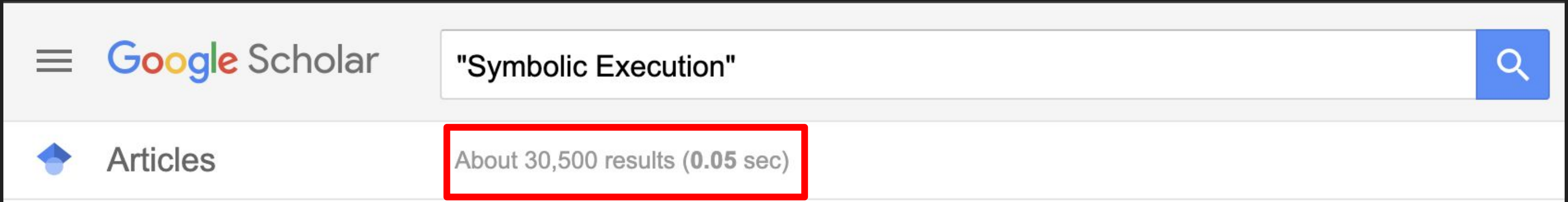
Symbolic Execution for Bug Hunting in Binaries

Fabio Gritti
(@degrigis)



Many many papers from an heterogeneous set of research groups!

Background

Bug Hunting

- Hunt for program states that are breaking the logic of the application and/or can be exploited to take control of the program itself.

- Hunting for bugs became the gold rush of our ages
  - Different hunter → different motivations

Symbolic Execution

Programming Languages

B. Wegbreit
Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

1975

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0)
            x = 2*(a+b);
    }
    assert(x-y != 0);
}
```

Are there any values for “a” and “b” for which the program breaks the assertion? 🤔

a=5, b=7

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0)
            x = 2*(a+b);
    }
    assert(x-y != 0); ✓
}
```

Are there any values for “a” and “b” for which the program breaks the assertion?

🤔

a=3, b=1

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0)
            x = 2*(a+b);
    }
    assert(x-y != 0);
}
```

Are there any values for “a” and “b” for which the program breaks the assertion?

🤔

Symbolic Execution

symbolic variables

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0)
            x = 2*(a+b);
    }
    assert(x-y != 0);
}
```

Are there any values for “a” and “b” for which the program breaks the assertion?

🤔

Symbolic Execution

symbolic variables

```
void foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3+x;
        if (b == 0)
            x = 2*(a+b);
    }
    assert(x-y != 0);
}
```

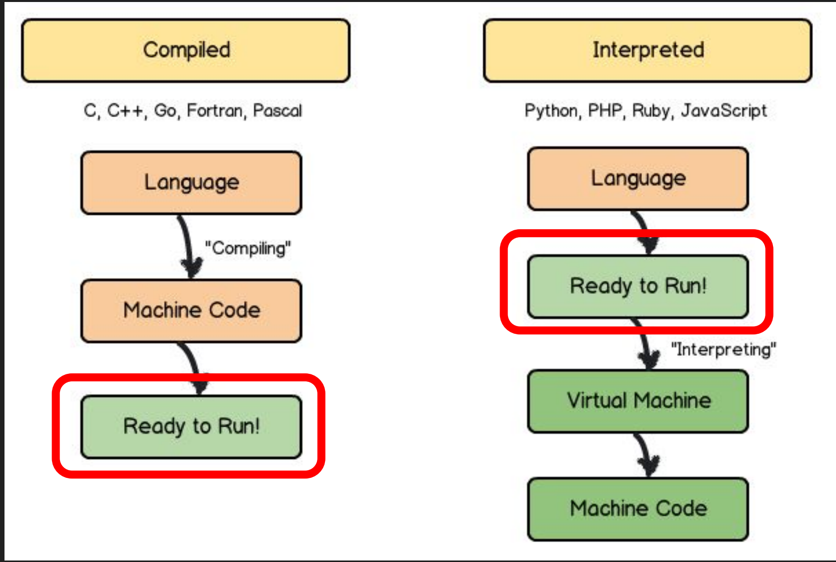
a=2,
b=0

Are there any values for “a” and “b” for which the program breaks the assertion?

🤔

Program Binary

- A binary is an artifact (i.e., a file) produced by a compiler after compiling a program's source code.
- Strict definition: binary contains assembly instructions executed by the CPU
- Loose Definition: binary can contain the IR interpreted by a VM
- Often, any debugging info/high level metadata coming from source code is unavailable in this artifact



SE for Bug Hunting in Binaries

- Use symbolic analysis over program binaries to hunt for as many bugs as we can!

- SE can be used for formal verification, but no false negative is tolerated
  - Verification is HARD and very tightly coupled with a strict specification

- Bug hunting → relax the soundness requirement (but we can still give some guarantees)

Motivation

Why Binaries?

- **Source language independent!**
  - Many high level languages are compiled down to same ASM
- **“What you fuzz is what you ship”** [62]
  - Truth lies in the binary!
- **Source code unavailable for specific domains**
  - Malware Analysis
  - Firmware Analysis

Why Symbolic Analysis?

- **Speed up the process of understanding a program's behaviors**
  - Which code is triggered by which input?
- **Can provide guarantees over some properties of the target program**
  - There exists no input for which the program reaches a specific state
- **Compensate limitations of black-box fuzzing**
  - Blackbox fuzzing blind-spots

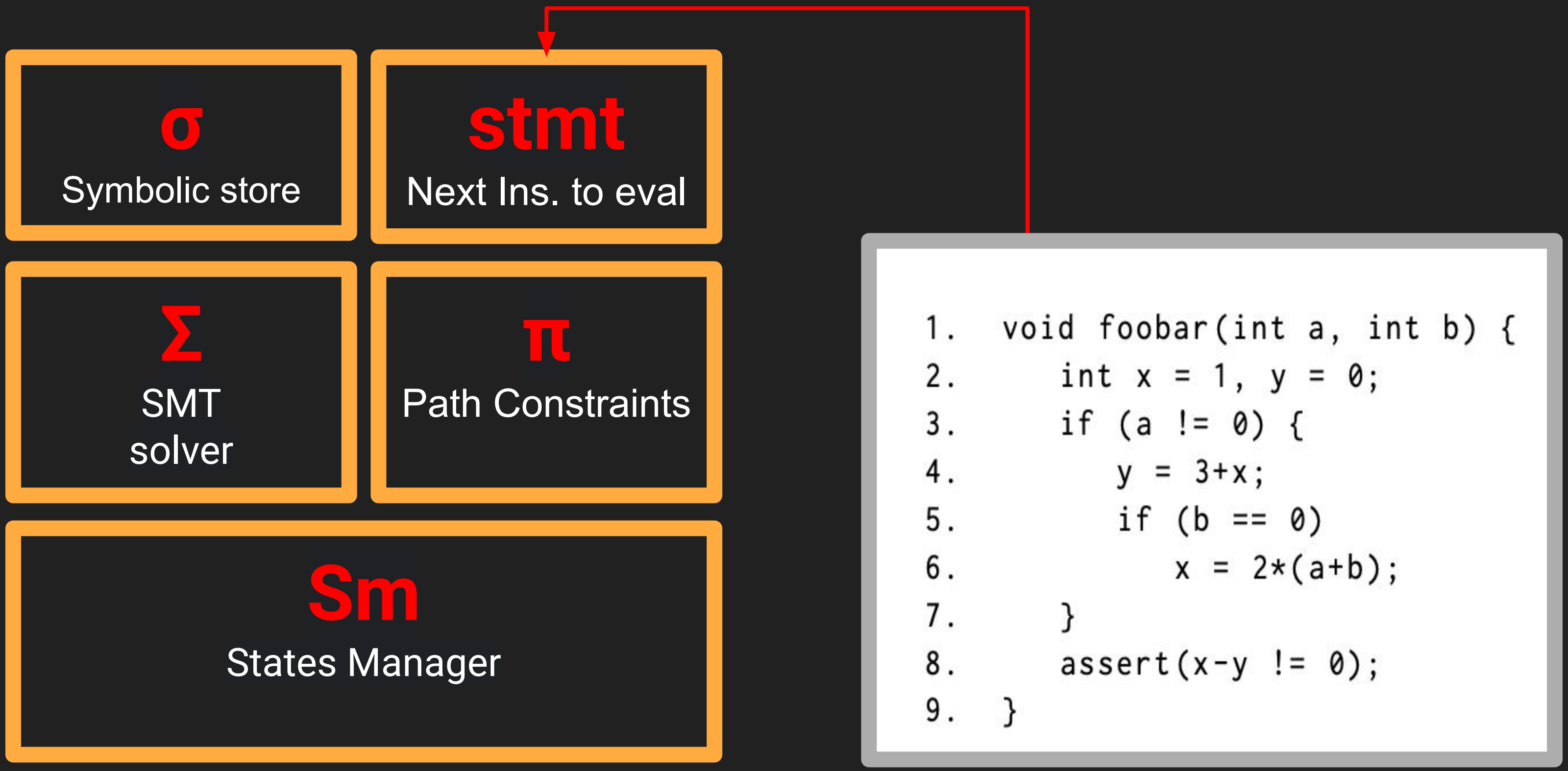
Why do we need this?

- Annual cost estimate for inadequate infrastructure for software testing was estimated to be ~$60 billion*

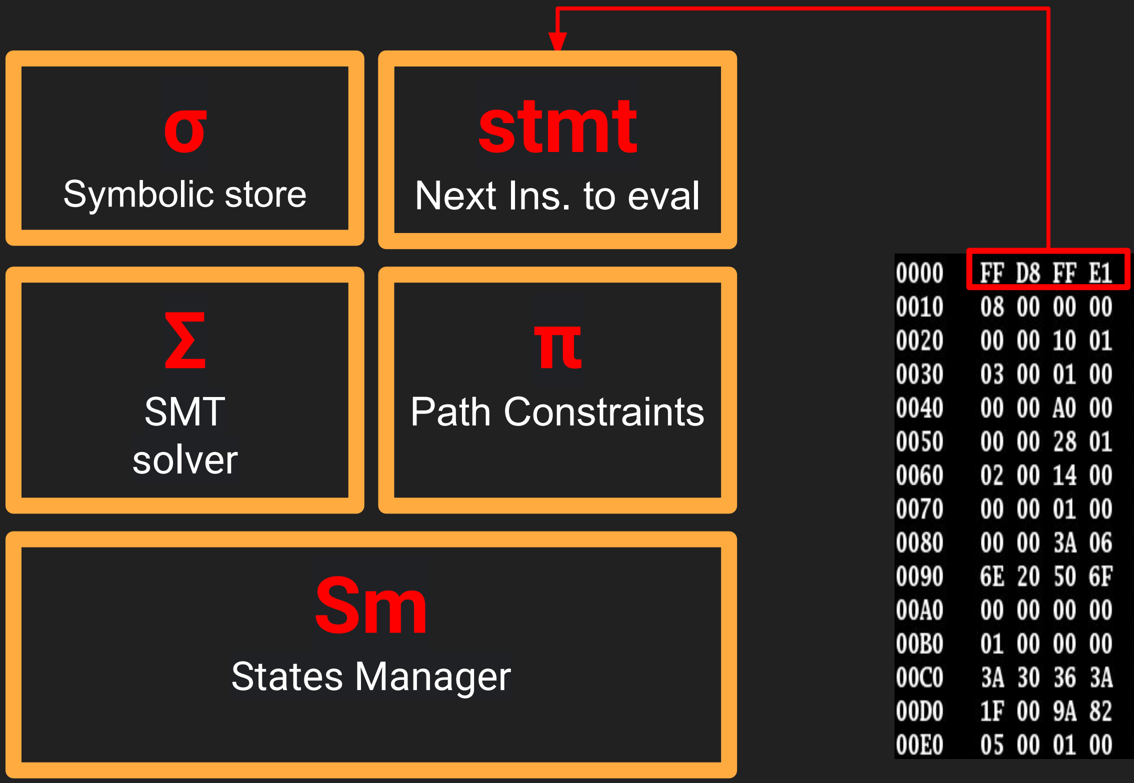
- The potential cost reduction from feasible infrastructure improvements was estimated to be ~$22.2 billion*

*Source: NIST report [66]

SE 101

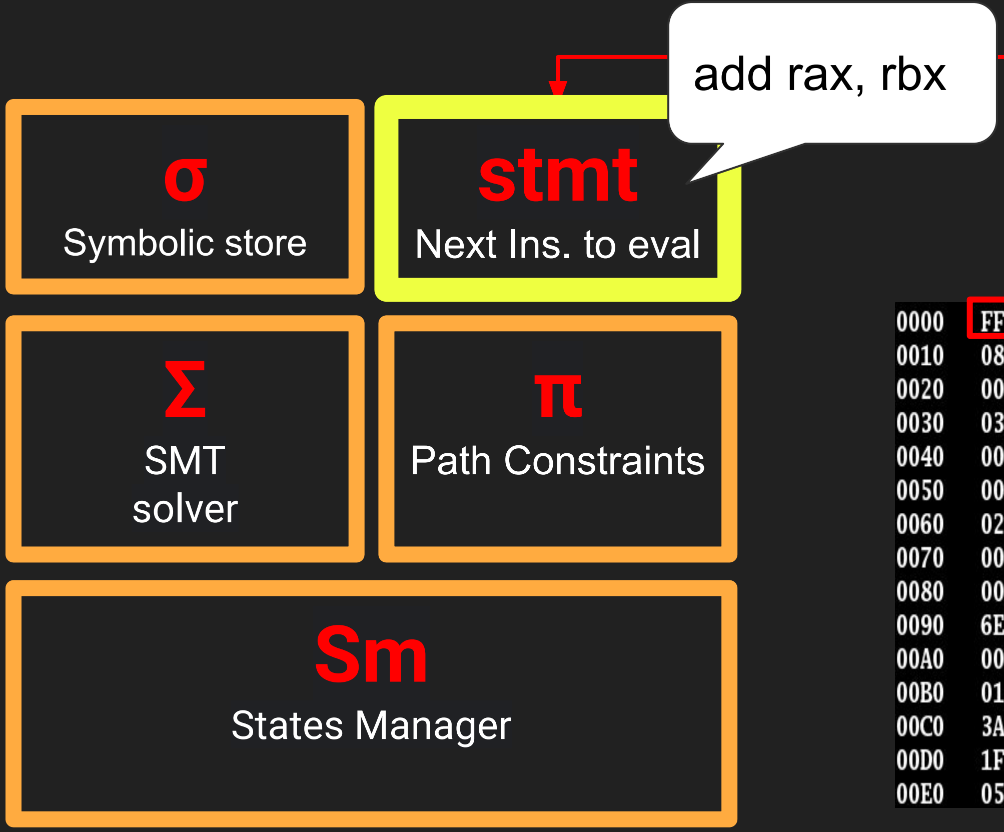


Target Program



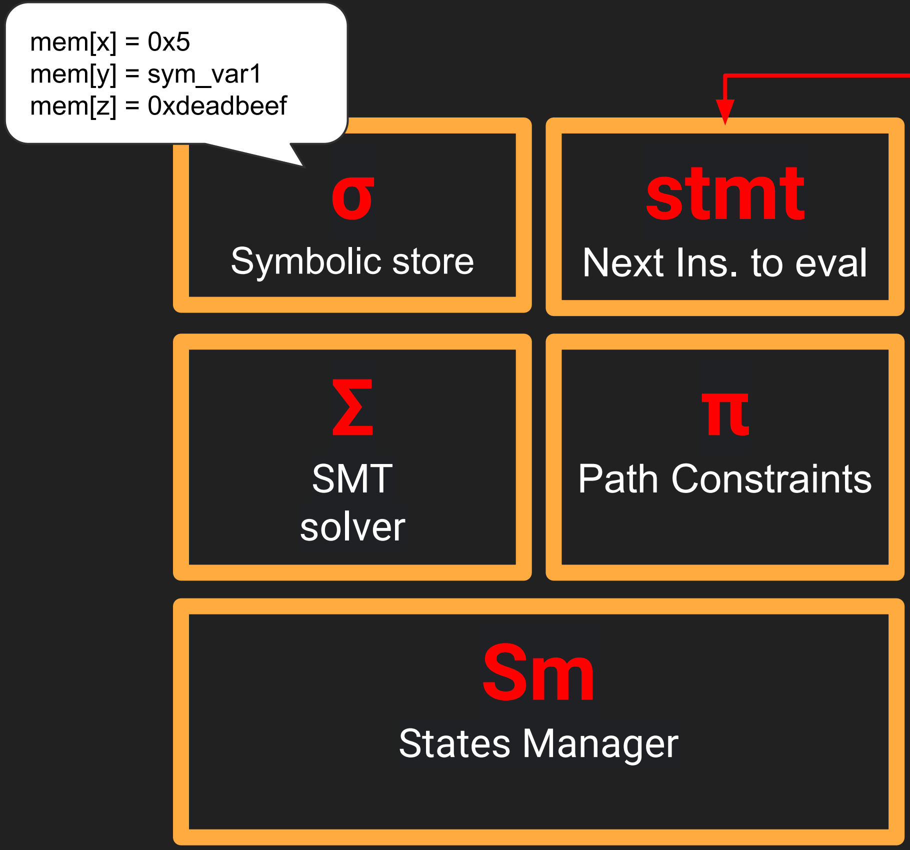
0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

Target Binary



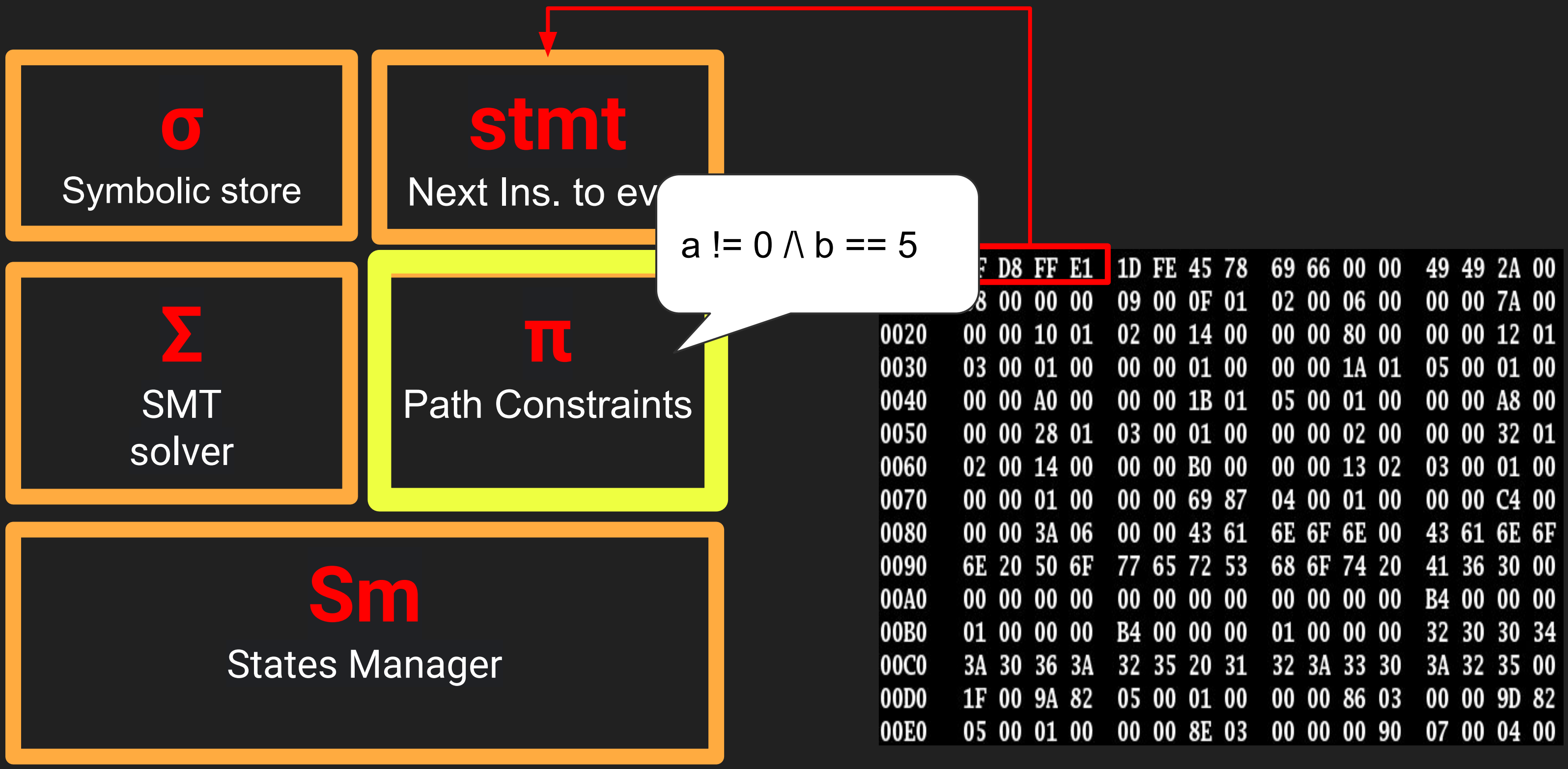
0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

