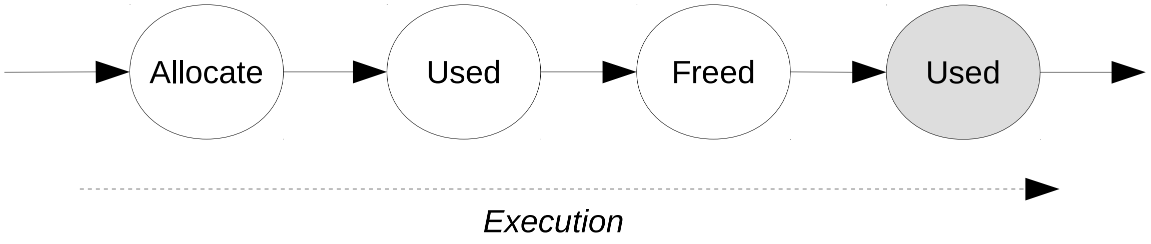
Objective Of This Talk

- Covering a function is not enough to find vulnerabilities
  - We must apply some “behavior analysis” at runtime using binary instrumentation
    - Really hard to build a generic analysis which find all kind of bugs
    - So, we must build specific analysis to find specific bugs
    - What kind of bugs we want to find? In this talk we will see how to find these kind of bugs:
      - Use-after-free
      - Overflow on heap / stack
      - Format string
      - {write, read}-what-where

Use-After-Free Analysis

Use-After-Free Analysis

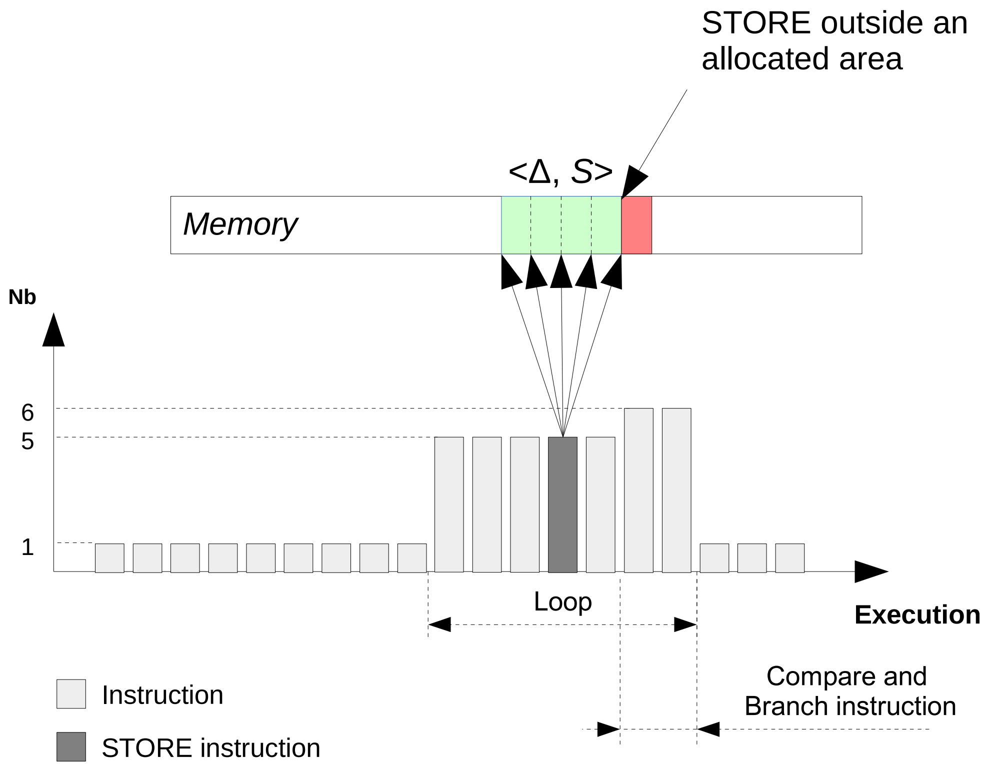
- An use-after-free mainly occurs when there is a LOAD/STORE on an already freed area
- First, we maintain an allocation map (TA) and a free map (TF) of $\langle\Delta, S\rangle$ items where Δ is the base address of the allocation and S its size. $\langle\Delta, S\rangle$ represents an area
  - Monitor the *malloc (*alloc)* function(s)
    - Δ is provided by the EAX register at the *malloc* return
    - S is provided by the argument of the *malloc* call
    - We add a new $\langle\Delta, S\rangle$ in TA and delete the $\langle\Delta, S\rangle$ in TF if it exist.
      - If $\Delta_{new} \in TF \land S_{new} \neq S_{old} \rightarrow \langle\Delta_{old}, S_{old}\rangle_{TF}$ is divided in two items where the first item will be in TA and the second item will still be in TF
  - When a *free* occurs we move the $\langle\Delta, S\rangle$ from TA to TF
  - When a LOAD/STORE occurs, we check if there is a Δ in TA or TF and applies these following rules:
    - If $\Delta \in TA \rightarrow$ valid memory access
    - If $\Delta \notin TA \land \Delta \notin TF \rightarrow$ invalid memory access
    - If $\Delta \notin TA \land \Delta \in TF \rightarrow$ use-after-free