Target Binary

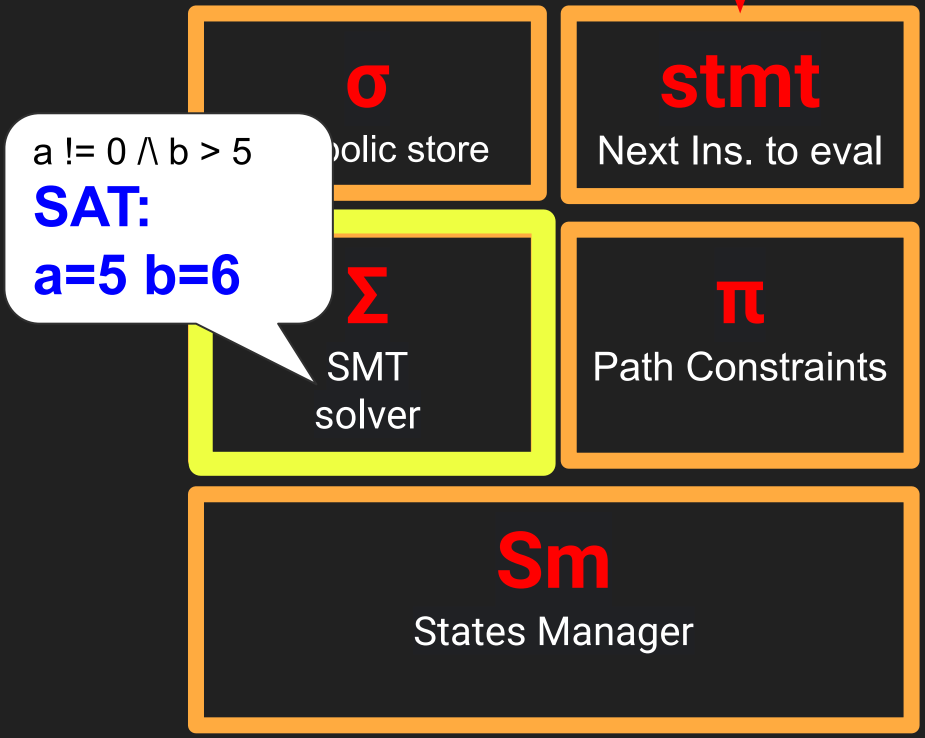


0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

Target Binary

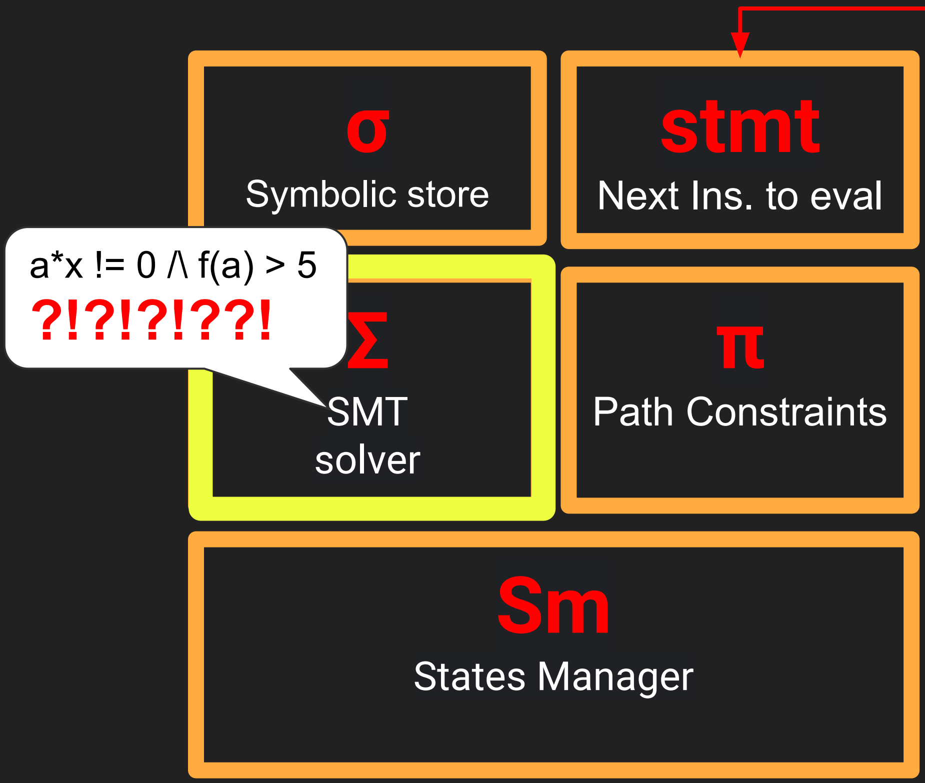


Target Binary



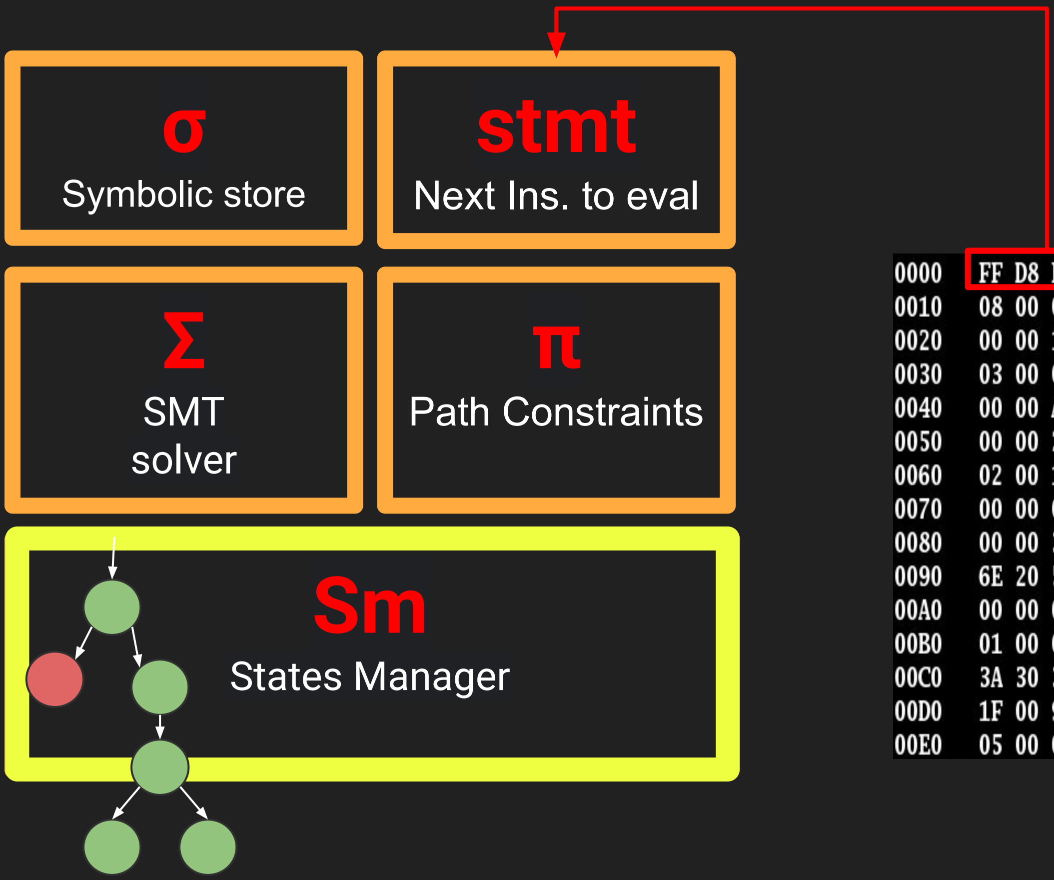
0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

Target Binary



0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

Target Binary



0000	FF D8 FF E1	1D FE 45 78	69 66 00 00	49 49 2A 00
0010	08 00 00 00	09 00 0F 01	02 00 06 00	00 00 7A 00
0020	00 00 10 01	02 00 14 00	00 00 80 00	00 00 12 01
0030	03 00 01 00	00 00 01 00	00 00 1A 01	05 00 01 00
0040	00 00 A0 00	00 00 1B 01	05 00 01 00	00 00 A8 00
0050	00 00 28 01	03 00 01 00	00 00 02 00	00 00 32 01
0060	02 00 14 00	00 00 B0 00	00 00 13 02	03 00 01 00
0070	00 00 01 00	00 00 69 87	04 00 01 00	00 00 C4 00
0080	00 00 3A 06	00 00 43 61	6E 6F 6E 00	43 61 6E 6F
0090	6E 20 50 6F	77 65 72 53	68 6F 74 20	41 36 30 00
00A0	00 00 00 00	00 00 00 00	00 00 00 00	B4 00 00 00
00B0	01 00 00 00	B4 00 00 00	01 00 00 00	32 30 30 34
00C0	3A 30 36 3A	32 35 20 31	32 3A 33 30	3A 32 35 00
00D0	1F 00 9A 82	05 00 01 00	00 00 86 03	00 00 9D 82
00E0	05 00 01 00	00 00 8E 03	00 00 00 90	07 00 04 00

Target Binary

symbolic variables

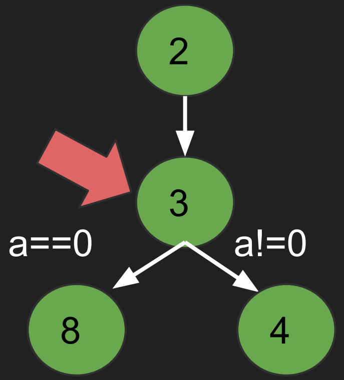
```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```

2

*States numbers correspond to code line (for simplicity we use program's source code here)

symbolic variables

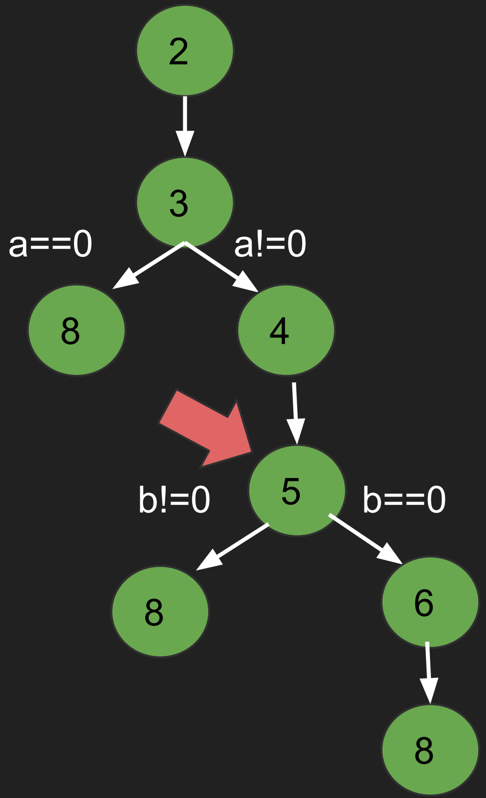
```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```



*States numbers correspond to code line (for simplicity we use program's source code here)

symbolic variables

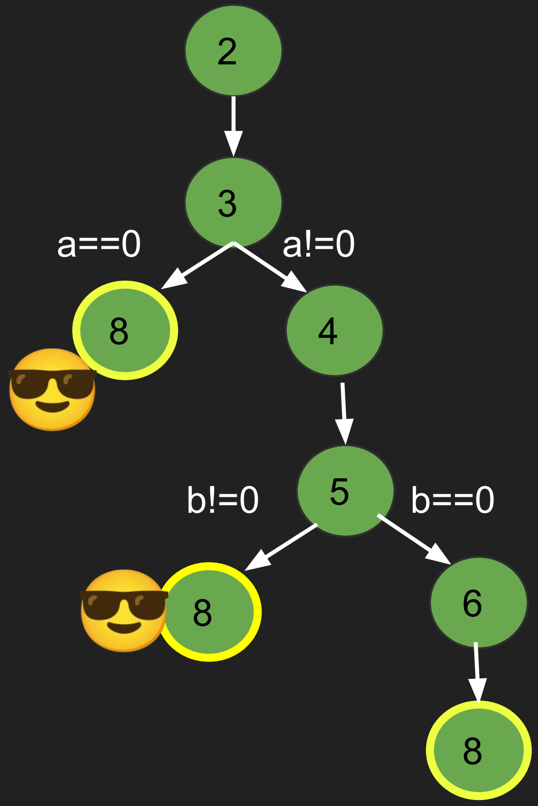
```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```



*States numbers correspond to code line (for simplicity we use program’s source code here)

symbolic variables

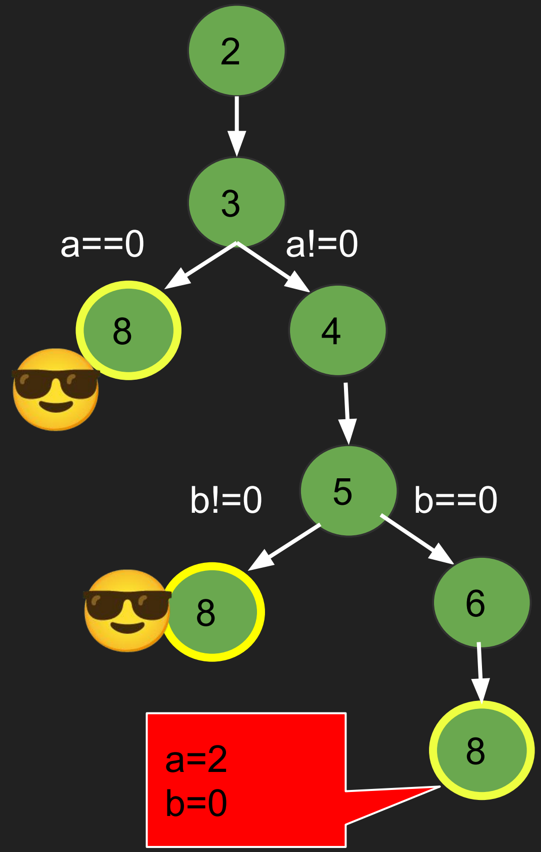
```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```



*States numbers correspond to code line (for simplicity we use program's source code here)

symbolic variables

```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```



*States numbers correspond to code line (for simplicity we use program's source code here)

Research Areas

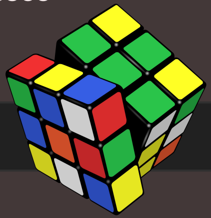
Research Challenges

State Explosion

- Memory exhaustion of the system
- No meaningful analysis progresses

Execution Performances

- Not enough coverage before timeout
- No meaningful analysis progresses

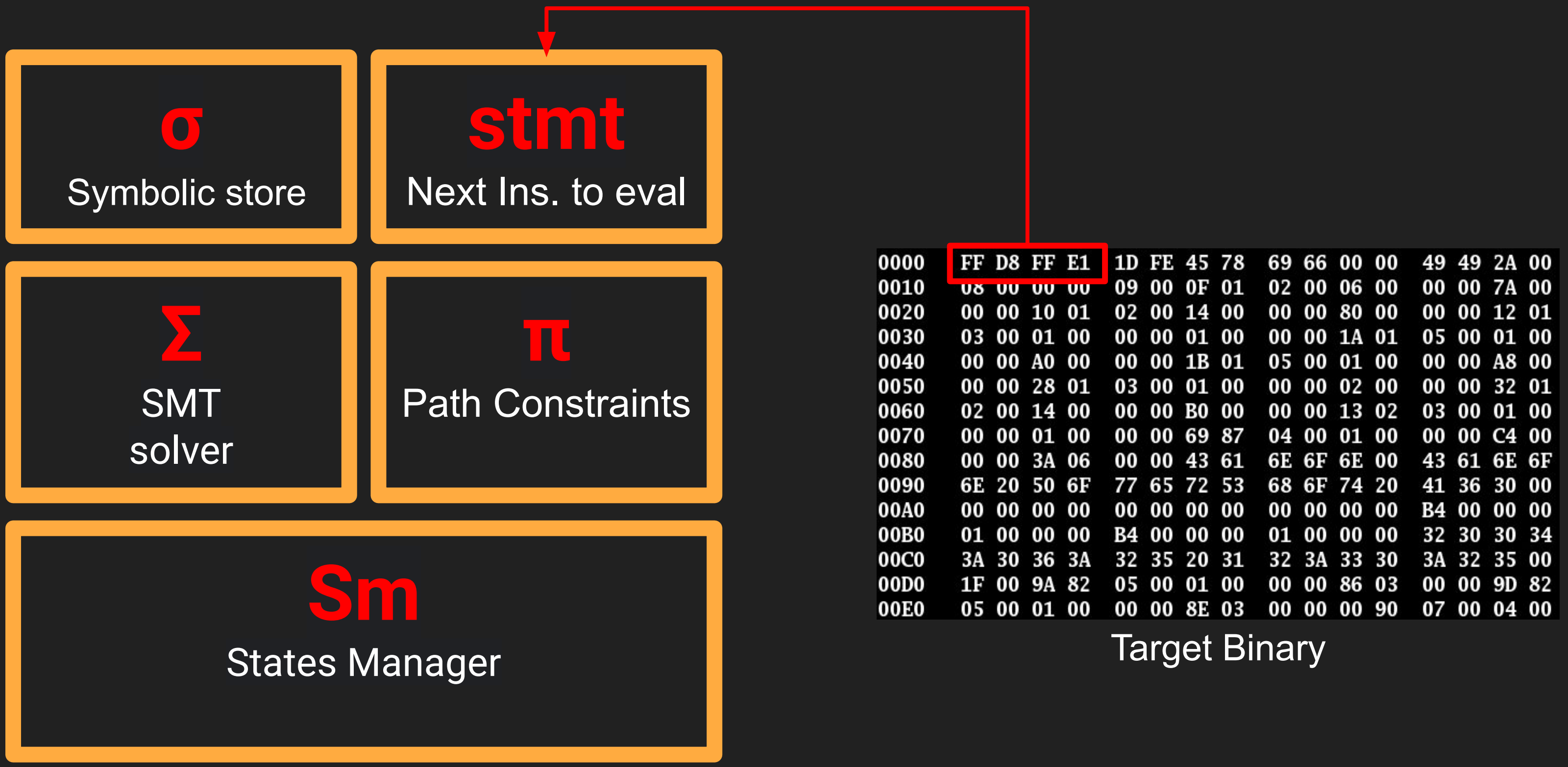


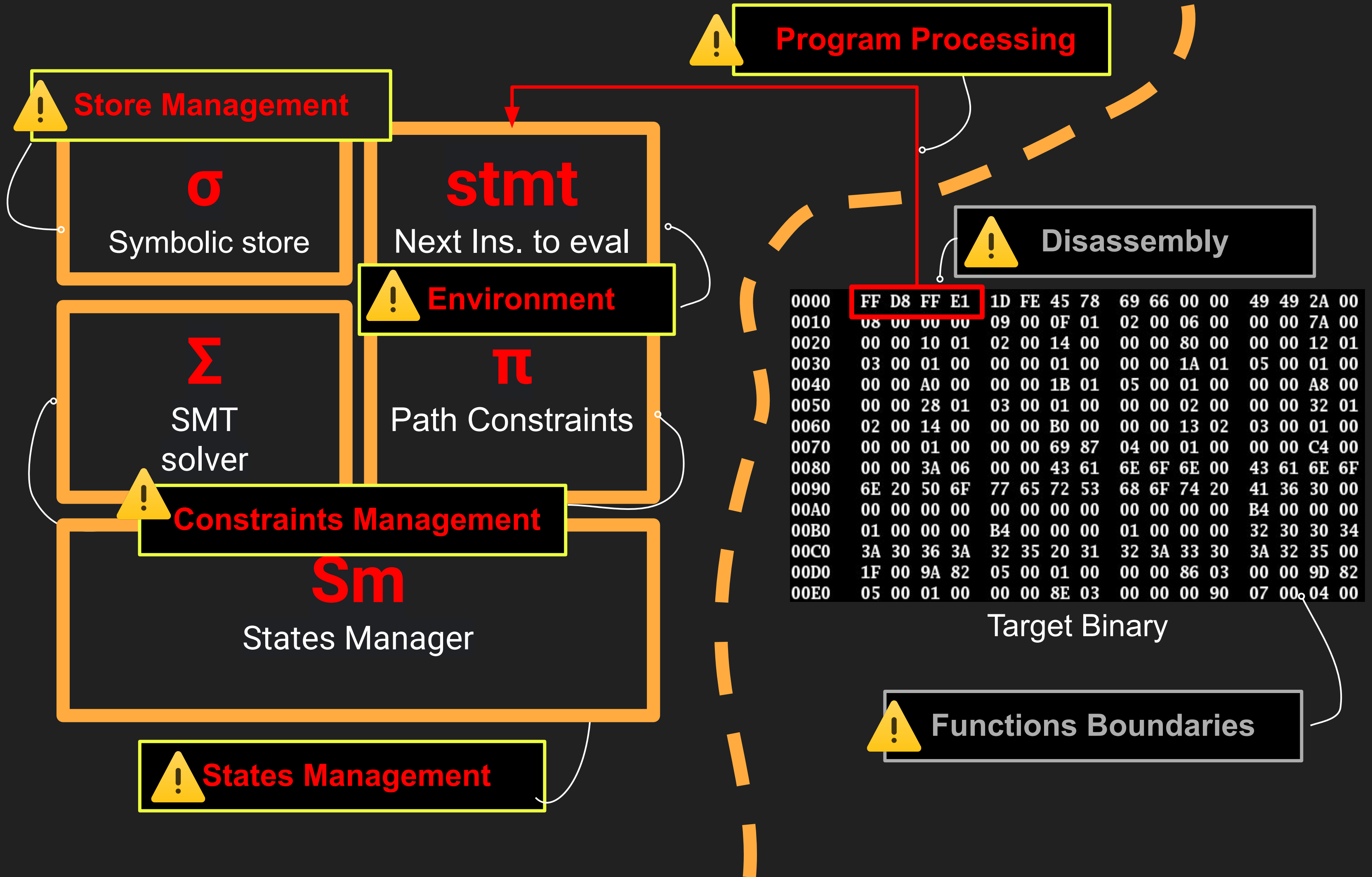
Imprecise Analysis

- Too many false positives/negatives
- Invalid/unusable results

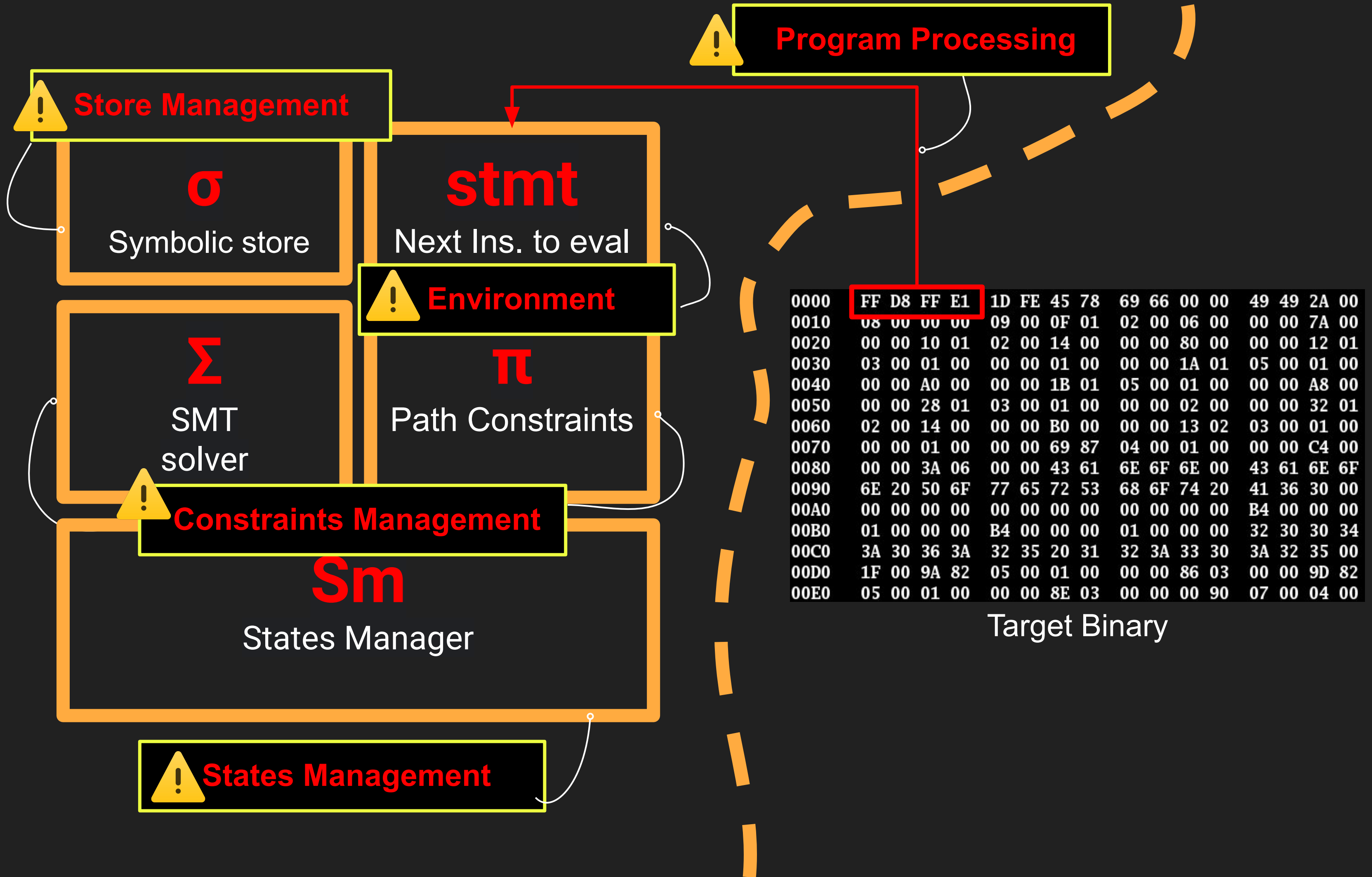
Constraints Solving

- Solving time timeout
- Unfeasible constraints

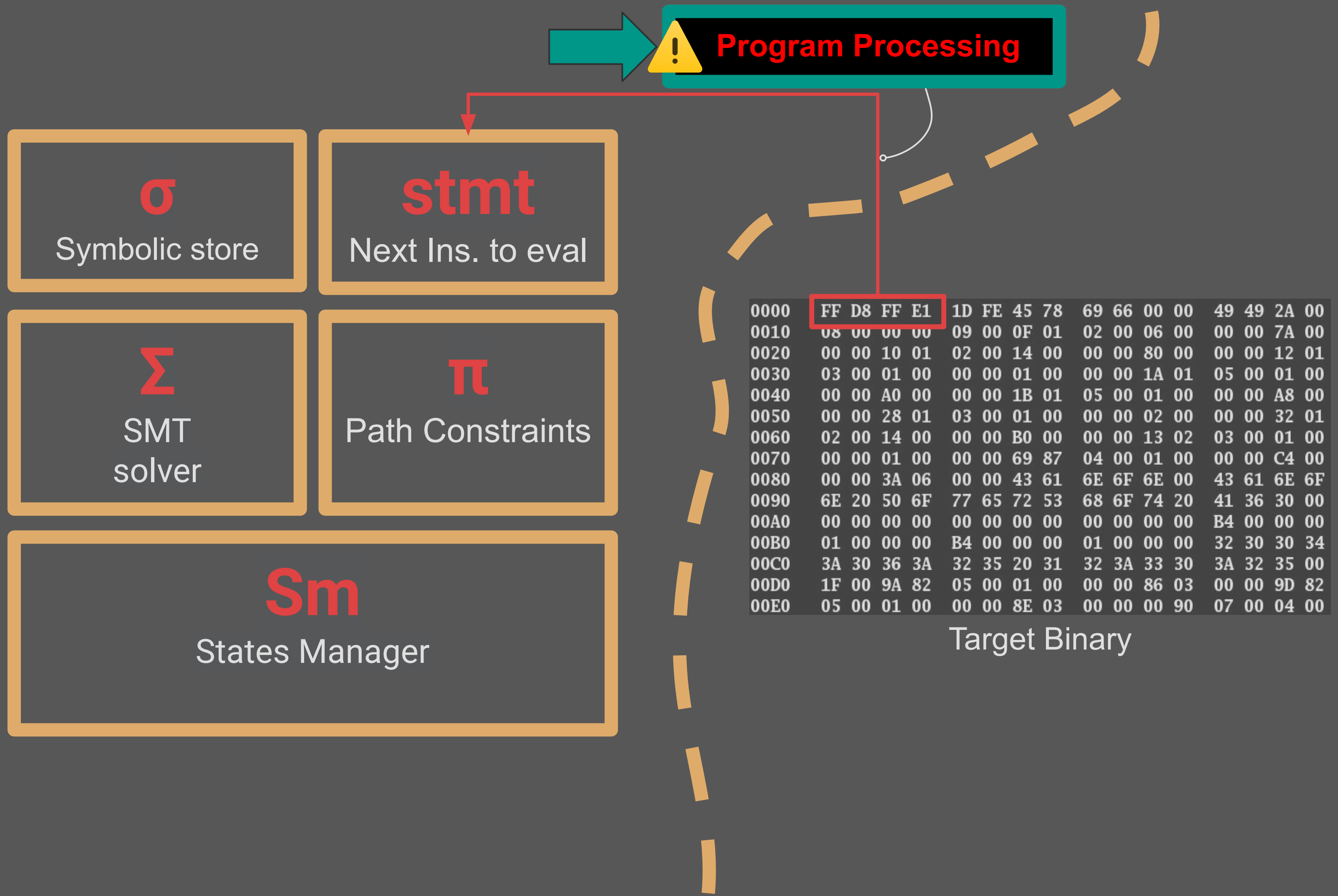

σ
Symbolic store
stmt
Next Ins. to eval
Σ
SMT solver
π
Path Constraints
Sm
States Manager
0000 FF D8 FF E1 1D FE 45 78 69 66 00 00 49 49 2A 00
0010 08 00 00 00 09 00 0F 01 02 00 06 00 00 00 7A 00
0020 00 00 10 01 02 00 14 00 00 00 80 00 00 00 12 01
0030 03 00 01 00 00 00 01 00 00 00 1A 01 05 00 01 00
0040 00 00 A0 00 00 00 1B 01 05 00 01 00 00 00 A8 00
0050 00 00 28 01 03 00 01 00 00 00 02 00 00 00 32 01
0060 02 00 14 00 00 00 B0 00 00 00 13 02 03 00 01 00
0070 00 00 01 00 00 00 69 87 04 00 01 00 00 00 C4 00
0080 00 00 3A 06 00 00 43 61 6E 6F 6E 00 43 61 6E 6F
0090 6E 20 50 6F 77 65 72 53 68 6F 74 20 41 36 30 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 B4 00 00 00
00B0 01 00 00 00 B4 00 00 00 01 00 00 00 32 30 30 34
00C0 3A 30 36 3A 32 35 20 31 32 3A 33 30 3A 32 35 00
00D0 1F 00 9A 82 05 00 01 00 00 00 86 03 00 00 9D 82
00E0 05 00 01 00 00 00 8E 03 00 00 00 90 07 00 04 00
Target Binary



Program Processing
Store Management
σ
Symbolic store
stmt
Next Ins. to eval
Environment
Σ
SMT solver
π
Path Constraints
Constraints Management
Sm
States Manager
States Management
Disassembly
0000 FF D8 FF E1 1D FE 45 78 69 66 00 00 49 49 2A 00
0010 08 00 00 00 09 00 0F 01 02 00 06 00 00 00 7A 00
0020 00 00 10 01 02 00 14 00 00 00 80 00 00 00 12 01
0030 03 00 01 00 00 00 01 00 00 00 1A 01 05 00 01 00
0040 00 00 A0 00 00 00 1B 01 05 00 01 00 00 00 A8 00
0050 00 00 28 01 03 00 01 00 00 00 02 00 00 00 32 01
0060 02 00 14 00 00 00 B0 00 00 00 13 02 03 00 01 00
0070 00 00 01 00 00 00 69 87 04 00 01 00 00 00 C4 00
0080 00 00 3A 06 00 00 43 61 6E 6F 6E 00 43 61 6E 6F
0090 6E 20 50 6F 77 65 72 53 68 6F 74 20 41 36 30 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 B4 00 00 00
00B0 01 00 00 00 B4 00 00 00 01 00 00 00 32 30 30 34
00C0 3A 30 36 3A 32 35 20 31 32 3A 33 30 3A 32 35 00
00D0 1F 00 9A 82 05 00 01 00 00 00 86 03 00 00 9D 82
00E0 05 00 01 00 00 00 8E 03 00 00 00 90 07 00 04 00
Target Binary
Functions Boundaries



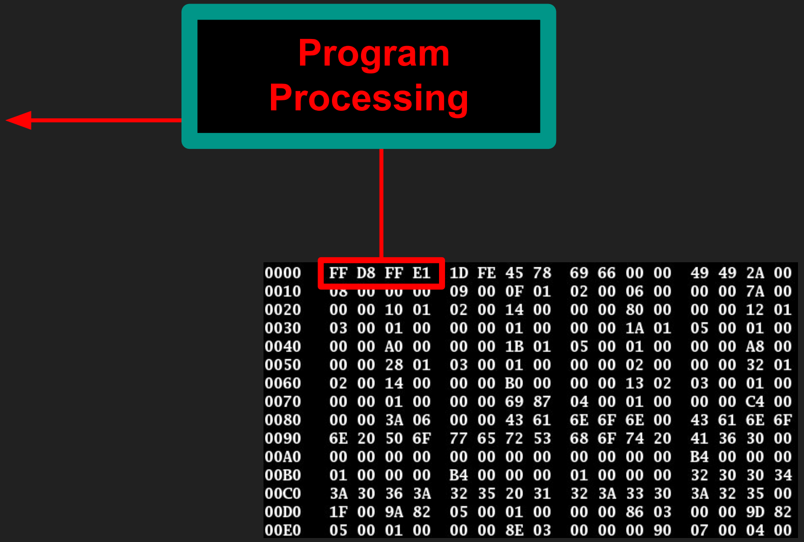
Program Processing
Store Management
σ
Symbolic store
stmt
Next Ins. to eval
Environment
Σ
SMT solver
π
Path Constraints
Constraints Management
Sm
States Manager
States Management
0000 FF D8 FF E1 1D FE 45 78 69 66 00 00 49 49 2A 00
0010 08 00 00 00 09 00 0F 01 02 00 06 00 00 00 7A 00
0020 00 00 10 01 02 00 14 00 00 00 80 00 00 00 12 01
0030 03 00 01 00 00 00 01 00 00 00 1A 01 05 00 01 00
0040 00 00 A0 00 00 00 1B 01 05 00 01 00 00 00 A8 00
0050 00 00 28 01 03 00 01 00 00 00 02 00 00 00 32 01
0060 02 00 14 00 00 00 B0 00 00 00 13 02 03 00 01 00
0070 00 00 01 00 00 00 69 87 04 00 01 00 00 00 C4 00
0080 00 00 3A 06 00 00 43 61 6E 6F 6E 00 43 61 6E 6F
0090 6E 20 50 6F 77 65 72 53 68 6F 74 20 41 36 30 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 B4 00 00 00
00B0 01 00 00 00 B4 00 00 00 01 00 00 00 32 30 30 34
00C0 3A 30 36 3A 32 35 20 31 32 3A 33 30 3A 32 35 00
00D0 1F 00 9A 82 05 00 01 00 00 00 86 03 00 00 9D 82
00E0 05 00 01 00 00 00 8E 03 00 00 00 90 07 00 04 00
Target Binary


Program Processing
σ
Symbolic store
stmt
Next Ins. to eval
Σ
SMT solver
π
Path Constraints
Sm
States Manager
0000 FF D8 FF E1 1D FE 45 78 69 66 00 00 49 49 2A 00
0010 08 00 00 00 09 00 0F 01 02 00 06 00 00 00 7A 00
0020 00 00 10 01 02 00 14 00 00 00 80 00 00 00 12 01
0030 03 00 01 00 00 00 01 00 00 00 1A 01 05 00 01 00
0040 00 00 A0 00 00 00 1B 01 05 00 01 00 00 00 A8 00
0050 00 00 28 01 03 00 01 00 00 00 02 00 00 00 32 01
0060 02 00 14 00 00 00 B0 00 00 00 13 02 03 00 01 00
0070 00 00 01 00 00 00 69 87 04 00 01 00 00 00 C4 00
0080 00 00 3A 06 00 00 43 61 6E 6F 6E 00 43 61 6E 6F
0090 6E 20 50 6F 77 65 72 53 68 6F 74 20 41 36 30 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 B4 00 00 00
00B0 01 00 00 00 B4 00 00 00 01 00 00 00 32 30 30 34
00C0 3A 30 36 3A 32 35 20 31 32 3A 33 30 3A 32 35 00
00D0 1F 00 9A 82 05 00 01 00 00 00 86 03 00 00 9D 82
00E0 05 00 01 00 00 00 8E 03 00 00 00 90 07 00 04 00
Target Binary

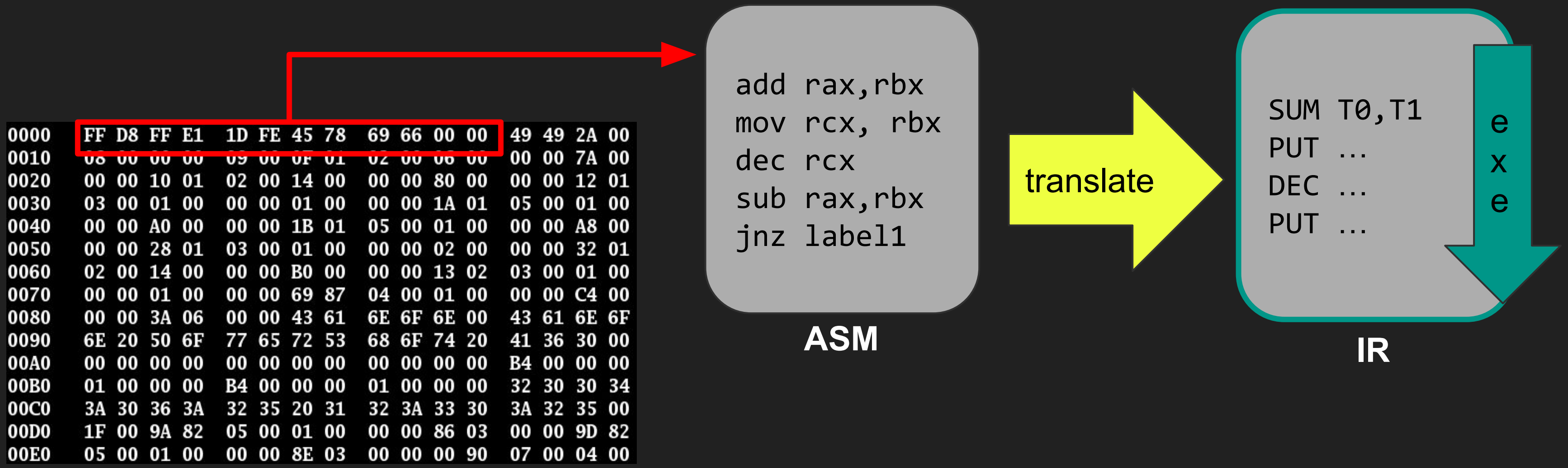
Program Processing

- How the symbolic execution engine is supposed to process the instructions contained in the target binary?

- Classic Approaches:
  - IR-Based Symbolic Execution
  - IR-Less Symbolic Execution



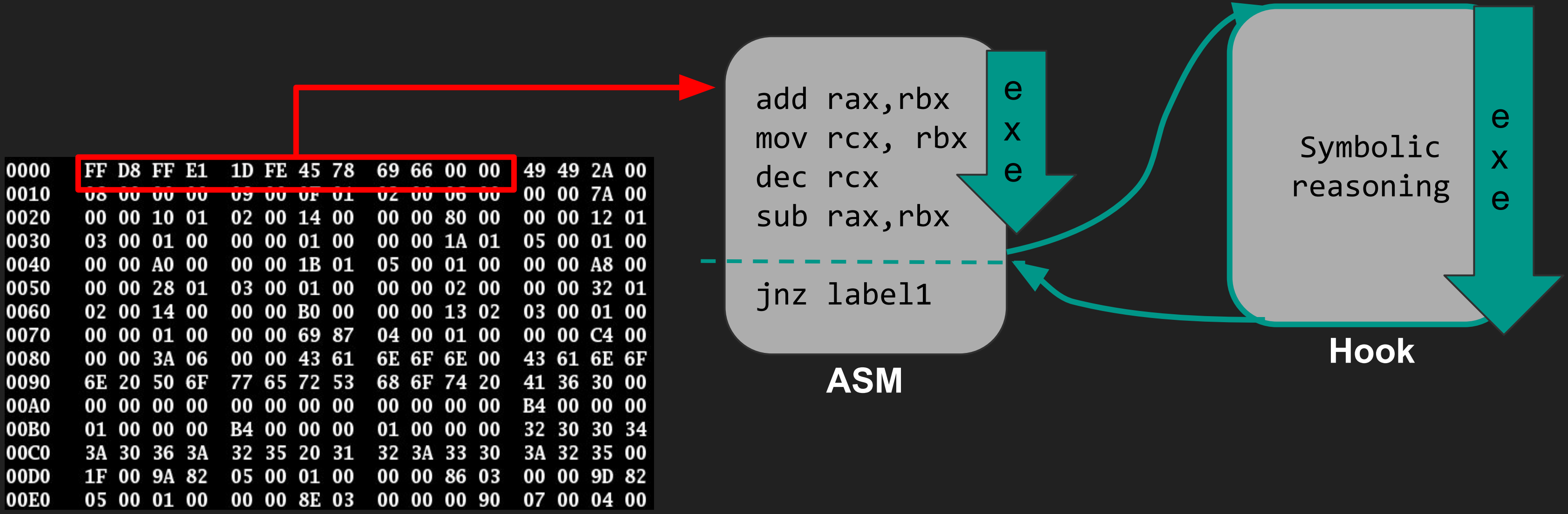
Binary Processing



IR-Based Symbolic Execution

(most popular)

Binary Processing



IR-Less Symbolic Execution

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	
Architecture agnostic	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	
Architecture agnostic	
Easier queries to the solver[39]	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	
Architecture agnostic	
Easier queries to the solver[39]	
Less robust (Unsupported native ins => Fatal error)	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	
Architecture agnostic	
Easier queries to the solver[39]	
Less robust (Unsupported native ins => Fatal error)	
Poor execution performance[39]	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	**Hard to implement** (thousands of instructions)
Architecture agnostic	
Easier queries to the solver[39]	
Less robust (Unsupported native ins => Fatal error)	
Poor execution performance[39]	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	Hard to implement (thousands of instructions)
Architecture agnostic	Architecture-dependent (not portable!)
Easier queries to the solver[39]	
Less robust (Unsupported native ins => Fatal error)	
Poor execution performance[39]	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	**Hard to implement** (thousands of instructions)
Architecture agnostic	**Architecture-dependent (not portable!)**
Easier queries to the solver[39]	**Harder queries to the solver**[39]
Less robust (Unsupported native ins => Fatal error)	
Poor execution performance[39]	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	**Hard to implement** (thousands of instructions)
Architecture agnostic	**Architecture-dependent (not portable!)**
Easier queries to the solver[39]	**Harder queries to the solver**[39]
Less robust (Unsupported native ins => Fatal error)	**More Robust** (Unsupported ins => Just execute concretely)
Poor execution performance[39]	

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
Easier to implement (Small number of high-level instructions)	**Hard to implement** (thousands of instructions)
Architecture agnostic	**Architecture-dependent (not portable!)**
Easier queries to the solver[39]	**Harder queries to the solver**[39]
Less robust (Unsupported native ins => Fatal error)	**More Robust** (Unsupported ins => Just execute concretely)
Poor execution performance[39]	**Good execution performance**[39]

KLEE[7], S2E[10], angr[25]

QSYM[34], SAGE[62], Triton[65]

Binary Processing

IR-Based Symbolic Execution	IR-Less Symbolic Execution
KLEE[7], S2E[10], angr[25]	QSYM[34], SAGE[62], Triton[65]

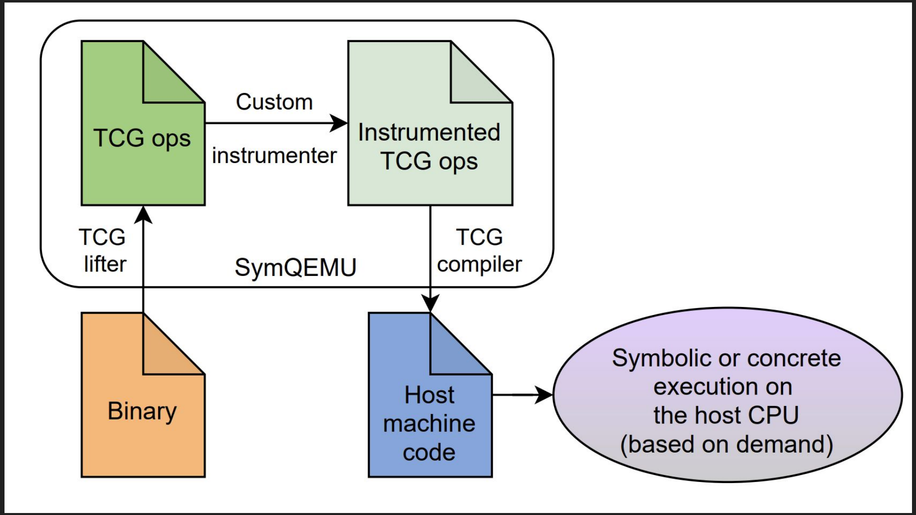
Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and its Generation

Sebastian Poeplau
Aurélien Francillon
sebastian.poeplau@eurecom.fr
aurelien.francillon@eurecom.fr
EURECOM
Sophia Antipolis, France

SymQEMU: Compilation-based symbolic execution for binaries

TAKEAWAYS

- Intuition: embed the symbolic executor code inside the application itself!
- instruments QEMU's IR (TCG) during dynamic binary translation to embed SE operations. Compile the final instrumented IR to machine code and execute!
- Symbolic reasoning of operations is borrowed from a previous project: SymCC



σ
Symbolic store

stmt
Next Ins. to eval

Σ
SMT
solver

π
Path Constraints

Sm
States Manager

⚠ **States Management**

0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

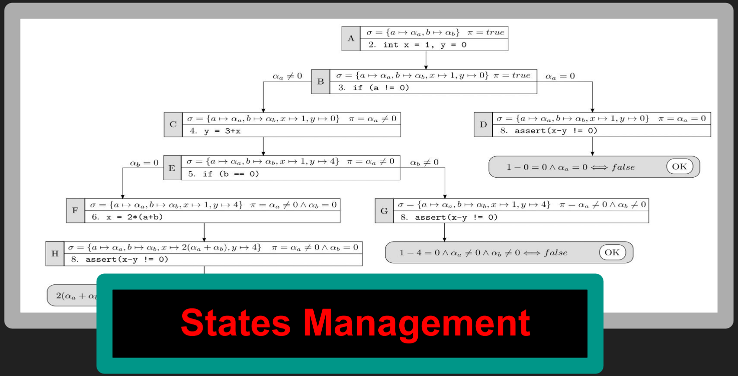
Target Binary

State Management

- How to efficiently explore the program's states in a given time budget?

- Issues:
  - Too many states to track (state/path explosion)
  - Wasting time analyzing useless path!
  - States have too complex path constraints

- Approaches:
  - Hybrid Execution (concrete+symbolic)
  - Program Summarization
  - Path Scheduling



Hybrid Execution

- Mix concrete and symbolic inputs to support symbolic analysis and increase code coverage (and therefore, possibility to find new bugs)

- This SE variant has been heavily used in modern hybrid fuzzers

- Approaches:
    - [2005] Concolic Execution (DSE)
    - [2014] Symcretic Symbolic Execution

Hybrid Execution

- Mix concrete and symbolic inputs to support symbolic analysis and increase code coverage (and therefore, possibility to find new bugs)
- This SE variant has been heavily used in modern hybrid fuzzers
- Approaches:
  - **[2005] Concolic Execution (DSE)**
    - Execute program with symbolic and concrete inputs. Collect path constraints and use concrete values to help symb-exec to get unstuck.
  - [2014] Symcretic Symbolic Execution

σ = symbolic store

π = path constraints

x=X0, y=Y0

x=1, y=2