Heap Overflow Analysis

Heap Overflow Analysis

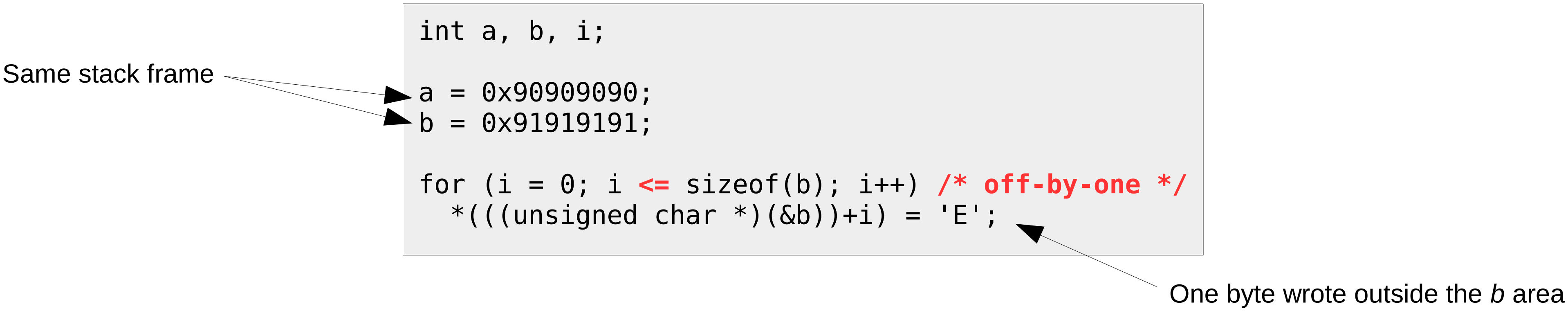
- Maintain an allocation map (*TA*) of <Δ,*S*> where Δ is the base address of the allocation and S its size (<Δ,*S*> represents an area)
- Monitor all STORE / LOAD and checkup if Δ ∈ TA
- We denote $\beta \in \mathbb{N}_0$ the iteration number and $<\Delta_\beta, S_\beta>$ the area description over each loop iteration
- When a loop applies a linear STORE, we apply these rules:
  - If $\beta \in \mathbb{N}^* \land \Delta_\beta = \Delta_{\beta-1} \land \beta < S \rightarrow$ OK
  - If $\beta \in \mathbb{N}^* \land \Delta_\beta \neq \Delta_{\beta-1} \land \beta >= S \rightarrow$ Heap overflow