```
void test_me(int x, int y){
    z = (y*y) % 50;
    if(z == x){
      // ERROR
    }else{
      // SOMETHING
    }
}
```

S2

σ

z = (y0y0) mod 50

Concolic Execution (DSE)

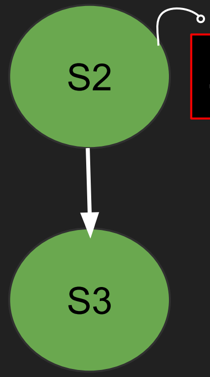
x=X0, y=Y0

x=1, y=2

```
void test_me(int x, int y){
    z = (y*y) % 50;
    if(z == x){
      // ERROR
    }else{
      // SOMETHING
    }
}
```

Concolic Execution (DSE)

σ = symbolic store **π = path constraints**



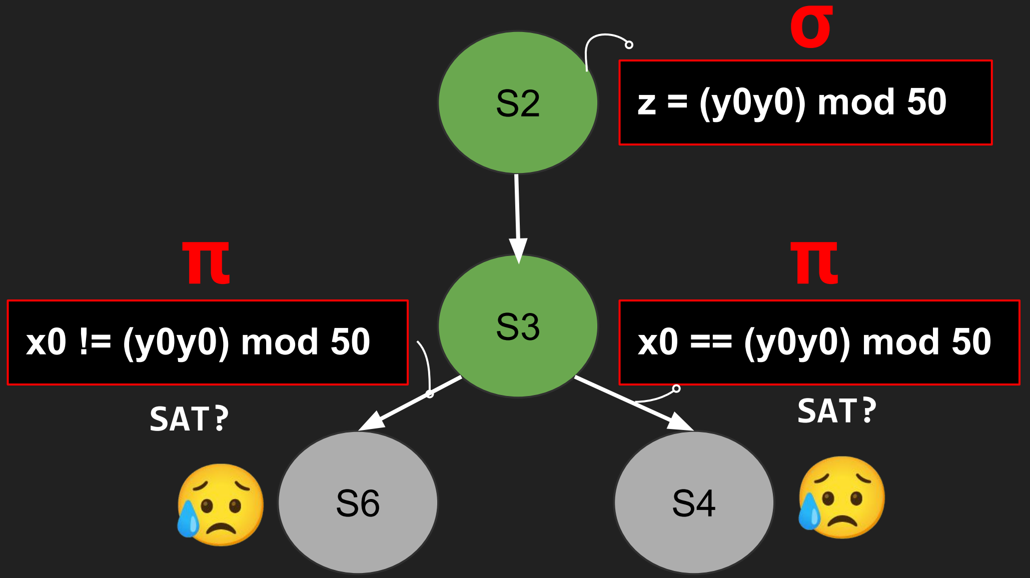
σ

z = (y0y0) mod 50

σ = symbolic store **π = path constraints**

x=X0, y=Y0
x=1, y=2

```
void test_me(int x, int y){
    z = (y*y) % 50;
    if(z == x){
      // ERROR
    }else{
      // SOMETHING
    }
}
```

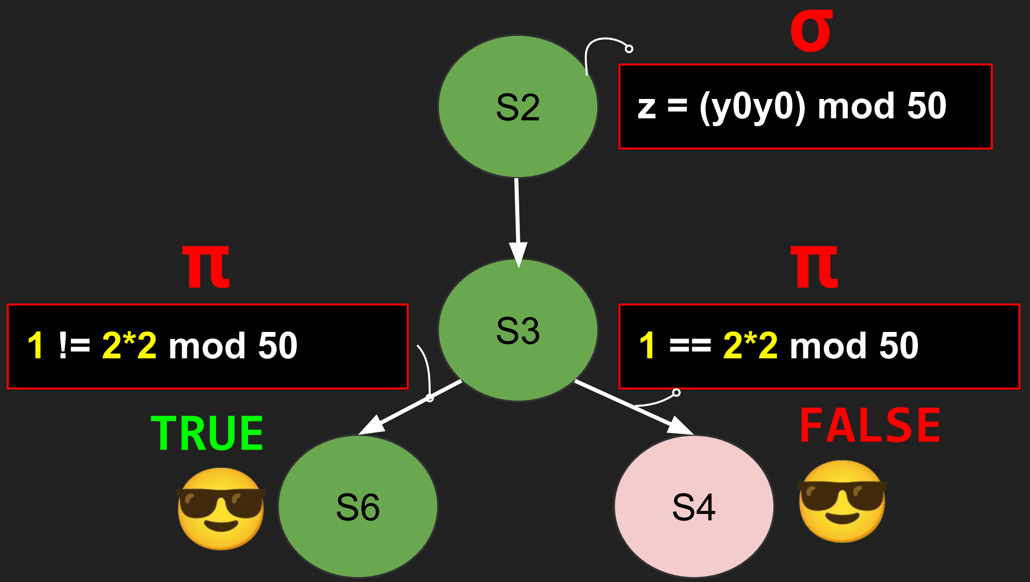


Concolic Execution (DSE)

σ = symbolic store   π = path constraints

x=X0, y=Y0
x=1, y=2

```
void test_me(int x, int y){
    z = (y*y) % 50;
    if(z == x){
      // ERROR
    }else{
      // SOMETHING
    }
}
```



Concolic Execution (DSE)

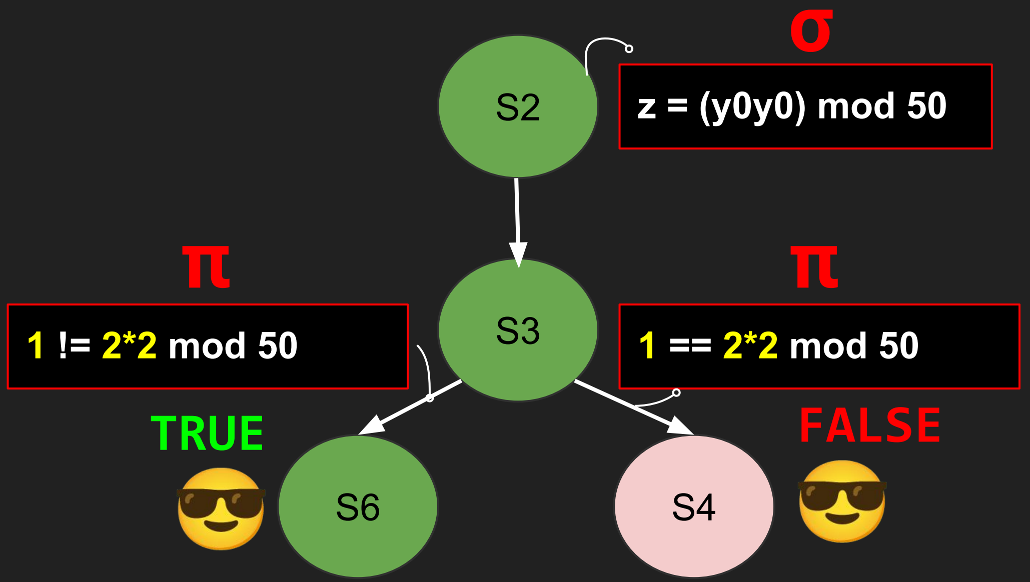
σ = symbolic store

π = path constraints

x=X0, y=Y0

x=1, y=2

```
void test_me(int x, int y){
   z = (y*y) % 50;
   if(z == x){
     // ERROR
   }else{
     // SOMETHING
   }
}
```



Negate constraints, generate new inputs, cover new code!

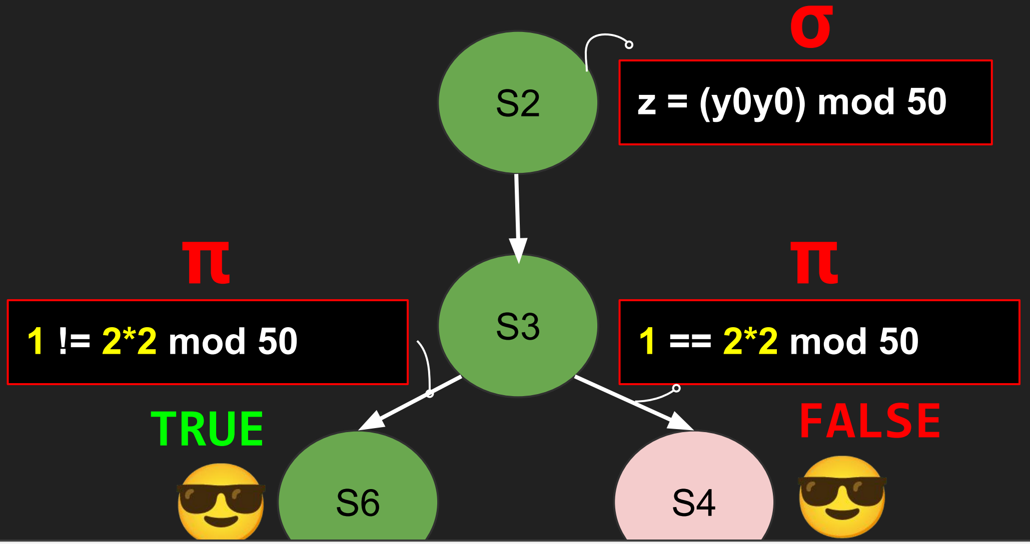
Concolic Execution (DSE)

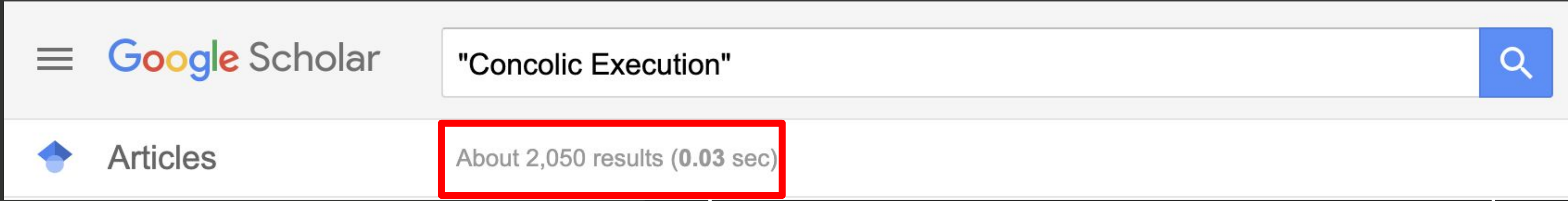
σ = symbolic store   π = path constraints

x=X0, y=Y0

x=1, y=2

```
void test_me(int x, int y){
    z = (y*y) % 50;
    if(z == x){
      // ERROR
    }else{
      // SOMETHING
    }
}
```





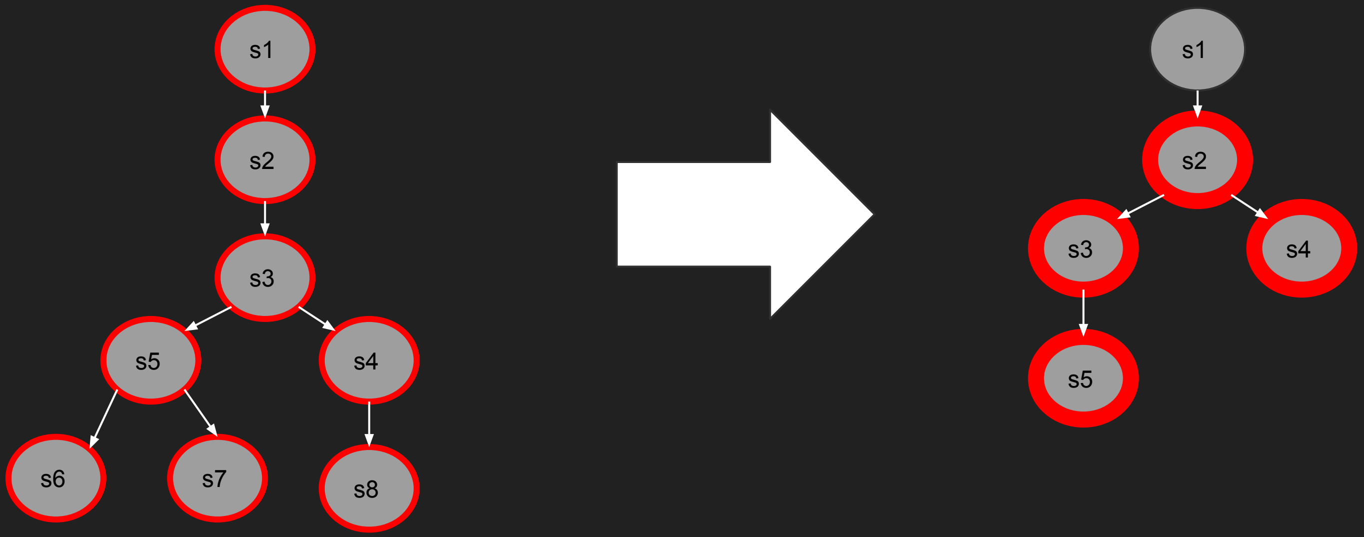
Concolic Execution (DSE)

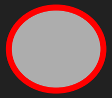
Hybrid Execution

- Mix concrete and symbolic inputs to support symbolic analysis and increase code coverage (and therefore, possibility to find new bugs)
- This SE variant has been heavily used in modern hybrid fuzzers
- Approaches:
  - [2005] Concolic Execution (DSE)
  - **[2014] Symcretic Symbolic Execution**
    - Use backward symbolic execution (BSE) and concrete execution to reason about specific target instruction (a.k.a. *line reachability problem)*

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications





Red border weight represents “state expressiveness” (i.e., its complexity in terms of constraints)

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
  - **Function summarization**
  - **Loop summarization**
  - **State merging**
  - **Third-party libraries summarization**

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
    - **Function summarization**
        - Avoid paying the cost of re-executing the same functions over and over.
    - **Loop summarization**
    - **States Merging**
    - **Third-party libraries summarization**

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
  - **Function summarization**
    - **[2007] Compositional Dynamic Test Generation**
      - Summarize functions with preconditions and postconditions
    - **[2008] Demand-driven Compositional Test Generation**
      - Summarize functions with pre- and post- targeting a specific path
  - **Loop summarization**
  - **States Merging**
  - **Third-party libraries summarization**

```c
int is_positive(int x){
    if (x>0) return 1;
    return 0;
}
```

```c
int is_positive(int x){
    if (x>0) return 1;
    return 0;
}
```

is_positive:

$$(\underbrace{x > 0}_{\text{pre-cond1}} \land \underbrace{ret = 1}_{\text{post-cond1}}) \lor (\underbrace{x \leq 0}_{\text{pre-cond2}} \land \underbrace{ret = 0}_{\text{post-cond2}})$$

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
  - **Function summarization**
  - **Loop summarization**
    - Avoid to pay the cost of re-executing the same loop every time a state enters it
  - **States Merging**
  - **Third-party libraries summarization**

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
    - **Function summarization**
    - **Loop summarization**
        - [2009] Loop-Extended Symbolic Execution on Binary Programs
        - [2011] Automatic Partial Loop Summarization in Dynamic Test Generation
        - [2016] Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis
    - **States Merging**
    - **Third-party libraries summarization**

Program Summarization

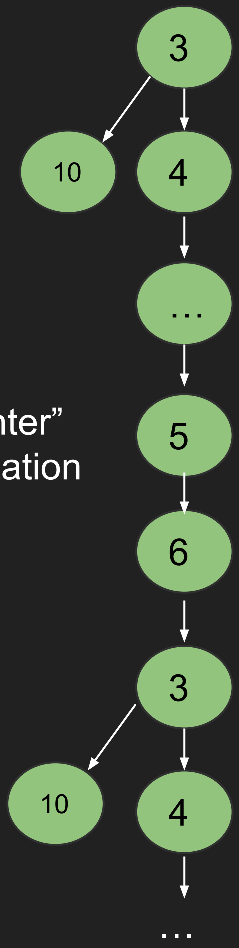
- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
    - **Function summarization**
    - **Loop summarization**
        - **[2009] Loop-Extended Symbolic Execution on Binary Programs**
            - Relate loop's dependent vars with program inputs to increase amount of information we can extract from a loop
        - [2011] Automatic Partial Loop Summarization in Dynamic Test Generation
        - [2016] Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis
    - **States Merging**
    - **Third-party libraries summarization**

symbolic input

```
int func(int X){
 counter = 0
 for(int i=0; i<X; i++){
   do_things();
   counter++
      if(counter == 1000){
        maybe_bug()
      }
   }
}
```

No “counter” symbolization



symbolic input

```
1  | int func(int X){
2  |  counter = 0
3  |  for(int i=0; i<X; i++){
4  |    do_things();
5  |    counter++
6  |       if(counter == 1000){
7  |          maybe_bug()
8  |       }
9  |    }
10 | }
```

“counter” symbolization

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
  - **Function summarization**
  - **Loop summarization**
    - [2009] Loop-Extended Symbolic Execution on Binary Programs
    - **[2011] Automatic Partial Loop Summarization in Dynamic Test Generation**
      - Capture loop's side-effects and summarize with pre-conditions and post-conditions formulas over input (single path loop)
    - [2016] Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis
  - **States Merging**
  - **Third-party libraries summarization**

```
1|   int n:=*;
2|   int x:=*;
3|   int z:=*;
4|   while (x<n){
5|       x++;
6|   }
```

Single-path Loop

Cycle 1: 4-5
Cycle 2: 4-5
…

$$\underbrace{x_0 < n}_{\text{pre-cond}} \wedge \underbrace{(x = n)}_{\text{post-cond}}$$

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
  - **Function summarization**
  - **Loop summarization**
    - [2009] Loop-Extended Symbolic Execution on Binary Programs
    - [2011] Automatic Partial Loop Summarization in Dynamic Test Generation
    - **[2016] Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis**
      - Summarize multi-paths loops with a disjunction of constraints
  - **States Merging**
  - **Third-party libraries summarization**

```
1|  int n:=*;
2|  int x:=*;
3|  int z:=*;
4|  while (x<n){
5|    if(z>x) x++;
6|    else z++;
7|  }
```

Multi-path Loop

Cycle 1: 4-5
Cycle 2: 4-5
Cycle X: 4-6
…

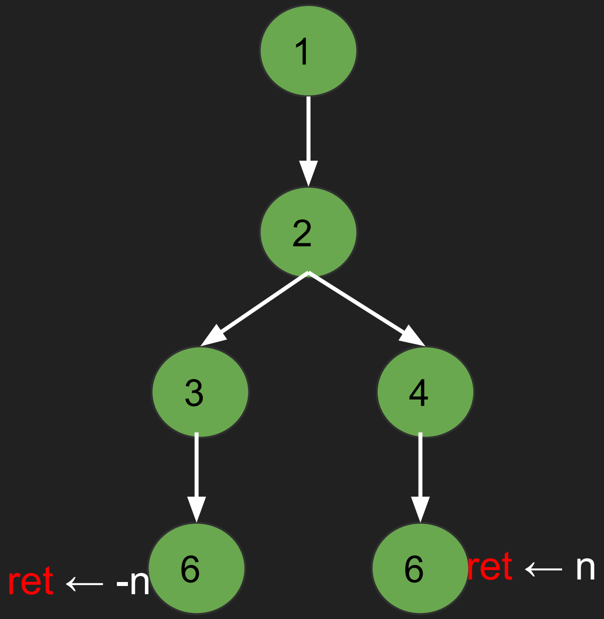
$$(\underbrace{x_0 \ge n}_{\text{pre-cond1}} \wedge \underbrace{x = x_0 \wedge z = z_0}_{\text{post-cond1}}) \vee (\underbrace{x_0 < n \le z_0}_{\text{pre-cond2}} \wedge \underbrace{x = n \wedge z = z_0}_{\text{post-cond2}}) \vee (\underbrace{x_0 < n \wedge z < n}_{\text{pre-cond3}} \wedge \underbrace{x = z_0 = n}_{\text{post-cond3}})$$

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

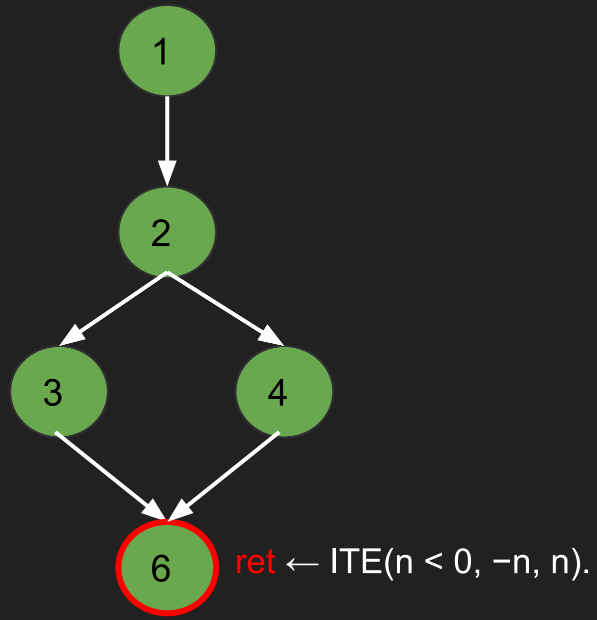
- Approaches:
    - **Function summarization**
    - **Loop summarization**
    - **States Merging**
        - Model state progression using path constraints rather than generating new states
    - Third-party libraries summarization

```
void abs(int n){
  int ret = 0;
  if (n < 0)
    ret = -n;
  else
    ret = n;
  return ret;
```



No merging

```
void abs(int n){
  int ret = 0;
  if (n < 0)
    ret = -n;
  else
    ret = n;
  return ret;
```



States merging

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
    - **Function summarization**
    - **Loop summarization**
    - **States Merging**
        - [2014] Enhancing Symbolic Execution with Veritesting
            - Leverages SSE to find opportunities of state merging during DSE
        - [2012] Efficient State Merging in Symbolic Execution
        - [2018] Boost Symbolic Execution Using Dynamic State Merging and Forking
    - Third-party libraries summarization

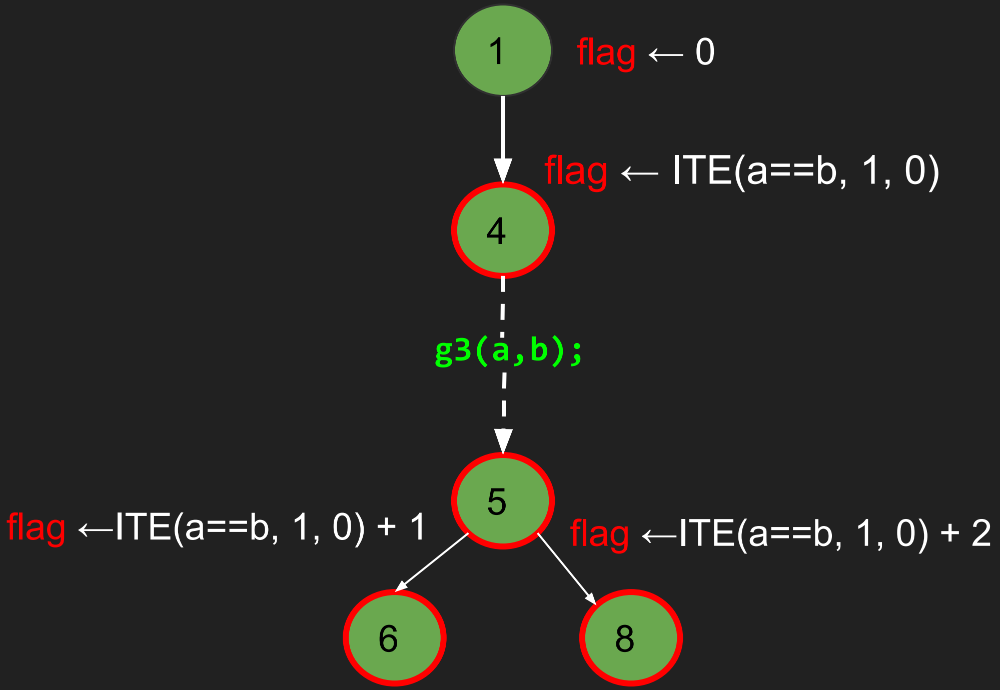
Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
    - **Function summarization**
    - **Loop summarization**
    - **States Merging**
        - [2014] Enhancing Symbolic Execution with Veritesting
        - [2012] Efficient State Merging in Symbolic Execution
            - Optimize states merging opportunities to avoid performance loss.
        - [2018] Boost Symbolic Execution Using Dynamic State Merging and Forking
            - Optimization of “Efficient State Merging in Symbolic Execution”
    - Third-party libraries summarization

```
  | void f3(int a,int b){
1 |  int flag = 0;
2 |  If (a == b){
3 |    flag = 1;
  |  }
4 |  g3(a,b);
5 |  If (flag)
6 |      g1(flag+1);
7 |  else
8 |      g2(flag+2);
  |  }
```

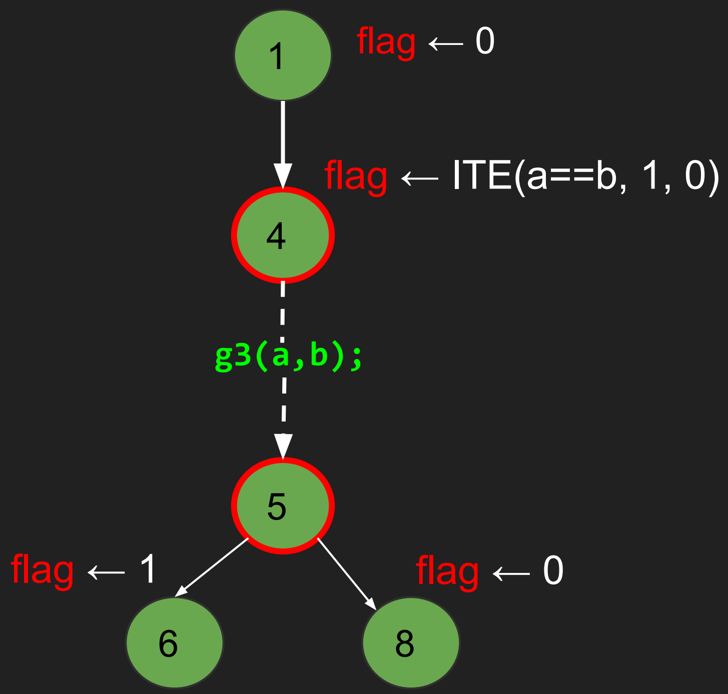
Assume parameters a,b are symbolic



Naive dynamic merging
(Efficient State Merging in Symbolic Execution)