Stack Overflow Analysis

Stack Overflow Analysis

- Two possible analysis:
  - Overflow outside the stack frame
  - Overflow between two variables of a same stack frame
    - We will focus on this analysis



Same stack frame

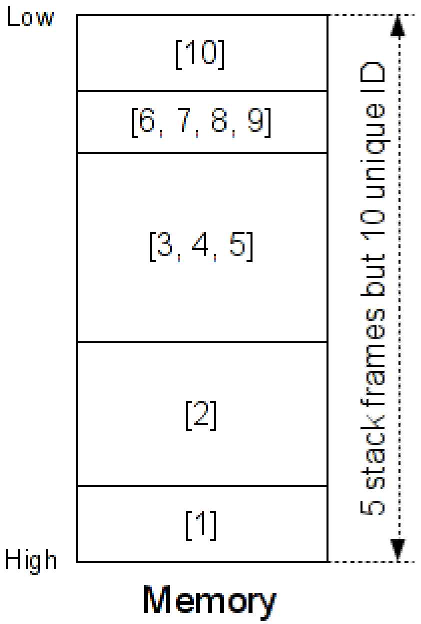
```
int a, b, i;

a = 0x90909090;
b = 0x91919191;

for (i = 0; i <= sizeof(b); i++) /* off-by-one */
  *(((unsigned char *)(&b))+i) = 'E';
```

One byte wrote outside the *b* area

How can I detect this off-by-one?

Stack Overflow Analysis

- We must:
  - Isolate all stack frames
    - Routine may be given by Pin or monitor all call/ret
    - Then, the area is given by the prologue
  - Find how many variables are in the stack frame
    - We use the *A-Locs* (Abstract Locations) methods from the Value-Set-Analysis paper [0]
  - Assign an area <ID,Δ,*S*> for each variable where ID ∈ ℕ is the unique stack frame id, Δ the base address of the variable and *S* the size of the variable
    - Like heap overflow analysis, check if there is a change of area <ID,Δ,*S*> during a linear STORE

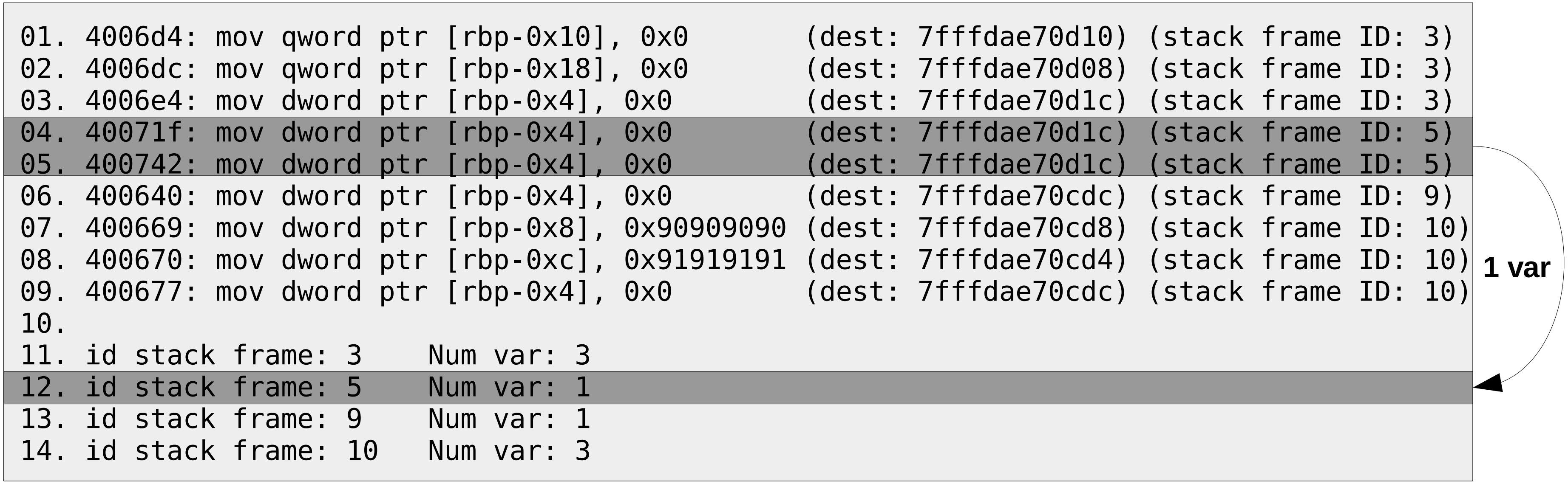


Each stack frame must have a unique ID

[0] Analyzing Memory Accesses in x86 Executables by Gogul Balakrishnan and Thomas Reps http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.1920&rep=rep1&type=pdf

Stack Overflow Analysis

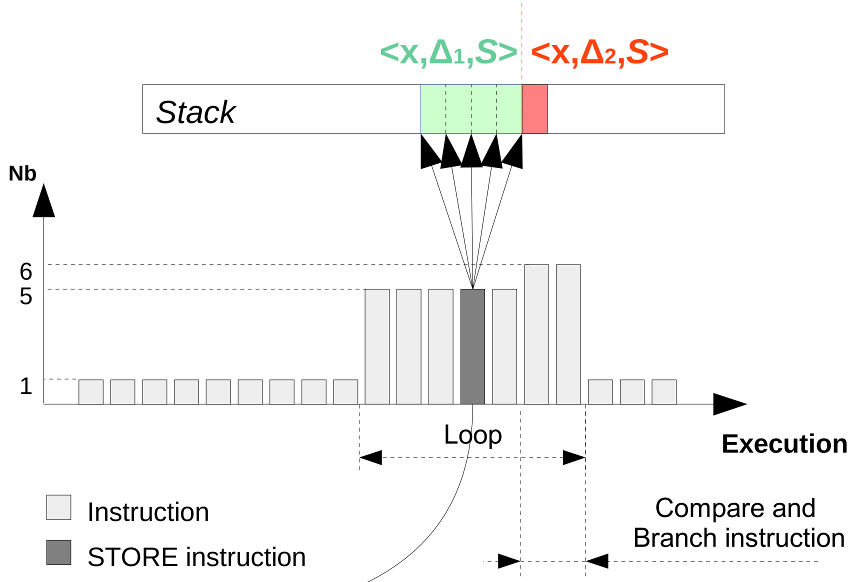
- If there is two assignments on a same <ID,Δ> → Interpret this as one variable area (take a look at line .04 and .05)
- However, if there is two assignments on <ID,Δ> and <ID,Δ+n> → Interpret this as two variables areas and we must refine the first area