```
  | void f3(int a,int b){
1 |  int flag = 0;
2 |  If (a == b){
3 |    flag = 1;
  |  }
4 |  g3(a,b);
5 |  If (flag)
6 |      g1(flag+1);
7 |  else
8 |      g2(flag+2);
  |  }
```

Assume parameters a,b are symbolic



Dynamic merging + active fork
(Boost Symbolic Execution Using Dynamic State Merging and Forking)

Program Summarization

- Reduce the amount of generated states by using constraints or simplifications

- Approaches:
  - **Function summarization**
  - **Loop summarization**
  - **States Merging**
  - **Third-party libraries summarization**
    - Use “models” to summarize side-effects of non-tracked functions on the symbolic states (We’ll see this later when discussing the execution Environment)

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
    - **Path Pruning**
        - Stop the symbolic engine to explore specific paths when certain conditions arise
    - **Path Prioritization**
        - Prioritize exploration of specific paths according to some conditions (e.g., bug detected, calls to specific functions), or, following a specific order (BFS, DFS, etc…)

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
  - **Path Pruning**
    - [2012] Pre-conditioned Symbolic Execution
    - [2015] Underconstrained Symbolic Execution
    - [2015] Post-conditioned Symbolic Execution
    - [2018] Dynamic Path Pruning in Symbolic Execution
  - **Path Prioritization**
    - [2008] Random Path Selection & Coverage Optimized Search
    - [2011] Directed Symbolic Execution
    - [2018] Chopped Symbolic Execution
    - [2021] SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning
    - [2021] Learning to Explore Paths for Symbolic Execution

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion
- Approaches:
  - **Path Pruning**
    - **[2012] Pre-conditioned Symbolic Execution**
      - Pre-constrain program's input to promote exploration of exploitable paths, prune the rest.
    - [2015] Underconstrained Symbolic Execution
    - [2015] Post-conditioned Symbolic Execution
    - [2018] Dynamic Path Pruning in Symbolic Execution
  - **Path Prioritization**
    - [+]

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
  - **Path Pruning**
    - [2012] Pre-conditioned Symbolic Execution
    - **[2015] Underconstrained Symbolic Execution**
      - Start symbolic execution from an arbitrary function rather than entry point. Pay the cost of many symbolic vars in memory.
    - [2015] Post-conditioned Symbolic Execution
    - [2018] Dynamic Path Pruning in Symbolic Execution
  - **Path Prioritization**
    - [+]

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

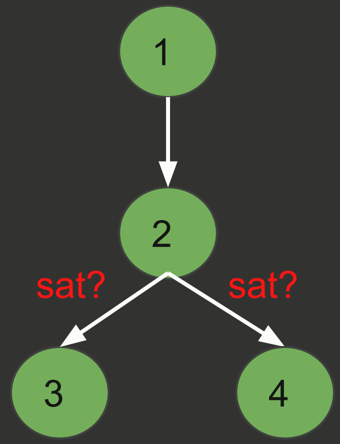
- Approaches:
    - **Path Pruning**
        - [2012] Pre-conditioned Symbolic Execution
        - [2015] Underconstrained Symbolic Execution
        - **[2015] Post-conditioned Symbolic Execution**
            - Avoid the analysis of common path suffixes to reduce the number of generated states
        - [2018] Dynamic Path Pruning in Symbolic Execution
    - **Path Prioritization**
        - [+]

Path Scheduling

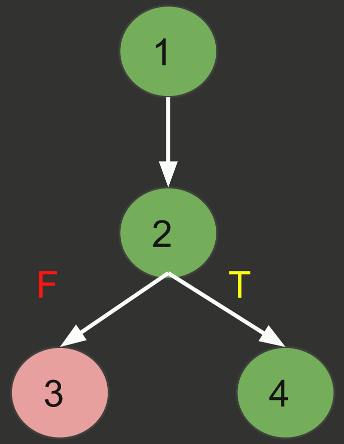
- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
    - **Path Pruning**
        - [2012] Pre-conditioned Symbolic Execution
        - [2015] Underconstrained Symbolic Execution
        - [2015] Post-conditioned Symbolic Execution
        - **[2018] Dynamic Path Pruning in Symbolic Execution**
            - Optimize the numbers of checks for SAT/UNSAT paths to speed up symbolic execution. CheckAll vs CheckNothing vs "CheckDynamically"
    - **Path Prioritization**
        - [+]

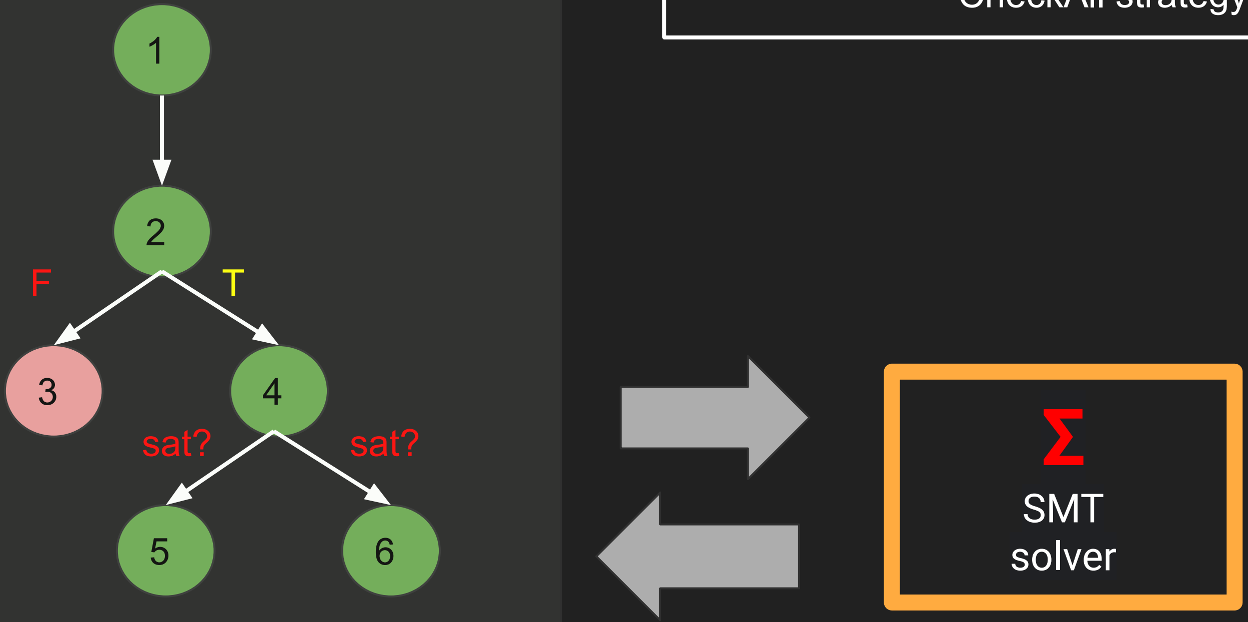
CheckAll strategy

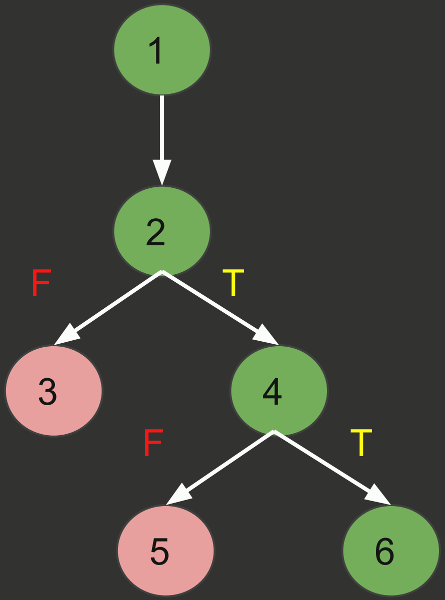


CheckAll strategy



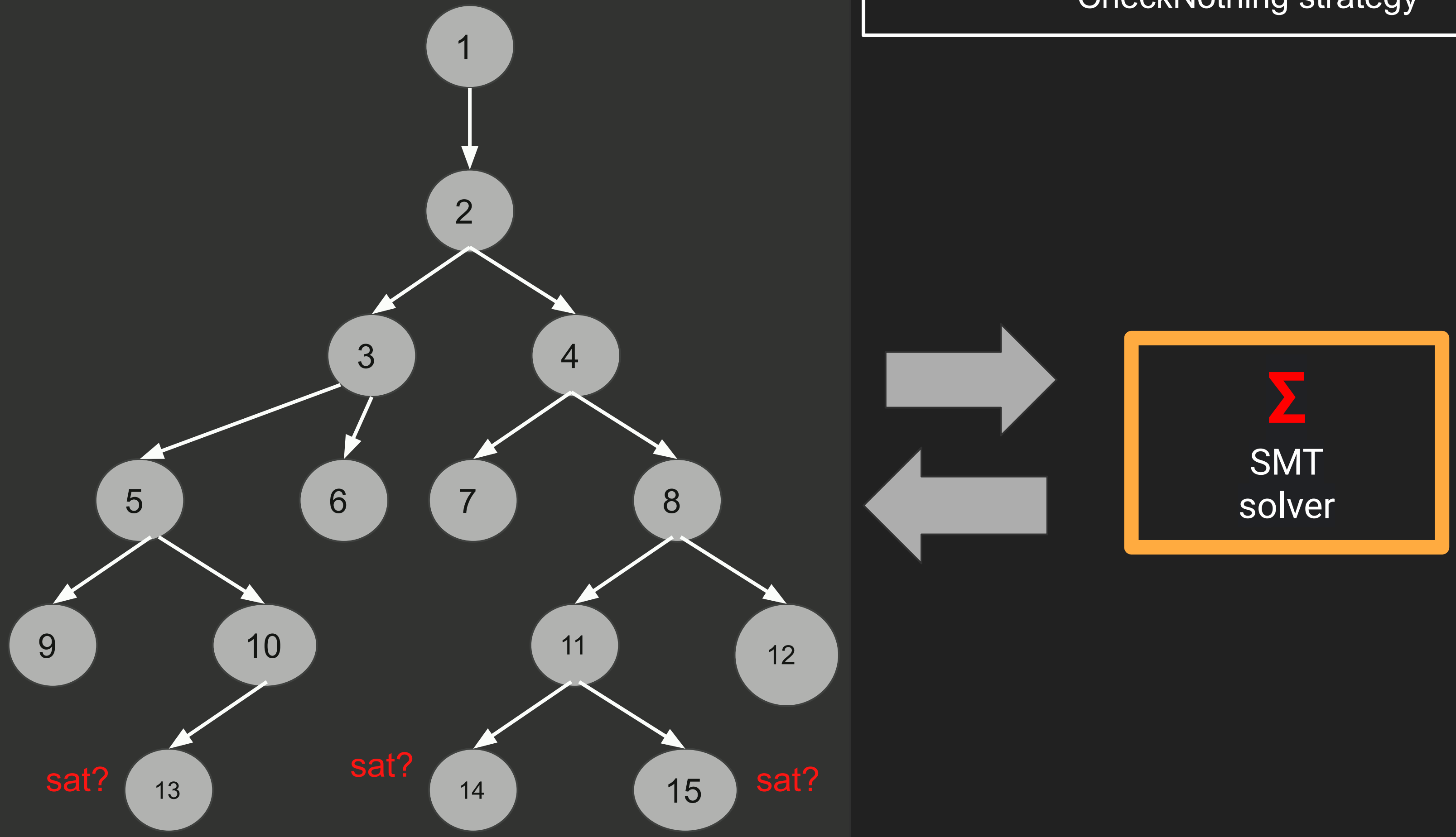
CheckAll strategy



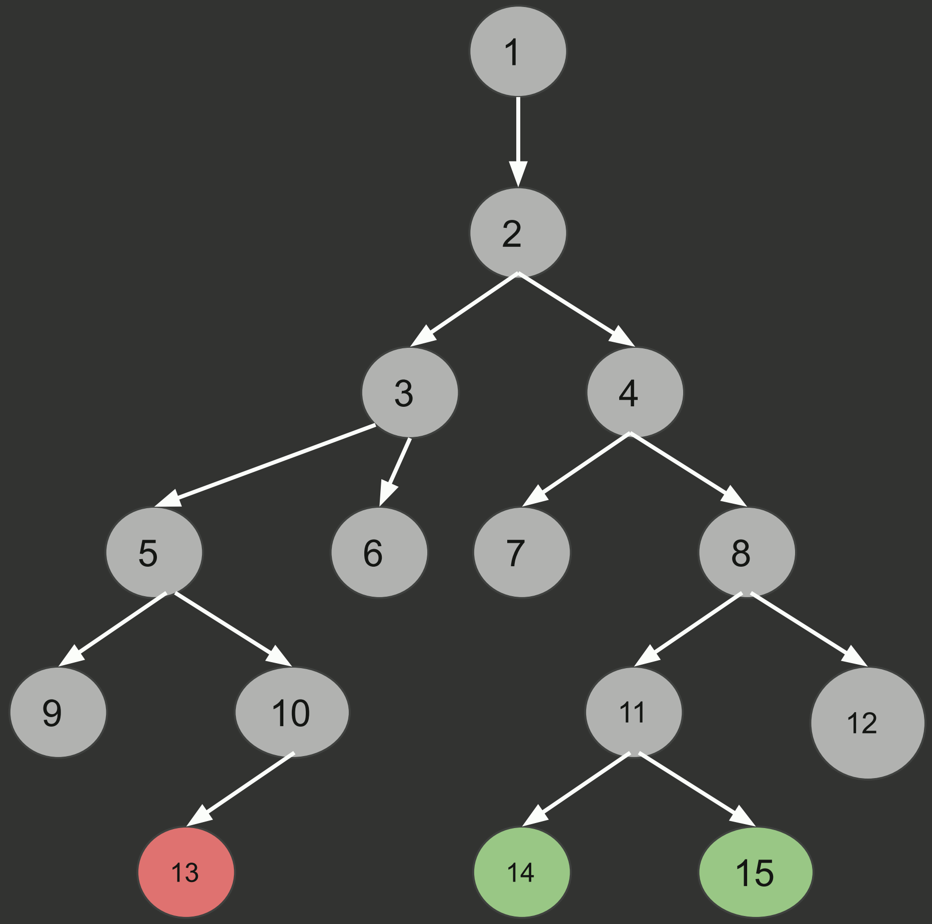


CheckAll strategy

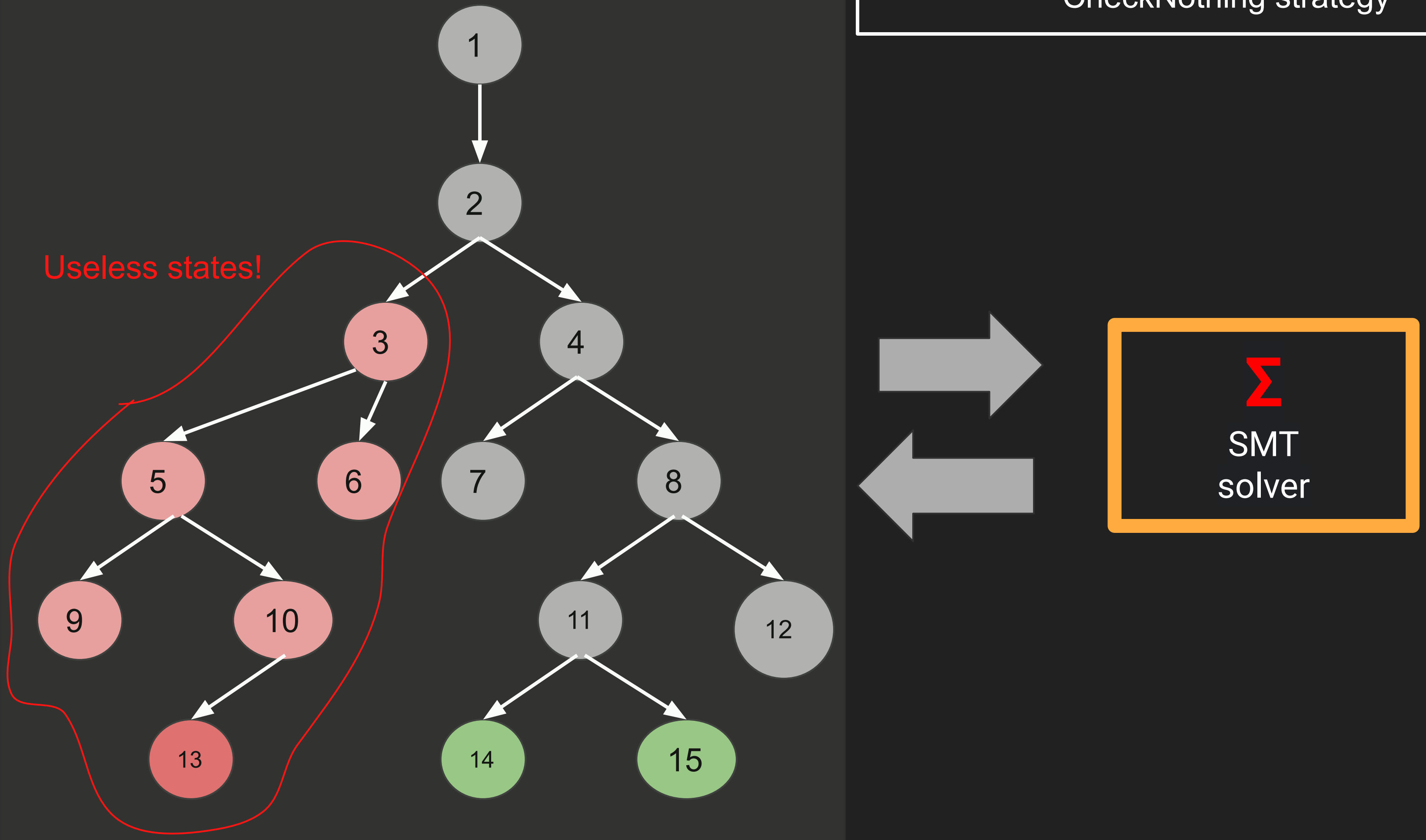
CheckNothing strategy



CheckNothing strategy

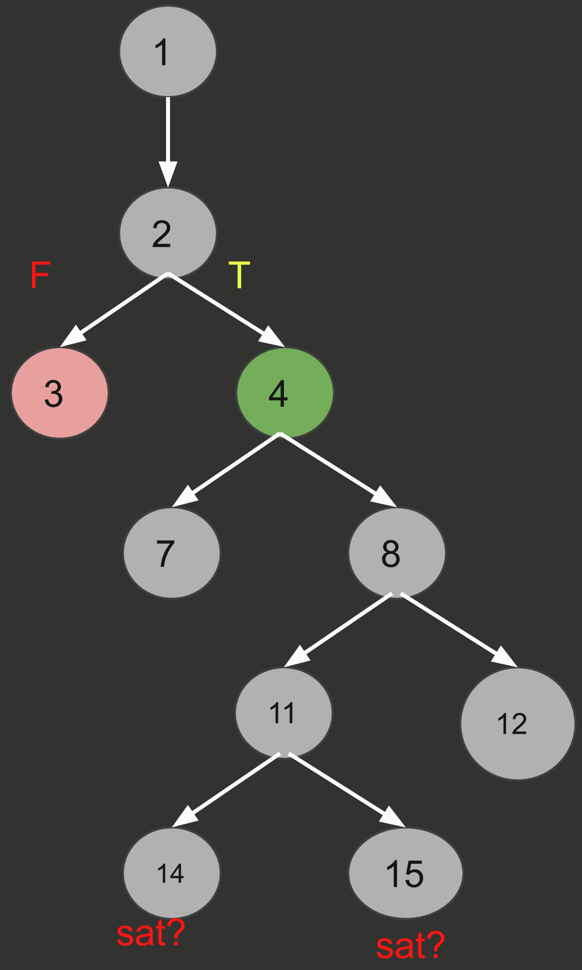


CheckNothing strategy



CheckDynamic strategy

(Dynamic Path Pruning in Symbolic Execution)



Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
    - **Path Pruning**
        - [+]
    - **Path Prioritization**
        - **[2008] Random Path Selection & Coverage Optimized Search**
            - Pick next state to explore by walking the tree of already explored states from the root and randomly take branches until a leaf
            - Pick next state that covers unseen instructions.
        - [2011] Directed Symbolic Execution
        - [2018] Chopped Symbolic Execution
        - [2021] SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning
        - [2021] Learning to Explore Paths for Symbolic Execution

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
  - **Path Pruning**
    - [+]
  - **Path Prioritization**
    - [2008] Random Path Selection & Coverage Optimized Search
    - **[2011] Directed Symbolic Execution**
      - Symbolic execution used to study how to reach a specific target line in a program
    - [2018] Chopped Symbolic Execution
    - [2021] SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning
    - [2021] Learning to Explore Paths for Symbolic Execution

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

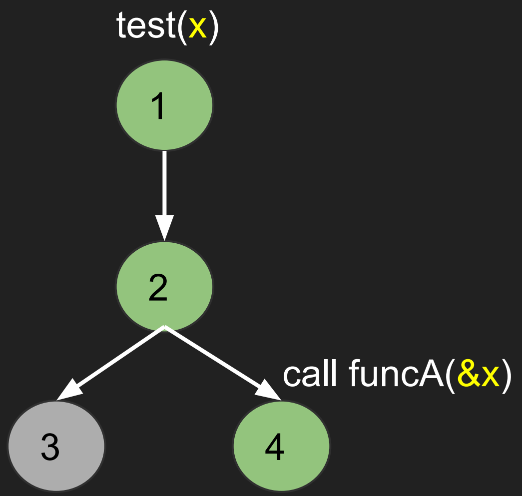
- Approaches:
  - **Path Pruning**
    - [+]
  - **Path Prioritization**
    - [2008] Random Path Selection & Coverage Optimized Search
    - [2011] Directed Symbolic Execution
    - **[2018] Chopped Symbolic Execution**
      - Pre-define part of the program that should be skipped during the symbolic-execution, come back later if needed.
    - [2021] SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning
    - [2021] Learning to Explore Paths for Symbolic Execution

Target:

- test

Skip:

- funcA
- funcB



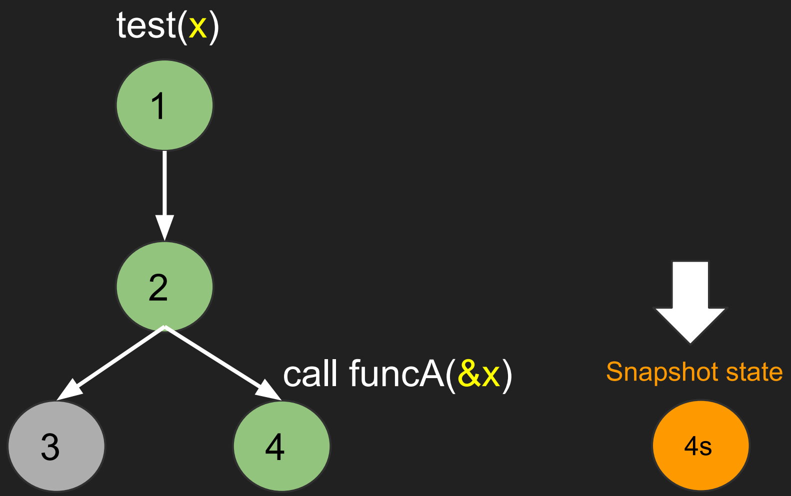
Chopped symbolic execution

Target:
- test

Skip:
- funcA
- funcB

WritesTo:
- funcA: x

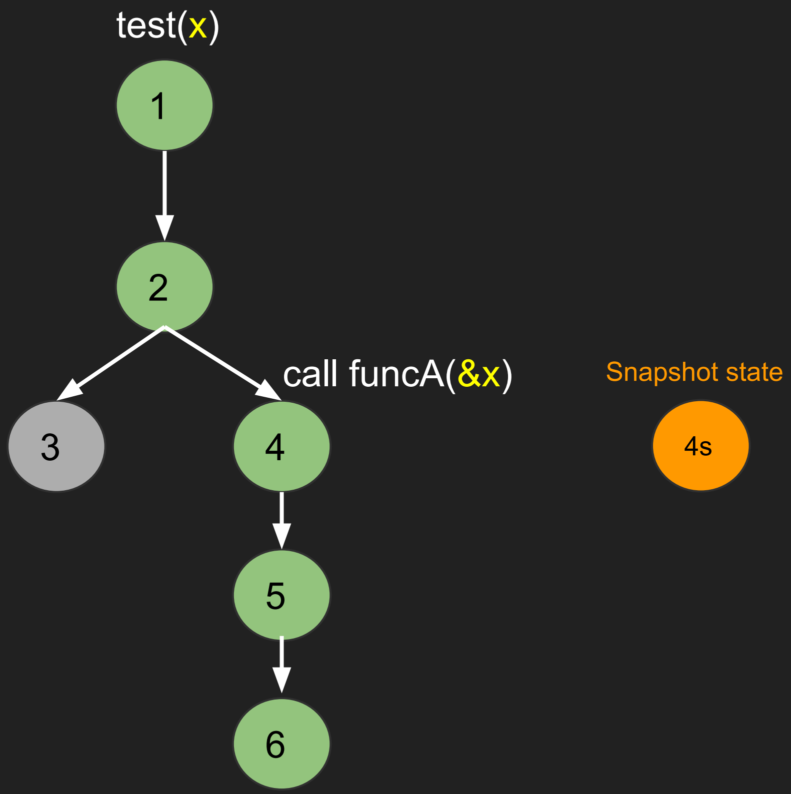
Computed with pointer analysis



Target:
- test

Skip:
- funcA
- funcB

WritesTo:
- funcA: x

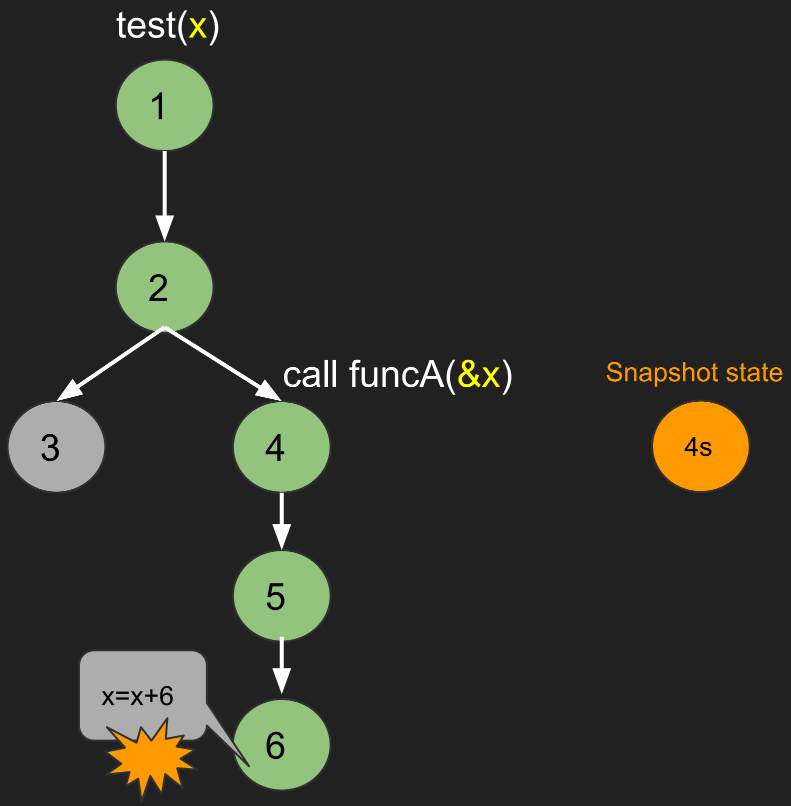


Chopped symbolic execution

Target:
- test

Skip:
- funcA
- funcB

WritesTo:
- funcA: x



Chopped symbolic execution

Target:

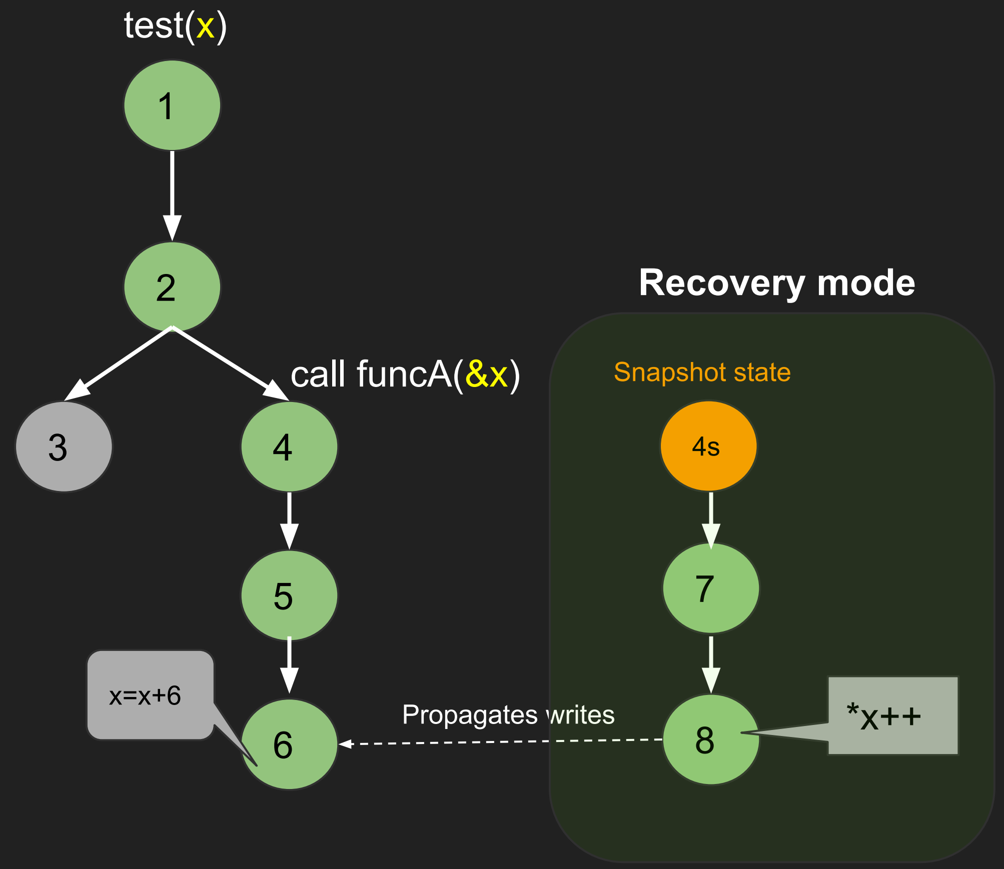
- test

Skip:

- funcA
- funcB

WritesTo:

- funcA: x



Chopped symbolic execution

Path Scheduling

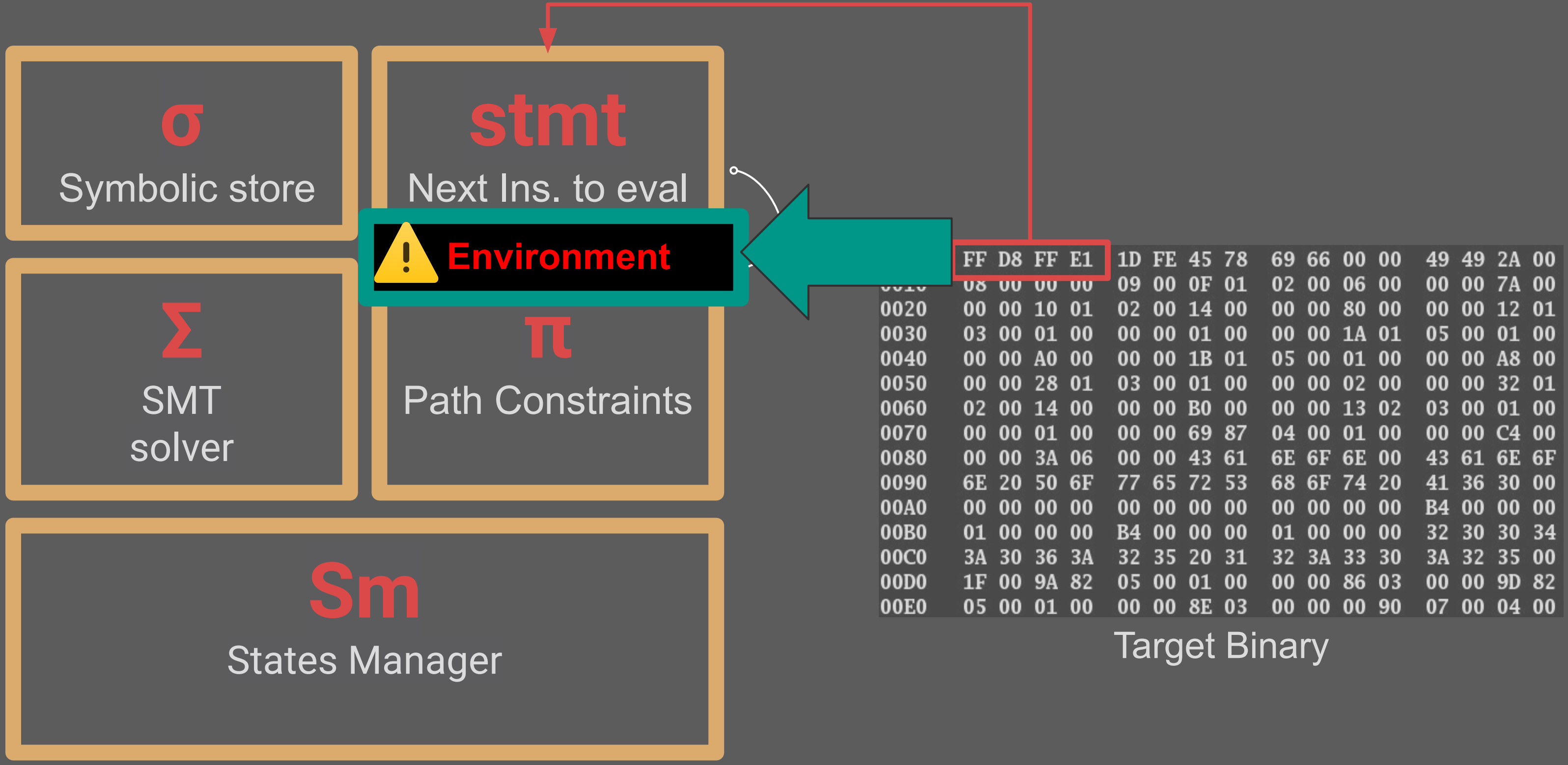
- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
  - **Path Pruning**
    - [+]
  - **Path Prioritization**
    - [2008] Random Path Selection & Coverage Optimized Search
    - [2011] Directed Symbolic Execution
    - [2018] Chopped Symbolic Execution
    - **[2021] SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning**
      - Use ML to decide which paths' are more promising to reach a vulnerability.
    - [2021] Learning to Explore Paths for Symbolic Execution

Path Scheduling

- Manage paths exploration to reach more interesting program's state and avoid state explosion

- Approaches:
  - **Path Pruning**
    - [+]
  - **Path Prioritization**
    - [2008] Random Path Selection & Coverage Optimized Search
    - [2011] Directed Symbolic Execution
    - [2018] Chopped Symbolic Execution
    - [2021] SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning
    - **[2021] Learning to Explore Paths for Symbolic Execution**
      - Attempting to generalize the "state searching" problem. Offline training to automatically derive searching strategies with a set of states' features that prioritize certain goals.



σ
Symbolic store
stmt
Next Ins. to eval
Environment
Σ
SMT
solver
π
Path Constraints
Sm
States Manager
FF D8 FF E1 1D FE 45 78 69 66 00 00 49 49 2A 00
08 00 00 00 09 00 0F 01 02 00 06 00 00 00 7A 00
0020 00 00 10 01 02 00 14 00 00 00 80 00 00 00 12 01
0030 03 00 01 00 00 00 01 00 00 00 1A 01 05 00 01 00
0040 00 00 A0 00 00 00 1B 01 05 00 01 00 00 00 A8 00
0050 00 00 28 01 03 00 01 00 00 00 02 00 00 00 32 01
0060 02 00 14 00 00 00 B0 00 00 00 13 02 03 00 01 00
0070 00 00 01 00 00 00 69 87 04 00 01 00 00 00 C4 00
0080 00 00 3A 06 00 00 43 61 6E 6F 6E 00 43 61 6E 6F
0090 6E 20 50 6F 77 65 72 53 68 6F 74 20 41 36 30 00
00A0 00 00 00 00 00 00 00 00 00 00 00 00 B4 00 00 00
00B0 01 00 00 00 B4 00 00 00 01 00 00 00 32 30 30 34
00C0 3A 30 36 3A 32 35 20 31 32 3A 33 30 3A 32 35 00
00D0 1F 00 9A 82 05 00 01 00 00 00 86 03 00 00 9D 82
00E0 05 00 01 00 00 00 8E 03 00 00 00 90 07 00 04 00
Target Binary

Environment

- How to handle code that interact with external environment or third party libraries?

- Issues:
  - Unmodeled interactions = add symbolic variable? stop execution? Execute everything symbolic?

- Approaches:
  - Abstract Models [7, 15, 25]
  - Concrete Delegation [10, 34, 40]

Environment

- How to handle code that interact with external environment or third party libraries?

- Issues:
    - Unmodeled interactions = add symbolic variable? stop execution? Execute everything symbolic?

- Approaches:
    - **Abstract Models**
        - Summarize a call to external procedure with a specific function
            - Function level VS Syscall level
    - Concrete Delegation

Environment

- How to handle code that interact with external environment or third party libraries?

- Issues:
  - Unmodeled interactions = add symbolic variable? stop execution? Execute everything symbolic?

- Approaches:
  - Abstract Models
  - **Concrete Delegation**
    - Execution of external functions is delegated to the real system outside of the symbolic executor

Environment

- How to handle code that interact with external environment or third party libraries?

- Issues:
  - Unmodeled interactions = add symbolic variable? stop execution? Execute everything symbolic?

- Approaches:
  - Abstract Models
  - **Concrete Delegation**
    - [2011] Selective Symbolic Execution
    - [2012] Unleashing Mayhem on Binary Code
    - [2020] Interleaved Symbolic Execution

Environment

- How to handle code that interact with external environment or third party libraries?

- Issues:
  - Unmodeled interactions = add symbolic variable? stop execution? Execute everything symbolic?

- Approaches:
  - Abstract Models
  - **Concrete Delegation**
    - **[2011] Selective Symbolic Execution**
      - Run entire software stack in emulator. Symbolically execute only a pre-selected part of the software stack, concretely execute the rest.
    - [2012] Unleashing Mayhem on Binary Code
    - [2020] Interleaved Symbolic Execution

Environment

- How to handle code that interact with external environment or third party libraries?

- Issues:
    - Unmodeled interactions = add symbolic variable? stop execution? Execute everything symbolic?

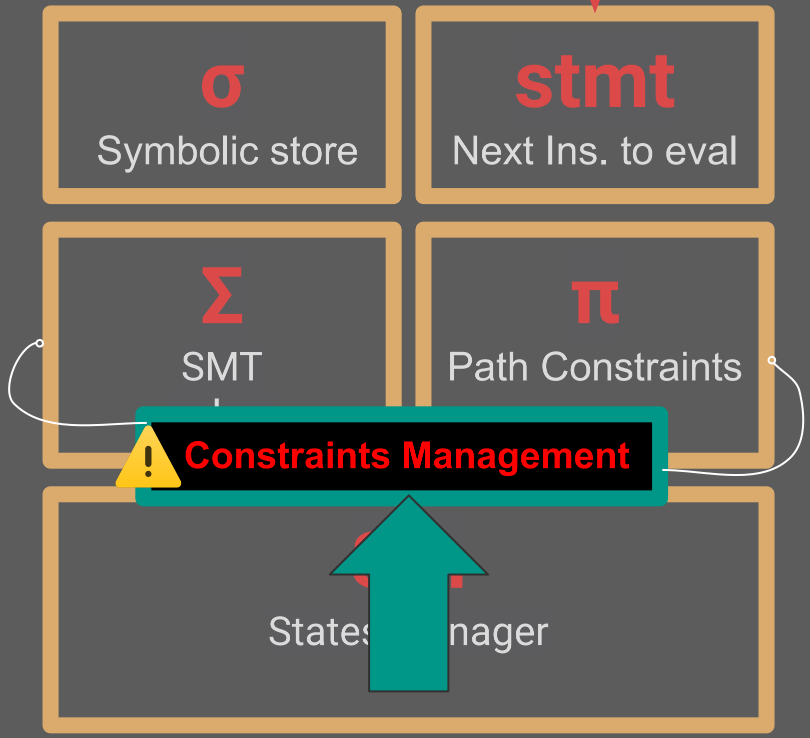
- Approaches:
    - Abstract Models
    - **Concrete Delegation**
        - [2011] Selective Symbolic Execution
        - **[2012] Unleashing Mayhem on Binary Code**
            - Maintain a lightweight virtual machine for every state.
        - [2020] Interleaved Symbolic Execution

Environment

- How to handle code that interact with external environment or third party libraries?

- Issues:
  - Unmodeled interactions = add symbolic variable? stop execution? Execute everything symbolic?

- Approaches:
  - Abstract Models
  - **Concrete Delegation**
    - [2011] Selective Symbolic Execution
    - [2012] Unleashing Mayhem on Binary Code
    - **[2020] Interleaved Symbolic Execution**
      - Manually interleave concrete and symbolic execution to reach “deeper” code in the program.



0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

Target Binary

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Issues:
    - Querying the SMT solver too often is a bottleneck
    - Some constraints CANNOT be solved

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
  - Constraints Reduction
  - Constraints Caching
  - Constraints Prediction
  - New Constraints Solving Techniques

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
    - **Constraints Reduction**
        - Simplify the constraints with equivalents ones to speed up solving time
    - Constraints Caching
    - Constraints Prediction
    - New Constraints Solving Techniques

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
    - **Constraints Reduction**
        - [2008] Expression Rewriting[7]
        - [2008] Constraint Set Simplification[7]
        - [2008] Implied Value Concretization[7]
        - [2008] Constraint Independence[7]
    - Constraints Caching
    - Constraints Prediction
    - New Constraints Solving Techniques

Expression Rewriting

$x+0 \rightarrow x$ | $x * 2^n = x \ll n$ | $2*x-x=x$

Constraint Set Simplification

$x > 10 \wedge x = 5 \rightarrow \text{True}$

Implied Value Concretization

$x + 1 = 10 \rightarrow x = 9$

Constraint Independence

$\{i < j, j < 20, k > 0\}, i=20? \rightarrow \{i < j, j < 20\}$

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
  - Constraints Reduction
  - **Constraints Caching**
    - **[2008] Counter-Example Cache**[7]
      - Cache constraints solutions and consider superset/subset when solving
    - [2012] Green: Reducing, reusing and recycling constraints in program analysis
  - Constraints Prediction
  - New Constraints Solving Techniques

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
  - Constraints Reduction
  - **Constraints Caching**
    - [2008] Counter-Example Cache
    - **[2012] Green: Reducing, reusing and recycling constraints in program analysis**
      - Universal constraints caching technique. Solutions re-usable across target programs, analysis, and tools.
  - Constraints Prediction
  - New Constraints Solving Techniques

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
    - Constraints Reduction
    - Constraints Caching
    - **Constraints Prediction**
        - **[2020] Constraint Solving with Deep Learning for Symbolic Execution**
            - Canonize and vectorize a set of constraints’ and their solutions (SAT vs UNSAT) to train a DNN. Use DNN oracle run-time to check for satisfiability.
        - [2021] Boosting symbolic execution via constraint solving time prediction
    - New Constraints Solving Techniques

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

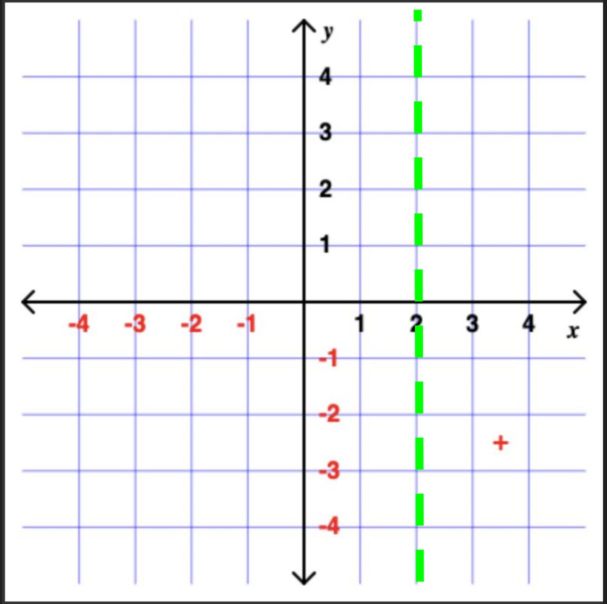
- Approaches:
  - Constraints Reduction
  - Constraints Caching
  - **Constraints Prediction**
    - [2020] Constraint Solving with Deep Learning for Symbolic Execution
    - **[2021] Boosting symbolic execution via constraint solving time prediction**
      - Use ML to predict how long it is going to take to solve a specific constraints for a target solver. Stir the execution somewhere else to avoid blocking the analysis.
  - New Constraints Solving Techniques

Constraints Management

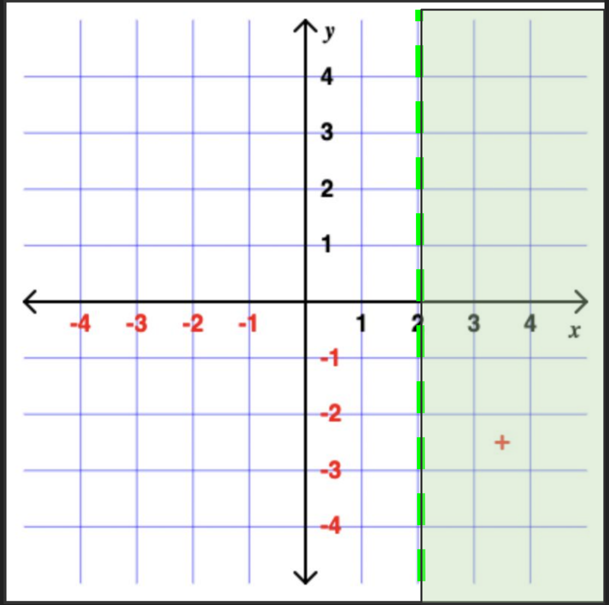
- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
    - Constraints Reduction
    - Constraints Caching
    - Constraints Prediction
    - **New Constraints Solving Techniques**
        - **[2014] Solving Complex Path Conditions through Heuristic Search on Induced Polytopes**
            - Solution for solving linear mixed with non-linear constraints. Search solutions within a polytope defined by constraints.
        - [2019] Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing
        - [2019] Enhancing Symbolic Execution by Machine Learning Based Solver Selection

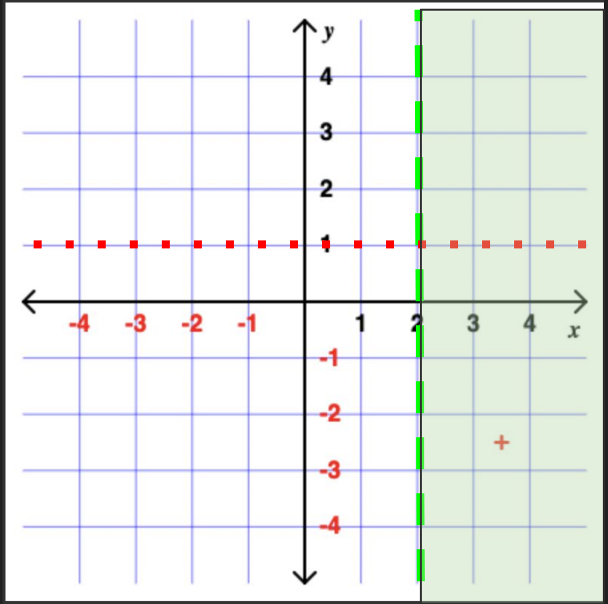
$$x = x * y \land x > 2$$



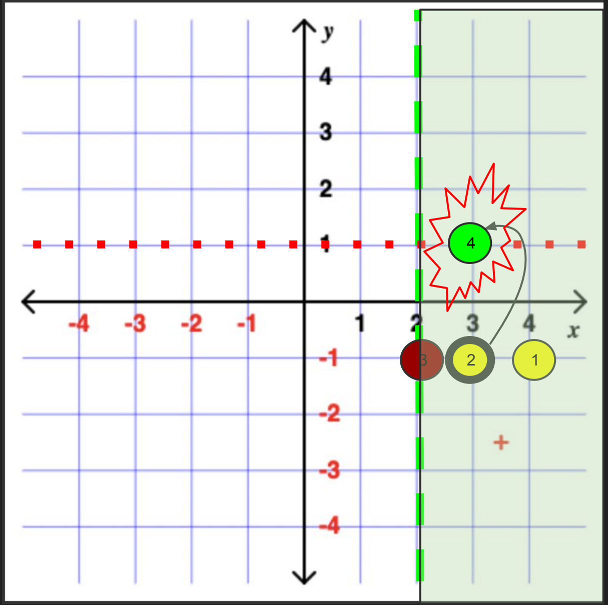
$$x = x * y \land x > 2$$



$$x = x * y \wedge x > 2$$



$x = x * y \land x > 2$



Tabu search

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
    - Constraints Reduction
    - Constraints Caching
    - Constraints Prediction
    - **New Constraints Solving Techniques**
        - [2014] Solving Complex Path Conditions through Heuristic Search on Induced Polytopes
        - **[2019] Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing**
            - Replace classic SAT solving reasoning techniques with fuzzing techniques
        - [2019] Enhancing Symbolic Execution by Machine Learning Based Solver Selection