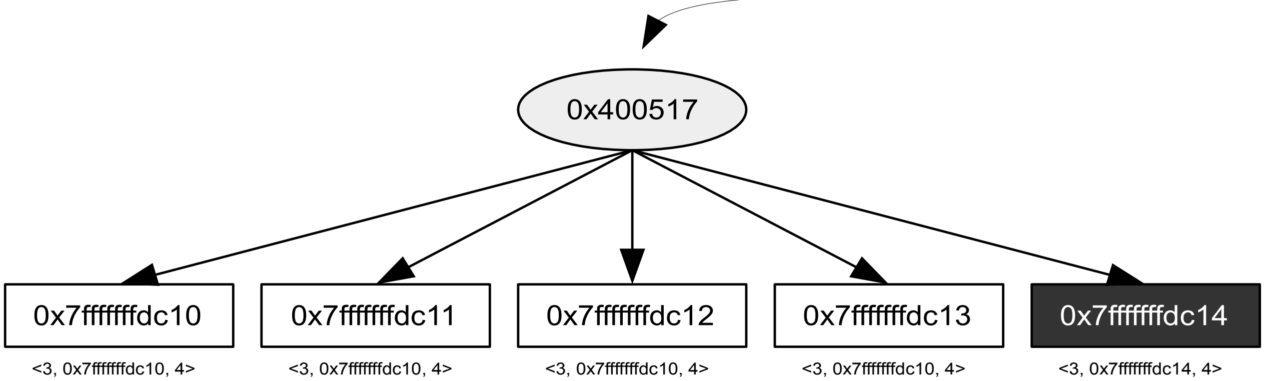


```
01. 4006d4: mov qword ptr [rbp-0x10], 0x0       (dest: 7fffdae70d10) (stack frame ID: 3)
02. 4006dc: mov qword ptr [rbp-0x18], 0x0       (dest: 7fffdae70d08) (stack frame ID: 3)
03. 4006e4: mov dword ptr [rbp-0x4], 0x0        (dest: 7fffdae70d1c) (stack frame ID: 3)
04. 40071f: mov dword ptr [rbp-0x4], 0x0        (dest: 7fffdae70d1c) (stack frame ID: 5)
05. 400742: mov dword ptr [rbp-0x4], 0x0        (dest: 7fffdae70d1c) (stack frame ID: 5)
06. 400640: mov dword ptr [rbp-0x4], 0x0        (dest: 7fffdae70cdc) (stack frame ID: 9)
07. 400669: mov dword ptr [rbp-0x8], 0x90909090 (dest: 7fffdae70cd8) (stack frame ID: 10)
08. 400670: mov dword ptr [rbp-0xc], 0x91919191 (dest: 7fffdae70cd4) (stack frame ID: 10)
09. 400677: mov dword ptr [rbp-0x4], 0x0        (dest: 7fffdae70cdc) (stack frame ID: 10)
10.
11. id stack frame: 3    Num var: 3
12. id stack frame: 5    Num var: 1
13. id stack frame: 9    Num var: 1
14. id stack frame: 10   Num var: 3
```

Stack Overflow Analysis

- As the heap overflow analysis, we must monitor each loop which applies a linear STORE and check if there is a change of area
- We denote $\beta \in \mathbb{N}_0$ the iteration number and $<ID_\beta,\Delta_\beta,S_\beta>$ the area description over each loop iteration
- If $\beta \in \mathbb{N}^* \wedge X_\beta = X_{\beta-1} \wedge \Delta_\beta = \Delta_{\beta-1} \rightarrow$ OK
- If $\beta \in \mathbb{N}^* \wedge X_\beta \neq X_{\beta-1} \rightarrow$ Overflow outside the stack frame
- If $\beta \in \mathbb{N}^* \wedge \Delta_\beta \neq \Delta_{\beta-1} \rightarrow$ Overflow between two variables