```
(declare-fun a () Float64)
(declare-fun b () Float64)
(define-fun div_rne () Float64 (fp.div RNE a b))
(define-fun div_rtp () Float64 (fp.div RTP a b))
(assert (not (fp.isNaN a)))
(assert (not (fp.isNaN b)))
(assert (not (fp.isNaN div_rne)))
(assert (not (fp.isNaN div_rtp)))
(assert (not (fp.eq div_rne div_rtp)))
(check-sat)
```

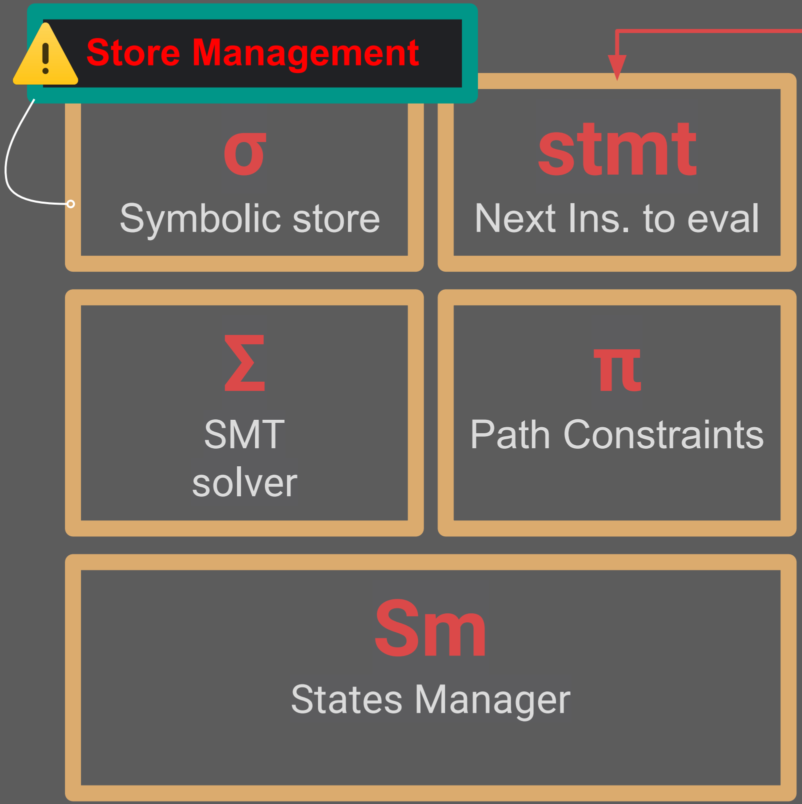
```
int FuzzOneInput(const uint8_t* data, size_t size) {
  double a = makeFloatFrom(data, size, 0, 63);
  double b = makeFloatFrom(data, size, 64, 127);
  if (!isnan(a)) {} else return 0;
  if (!isnan(b)) {} else return 0;
  double a_b_rne = div_rne(a, b);
  double a_b_rtp = div_rtp(a, b);
  if (!isnan(a_b_rne)) {} else return 0;
  if (!isnan(a_b_rtp)) {} else return 0;
  if (a_b_rne != a_b_rtp) {} else return 0;
  return 1; // TARGET REACHED
}
```


Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing

Constraints Management

- How to simplify/reduce queries to the SMT solver? How to efficiently solve constraints?

- Approaches:
    - Constraints Reduction
    - Constraints Caching
    - Constraints Prediction
    - **New Constraints Solving Techniques**
        - [2014] Solving Complex Path Conditions through Heuristic Search on Induced Polytopes
        - [2019] Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing
        - **[2019] Enhancing Symbolic Execution by Machine Learning Based Solver Selection**
            - Use ML to predict what is the best solver to handle a specific set of constraints and apply the decision run-time.



0000	FF	D8	FF	E1	1D	FE	45	78	69	66	00	00	49	49	2A	00
0010	08	00	00	00	09	00	0F	01	02	00	06	00	00	00	7A	00
0020	00	00	10	01	02	00	14	00	00	00	80	00	00	00	12	01
0030	03	00	01	00	00	00	01	00	00	00	1A	01	05	00	01	00
0040	00	00	A0	00	00	00	1B	01	05	00	01	00	00	00	A8	00
0050	00	00	28	01	03	00	01	00	00	00	02	00	00	00	32	01
0060	02	00	14	00	00	00	B0	00	00	00	13	02	03	00	01	00
0070	00	00	01	00	00	00	69	87	04	00	01	00	00	00	C4	00
0080	00	00	3A	06	00	00	43	61	6E	6F	6E	00	43	61	6E	6F
0090	6E	20	50	6F	77	65	72	53	68	6F	74	20	41	36	30	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	B4	00	00	00
00B0	01	00	00	00	B4	00	00	00	01	00	00	00	32	30	30	34
00C0	3A	30	36	3A	32	35	20	31	32	3A	33	30	3A	32	35	00
00D0	1F	00	9A	82	05	00	01	00	00	00	86	03	00	00	9D	82
00E0	05	00	01	00	00	00	8E	03	00	00	00	90	07	00	04	00

Target Binary

Store Management

- How to organize memory and how to support symbolic memory operations? (read/writes)

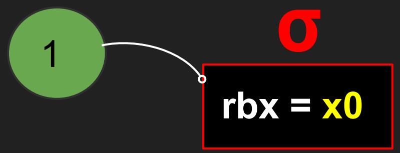
- Issues:
  - Handling memory read/writes with symbolic addresses
  - Organize the memory layout to reflect the real program

- Classic approaches
  - Single Concretization
  - Forking model
  - Merge Model
  - Flat Model
- New Approaches
  - [+]

σ = symbolic store **π** = path constraints

Single Concretization

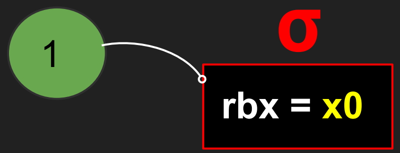
```
RBX ← X0
MOV RAX, [RBX]
```



Single Concretization

```
1| RBX ← X0
2| MOV RAX, [RBX]
```

σ = symbolic store **π** = path constraints

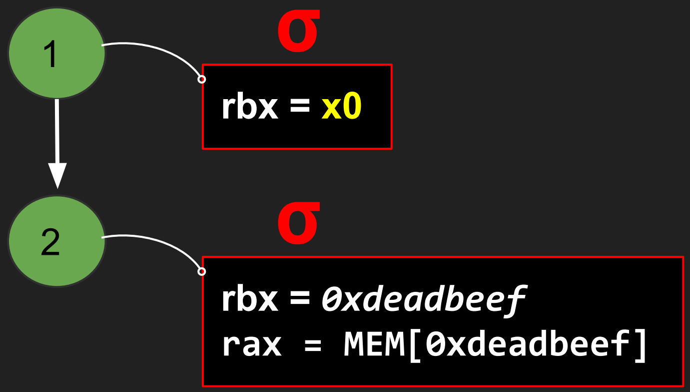


`RBX ← solver.solve_one(rbx) = 0xdeadbeef`

Single Concretization

```
1| RBX ← X0
2| MOV RAX, [RBX]
```

σ = symbolic store   π = path constraints

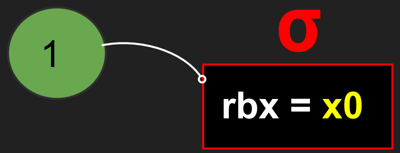


Forking Model

```
1| RBX ← X0
2| MOV RAX, [RBX]
```

σ = symbolic store

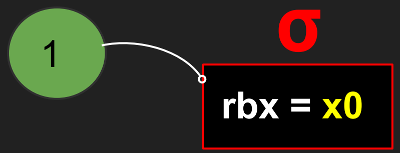
π = path constraints



σ = symbolic store π = path constraints

Forking Model

```
1| RBX ← X0
2| MOV RAX, [RBX]
```



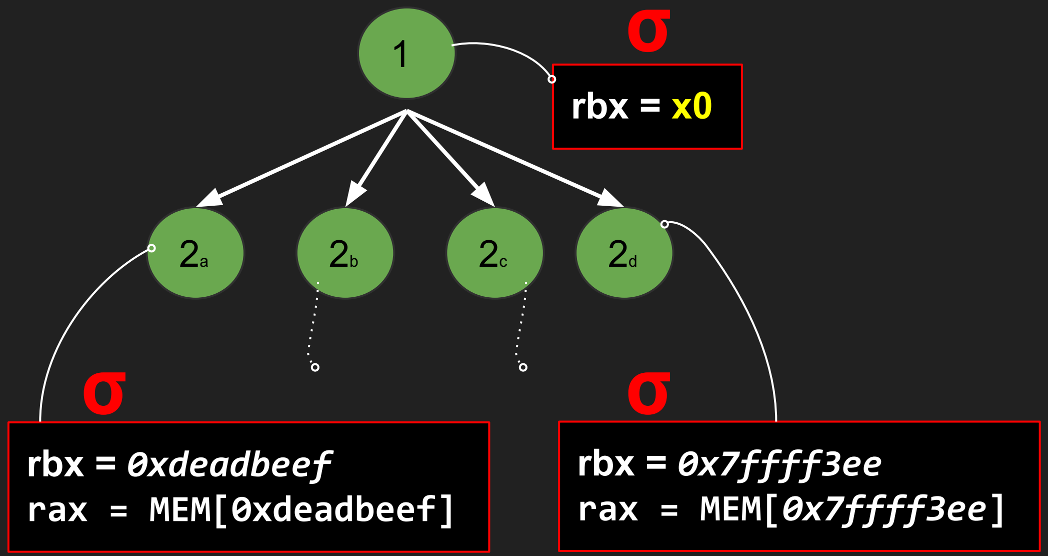