Stack Overflow Analysis

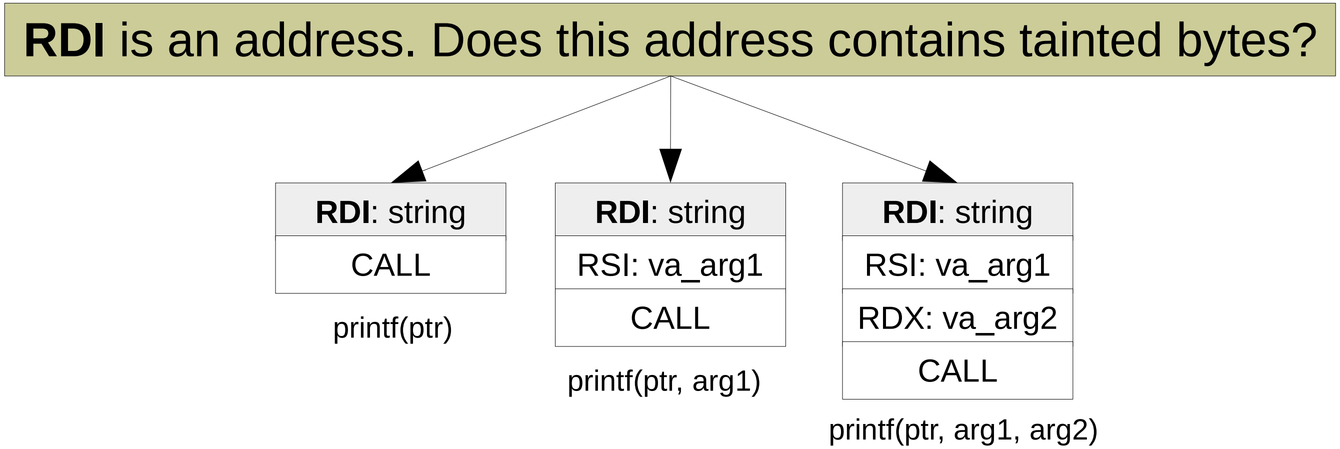
- The main weakness of this approach is the false positives in a specific case
- Lots of developers use the *memset* function to fill structures
- Let assume $A,B \in S$ such that $A \rightarrow <1,\Delta,4>$, $B \rightarrow <1,\Delta+4,4>$ and $S \rightarrow <1,\Delta,8>$
- When the *memset* function is applied, the analysis will detect a stack overflow from the area A to B

```
f(){
  struct s_foo S;

  memset(&S, 0, sizeof(struct s_foo));
  …
}
```

Format String Analysis

Format String Analysis

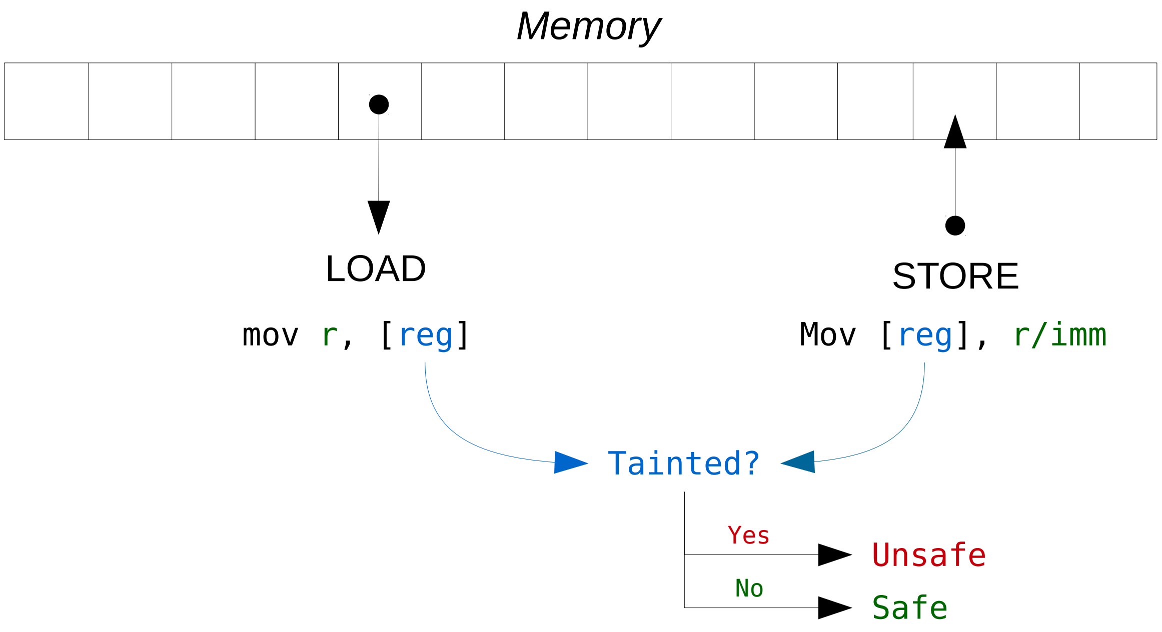
- This analysis mainly relies on the Taint Engine
- We monitor all functions susceptible to use string formats
  - Based on the calling convention, when a *CALL* occurs we get the function's arguments
  - If the first argument points on a tainted area, it means that the user can control the format string
    - Implies a format string bug



{Write, Read}-What-Where Analysis

{Write, Read}-What-Where Analysis

- This analysis mainly relies on the Taint Engine
- We must monitor all STORE/LOAD instruction and check if the destination/source is tainted
- **LOAD**:
  - If **reg** is tainted → *read-where* bug
- **STORE**:
  - If **reg** is tainted → *write-where* bug
  - If both operands are tainted (**reg**, **reg**) → *write-what-where* bug



Conclusion

Conclusion

- Covering a function or its state is not enough
- We must apply some behavior analysis during the execution to find bugs which do not make the program to crash
- Building a generic algorithm to find all kind of bugs is hard
  - We must build specific analysis to find specific bugs
- Lots of developments must be done before starting to work on the analysis part
  - Some analysis may reposes on the result of engines like the write-what-where or format string bugs which are based on the Taint Engine

The Triton Project

The Triton Project

- **Triton** provides some engines to improve analysis given by the Pin framework
- Developed in close collaboration with Florent Saudel
- Basically the Triton's engines are:
  - A Taint engine
    - Mainly used to know what variables and part of memory are controllable by the user at each program point
  - A Symbolic state engine
    - Mainly used to build symbolic expression for each register/memory at each program point
  - A Snapshot engine
    - Mainly used to replay traces directly in memory without running again the program
  - It also provides an Intermediate Representation in SMT2-LIB
    - Mainly used to solve equations with a theorem prover
  - It provides an interface with Z3 to solve symbolic expression like the paths condition
  - Then, it also applies all analysis described in this talk

- The **Triton** project will be detailed and released at **SSTIC 2015**

Final Words

Final Words

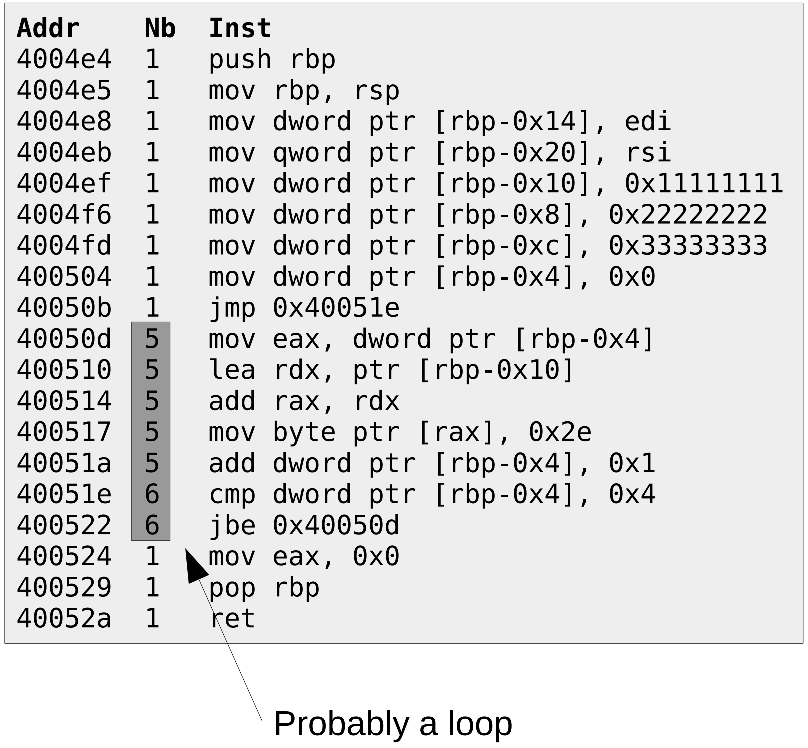
- Recap:
  - It possible to cover a function in memory using snapshot and dynamic symbolic execution
  - It possible to find bugs without side effect (like SIGSEGV)
    - Some bugs do not crash the program
  - Really **hard** to make generic algorithms to find all kind of bugs
    - We must build specific analysis for each bugs category
  - Use **symbolic execution** for code coverage and **dynamic behavior analysis** to find vulnerabilities in paths
    - Increase your chance to find bugs
  - Triton project announcement

Thanks For Your Attention
Question(s)?

- Contact
  - Mail: jsalwan@quarkslab.com
  - Twitter: @JonathanSalwan
- Thanks
  - I would like to thank the st'hack's staff and especially Florian Gaultier for the invitation and his hard work. Thanks also to Jean-Christophe Delaunay, Serge Guelton and Eloi Vanderbeken for the proofreading

Q&A - Problems Encountered

- **How did you detect the loops?**
- The main problem was for the stack and the heap overflow analysis
- Detect loops at runtime is a kind of challenge
  - Lots of papers apply a first pass of static analysis to build CFG and locates the loops
- Actually, what we did is a kind of “hack” and we did not found the good way yet...
- At runtime, we maintains a map of <Δ:*n*> where Δ is the address of the current instruction and $n \in \mathbb{N}^*$ the number of hits
- Generally a loop ends by a branch instruction and contains more than 1 hits (*n*)
- We apply some heuristics based on these “tricks”
- One of the problem with this, is that we can't detect a loop of 1 iteration. However, should we consider this as a loop?
- Even if we don't apply runtime analysis, all results of the trace can be stored in a database and further processed



```
Addr    Nb  Inst
4004e4  1   push rbp
4004e5  1   mov rbp, rsp
4004e8  1   mov dword ptr [rbp-0x14], edi
4004eb  1   mov qword ptr [rbp-0x20], rsi
4004ef  1   mov dword ptr [rbp-0x10], 0x11111111
4004f6  1   mov dword ptr [rbp-0x8], 0x22222222
4004fd  1   mov dword ptr [rbp-0xc], 0x33333333
400504  1   mov dword ptr [rbp-0x4], 0x0
40050b  1   jmp 0x40051e
40050d  5   mov eax, dword ptr [rbp-0x4]
400510  5   lea rdx, ptr [rbp-0x10]
400514  5   add rax, rdx
400517  5   mov byte ptr [rax], 0x2e
40051a  5   add dword ptr [rbp-0x4], 0x1
40051e  6   cmp dword ptr [rbp-0x4], 0x4
400522  6   jbe 0x40050d
400524  1   mov eax, 0x0
400529  1   pop rbp
40052a  1   ret
```

Probably a loop

Q&A - Benchmarks

- Does your analysis imply overheads?
- Yes, of course as a lots of tools
- By default DBI increases the time of the execution. Add others analysis and you got an overhead of 500% to 1000%
- For example, Triton processes 5,120,000 of expressions (with dataflow, SMT translation, symbolic state,...) around 140 seconds with 12Go of consumed RAM
  - Tested on a Lenovo x230 - i7-3520M CPU @ 2.90GHz
- Still unworkable on a whole binary as Firefox, chromium, etc...
  - That is why we target specific functions

Q&A – Future Work?

- Do you have some future ideas?
- Yes:
  - First of all: still working on the Triton design
  - Optimize the symbolic execution processing using a semantics' dictionary (poke Florent)
  - Optimize the memory usage caused by the execution using a custom remote allocation implemented as a kernel module
  - Use abstract interpretation in specific cases (poke Eloi)
  - Build a real runtime models checking
  - Search how can we parallelize the execution (fork at each branch?)
  - Manage the memory snapshotting using memory versioning
  - IDA plugin
  - And still lots of secret ideas :)