```
RBX ← solver.solve(rbx) = [0xdeadbeef,
                           0x41414100,
                           0x7fffffff,
                           0x7ffff3ee]
```

σ = symbolic store

π = path constraints

Forking Model

```
1| RBX ← X0
2| MOV RAX, [RBX]
```

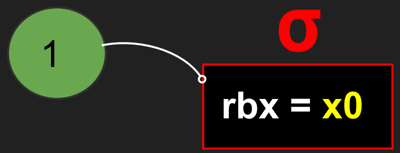


σ = symbolic store

π = path constraints

Merge Model

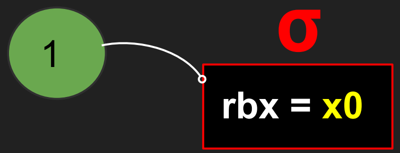
```
1| RBX ← X0
2| MOV RAX, [RBX]
```



Merge Model

```
RBX ← X0
MOV RAX, [RBX]
```

σ = symbolic store **π = path constraints**

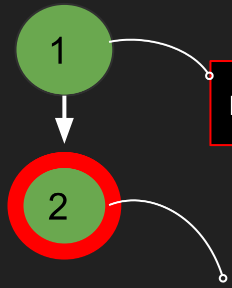



```
RBX ← solver.solve(rbx) = [0xdeadbeef,
                            0x41414100]
```

σ = symbolic store π = path constraints

Merge Model

```
1| RBX ← X0
2| MOV RAX, [RBX]
```



σ

rbx = x0

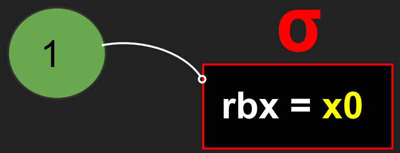
σ

rbx = x0
rax = ITE(x0 == *0xdeadbeef*, ***MEM****[0xdeadbeef]*,
ITE(x0== *0x41414100*
MEM*[0x41414100], ….)*

σ = symbolic store   π = path constraints

Flat Model + SMT Array Theory

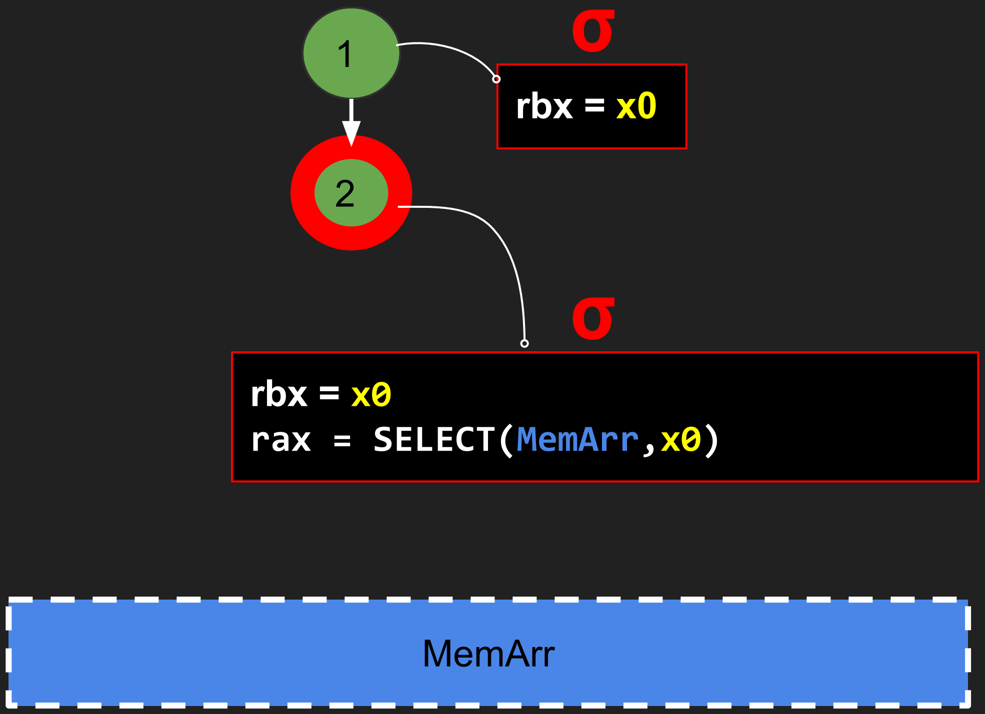
```
1| RBX ← X0
2| MOV RAX, [RBX]
```



σ = symbolic store   **π** = path constraints

Flat Model + SMT Array Theory

```
1| RBX ← X0
2| MOV RAX, [RBX]
```



(Memory as array)

Store Management

- How to organize memory and how to support symbolic memory operations? (read/writes)

- New Approaches
  - **[2012] Unleashing Mayhem on binary code**
    - Partial merge memory model (i.e., concretize writes, keep read symbolic)
    - <u>Many</u> optimizations over symbolic read reasoning
      - Boundary refinements, ITE predicates organized as ISTs
  - [2017] MEMTHINK: Rethinking pointer reasoning in symbolic execution
  - [2019] A Segmented Memory Model for Symbolic Execution
  - [2020] Relocatable Addressing Model for Symbolic Execution

Store Management

- How to organize memory and how to support symbolic memory operations? (read/writes)

- New Approaches
    - [2012] Unleashing Mayhem on binary code
    - **[2017] MEMTHINK: Rethinking pointer reasoning in symbolic execution**
        - Flat memory model. Never concretize memory addresses, keep them symbolic and use ITE expressions.
    - [2019] A Segmented Memory Model for Symbolic Execution
    - [2020] Relocatable Addressing Model for Symbolic Execution

x = load(α)

α is getting concretized to some value, e.g., 0x7fffffff

MEMORY

Address	Value
0x7fffffff	0x23
0xdeadbeef	0x41

x = load(α)

α is getting concretized to some value, e.g., 0x7fffffff

$\boxed{x = 23}$

MEMORY

Address	Value
0x7fffffff	0x23
0xdeadbeef	0x41

x = load(α)

α is getting concretized to some values, e.g., 0x7fffffff, 0xdeadbeef

MEMORY

Address	Value
0x7fffffff	0x23
0xdeadbeef	0x41

x = load(α)

```
X = ITE( α == 0x7fffffff, 0x23,
         ITE( α == 0xdeadbeef, 0x41, 0))
```

MEMORY

Address	Value
0x7fffffff	0x23
0xdeadbeef	0x41

Always keep mapping between <u>concrete</u> memory addresses and their values

MEMTHINK APPROACH

$x = \text{load}(\alpha)$

$X = \text{ITE}(\alpha == A, 0x23, \text{ITE}(\alpha == B, C, 0))$

MEMORY

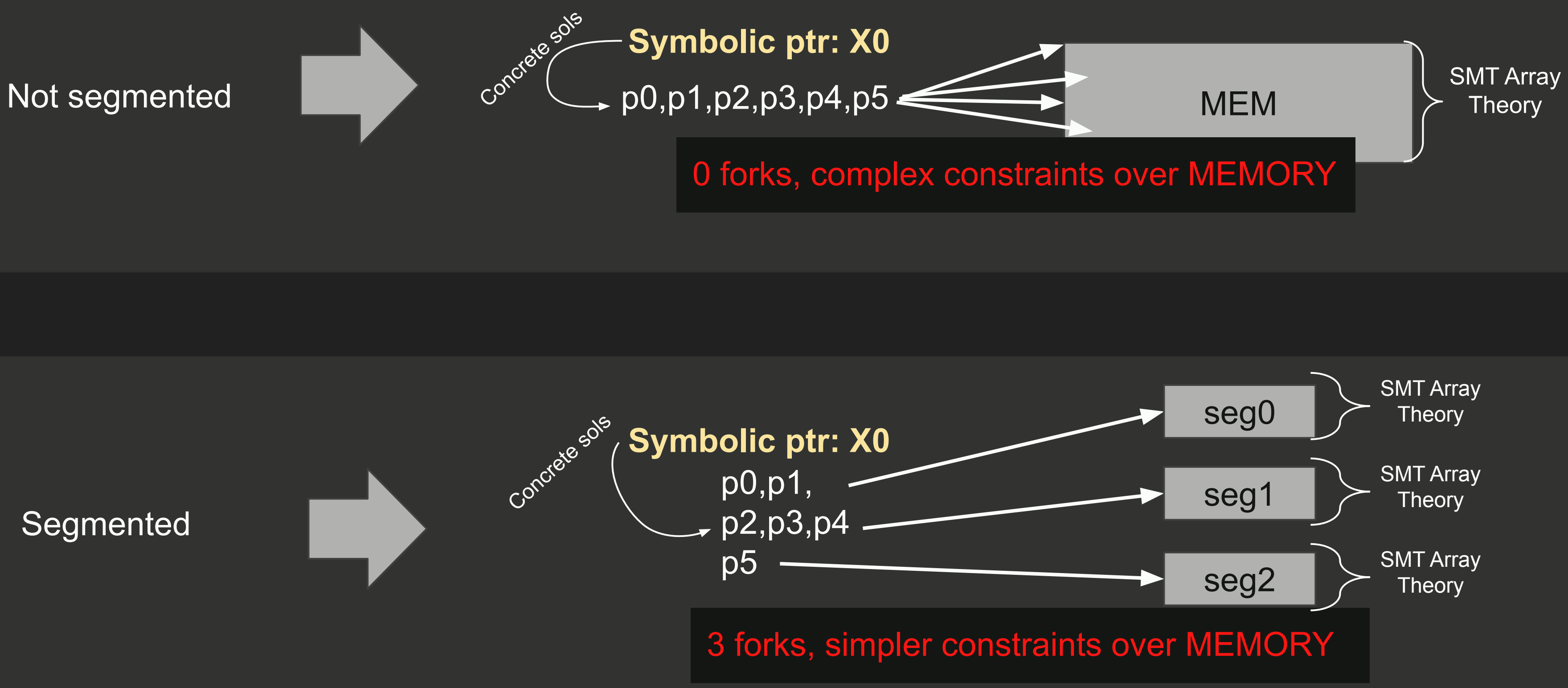
Address	Value
A	0x23
B	C

Always keep mapping between <u>symbolic</u> memory addresses and their values

Store Management

- How to organize memory and how to support symbolic memory operations? (read/writes)

- New Approaches
  - [2012] Unleashing Mayhem on binary code
  - [2017] MEMTHINK: Rethinking pointer reasoning in symbolic execution
  - **[2019] A Segmented Memory Model for Symbolic Execution**
    - Split memory into pre-computed segments, use Array Theory within the segments to handle symbolic accesses
  - [2020] Relocatable Addressing Model for Symbolic Execution



A Segmented Memory Model for Symbolic Execution

Store Management

- How to organize memory and how to support symbolic memory operations? (read/writes)

- New Approaches
    - [2012] Unleashing Mayhem on binary code
    - [2017] MEMTHINK: Rethinking pointer reasoning in symbolic execution
    - [2019] A Segmented Memory Model for Symbolic Execution
    - **[2020] Relocatable Addressing Model for Symbolic Execution**
        - Puts together the techniques of the previous two papers:
            - Keep memory addresses symbolic (but using *Array Theory*)
            - Dynamically Segmented Memory Model
                - (powered by segments relocation)

Conclusions

Conclusions

- Current state of research really pushed the boundaries of the design of original symbolic executors, unfortunately, research is VERY fragmented.
  - angr, KLEE, S2E, Mayhem, McSema, PathFinder, SymQEMU, ….
- Quality of tools and techniques comparison is debatable, no real incentive for that [26]
  - Comparison with old versions, no fair comparisons…
- Conflicts between SE techniques in different research areas is poorly understood.
  - I believe this field desperately needs more measurements research!

Conclusions

- Fundamental ideas for a modern symbolic executor:
    - **States Management:**
        - Concolic Execution
        - Efficient state merging technique
        - Dynamic path pruning
        - Flexible search strategy
        - Summarization of functions and loops
        - Symbolic execution caching [12][43]

Conclusions

- Fundamental ideas for a modern symbolic executor:
  - **States Management:**
    - Concolic Execution
    - Efficient state merging technique
    - Dynamic path pruning
    - Flexible search strategy
    - Summarization of functions and loops
    - Symbolic execution caching [12][43]
  - **Binary Processing:**
    - Hybrid binary processing (i.e., SymQemu approach)

Conclusions

- Fundamental ideas for a modern symbolic executor:
  - **States Management:**
    - Concolic Execution
    - Efficient state merging technique
    - Dynamic path pruning
    - Flexible search strategy
    - Summarization of functions and loops
    - Symbolic execution caching [12][43]
  - **Binary Processing:**
    - Hybrid binary processing (i.e., SymQemu approach)
  - **Constraints Solving:**
    - Constraints reduction/caching
    - Supersolver for constraints solving

Conclusions

- Fundamental ideas for a modern symbolic executor:
  - **Environment:**
    - Concrete delegation

Conclusions

- Fundamental ideas for a modern symbolic executor:
  - **Environment:**
    - Concrete delegation
  - **Store management:**
    - New solutions based on Array Theory are promising, but need more benchmarks

Conclusions

- Fundamental ideas for a modern symbolic executor:
    - **Environment:**
        - Concrete delegation
    - **Store management:**
        - New solutions based on Array Theory are promising, but need more benchmarks
    - **Introspection capabilities:**
        - Automatically identify pitfalls of SE
        - Clear understanding of how the analysis is being progressed and how constraints are generated
        - Profilers
        - …

Future Work

- Standard benchmarking framework for a systematic comparison between techniques.

Future Work

- Standard benchmarking framework for a systematic comparison between techniques.

- RA that needs the most work:
  - **Symbolic storage**: SMT Array Theory is great, but how can we refine the balance between constraints complexity and performances?
  - **Environment interactions**: how to systematically synchronize the delegated concrete operations with the symbolic execution by balancing performances and precision?
  - **Summarization techniques**: how to improve the summarization techniques to model more complex loops and functions?

Future Work

- Standard benchmarking framework for a systematic comparison between techniques.

- RA that needs the most work:
    - **Symbolic storage**: SMT Array Theory is great, but how can we refine the balance between constraints complexity and performances?
    - **Environment interactions**: how to systematically synchronize the delegated concrete operations with the symbolic execution by balancing performances and precision?
    - **Summarization techniques**: how to improve the summarization techniques to model more complex loops and functions?

- Introspection/measurements techniques must be improved to better understand our tools

Thanks!

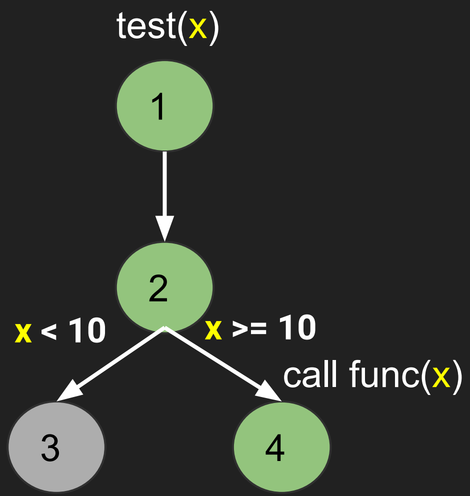
References

[1] High coverage detection of input-related security faults
[2] CUTE: A Concolic Unit Testing Engine for C
[3] DART: directed automated random testing
[4] Compositional Dynamic Test Generation
[5] Demand-driven compositional symbolic execution
[6] EXE: Automatically Generating Inputs of Death
[7] KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
[8] Loop-extended symbolic execution on binary programs
[9] All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution
[10] S2E: a platform for in-vivo multi-path analysis of software systems
[11] Automatic partial loop summarization in dynamic test generation
[12] Memoized symbolic execution
[13] Efficient State Merging in Symbolic Execution
[14] Probabilistic symbolic execution
[15] Unleashing Mayhem on Binary Code
[16] Green: reducing, reusing and recycling constraints in program analysis
[17] Targeted test input generation using symbolic-concrete backward execution
[18] Enhancing symbolic execution with veritesting
[19] Solving complex path conditions through heuristic search on induced polytopes
[20] Under-Constrained Symbolic Execution: Correctness Checking for Real Code

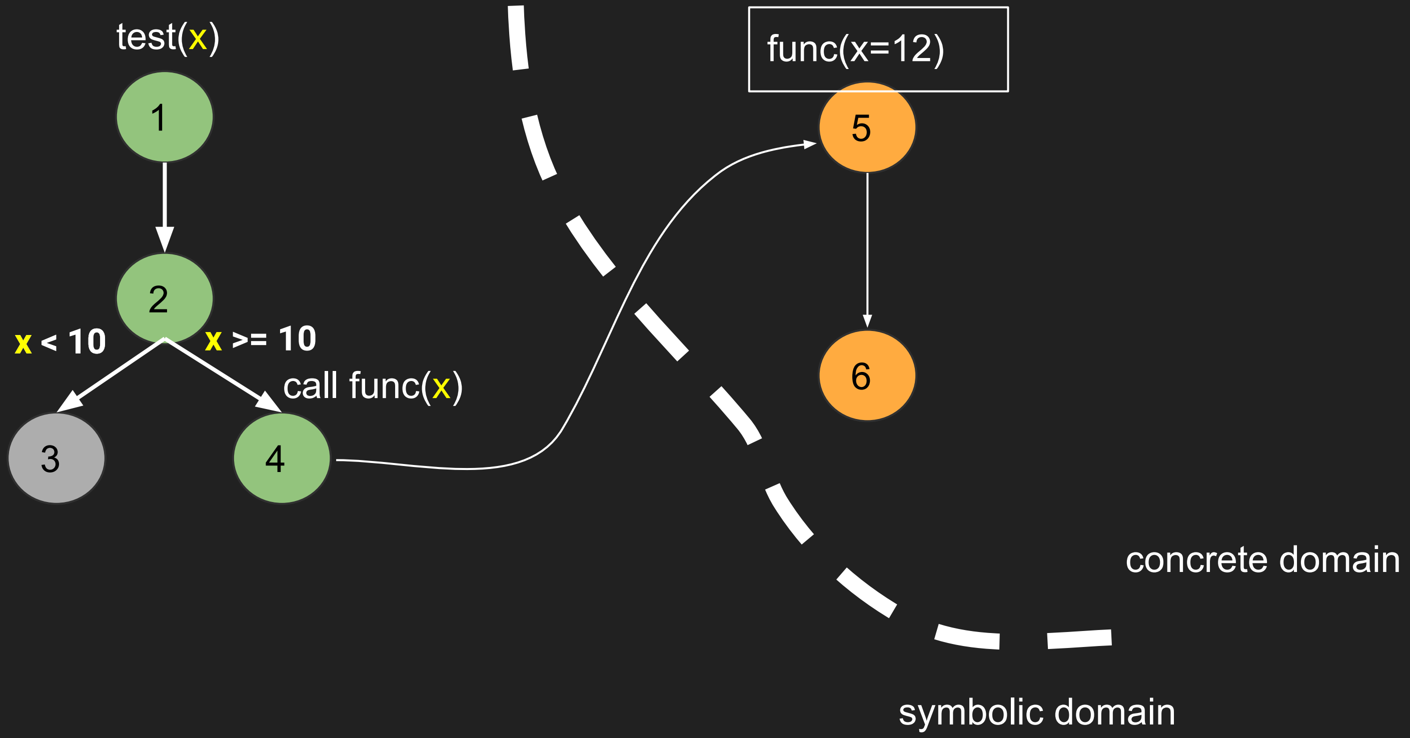
[21] Postconditioned Symbolic Execution
[22] DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing
[23] Compositional Symbolic Execution using Fine-Grained Summaries
[24] Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis
[25] SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis
[26] On the Techniques We Create, the Tools We Build, and Their Misalignments: a Study of KLEE
[27] StatSym: Vulnerable Path Discovery through Statistics-Guided Symbolic Execution
[28] Rethinking pointer reasoning in symbolic execution
[29] A Survey of Symbolic Execution Techniques
[30] Chopped symbolic execution
[31] Dynamic Path Pruning in Symbolic Execution
[32] Automatically generating search heuristics for concolic testing
[33] Boost Symbolic Execution Using Dynamic State Merging and Forking
[34] QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing
[35] Concolic testing with adaptively changing search heuristics
[36] Just fuzz it: solving floating-point constraints using coverage-guided fuzzing
[37] Enhancing Symbolic Execution by Machine Learning Based Solver Selection
[38] A segmented memory model for symbolic execution
[39] Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and its Generation
[40] SYMBION: Interleaving Symbolic with Concrete Execution
[41] Symbolic execution with SymCC: Don't interpret, compile!
[42] Making Symbolic Execution Promising by Learning Aggressive State-Pruning Strategy
[43] Running symbolic execution forever
[44] Constraint Solving with Deep Learning for Symbolic Execution
[45] Relocatable addressing model for symbolic execution

[46] Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once
[47] Boosting symbolic execution via constraint solving time prediction
[48] Fuzzy-Sat: Fuzzing Symbolic Expressions
[49] Address-Aware Query Caching for Symbolic Execution
[50] SymQEMU:Compilation-based symbolic execution for binaries
[51] TASE: Reducing Latency of Symbolic Execution with Transactional Memory
[52] Pending Constraints in Symbolic Execution for Better Exploration and Seeding
[53] WISE: Automated Test Generation for Worst-Case Complexity
[54] Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints
[55] Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths
[56] Steering Symbolic Execution to Less Traveled Paths
[57] Learning to Explore Paths for Symbolic Execution
[58] AEG: Automatic Exploit Generation
[59] SE: SELECT—a formal system for testing and debugging programs by symbolic execution
[60] Generalized Symbolic Execution for Model Checking and Testing
[61] Test Input Generation with Java PathFinder
[62] Billions and Billions of Constraints: Whitebox Fuzz Testing in Production
[63] Automated Whitebox Fuzz Testing
[64] Memory models in symbolic execution: key ideas and new thoughts
[65] Triton: A dynamic binary analysis framework (https://triton.quarkslab.com/)
[66] The Economic Impacts of Inadequate Infrastructure for Software Testing

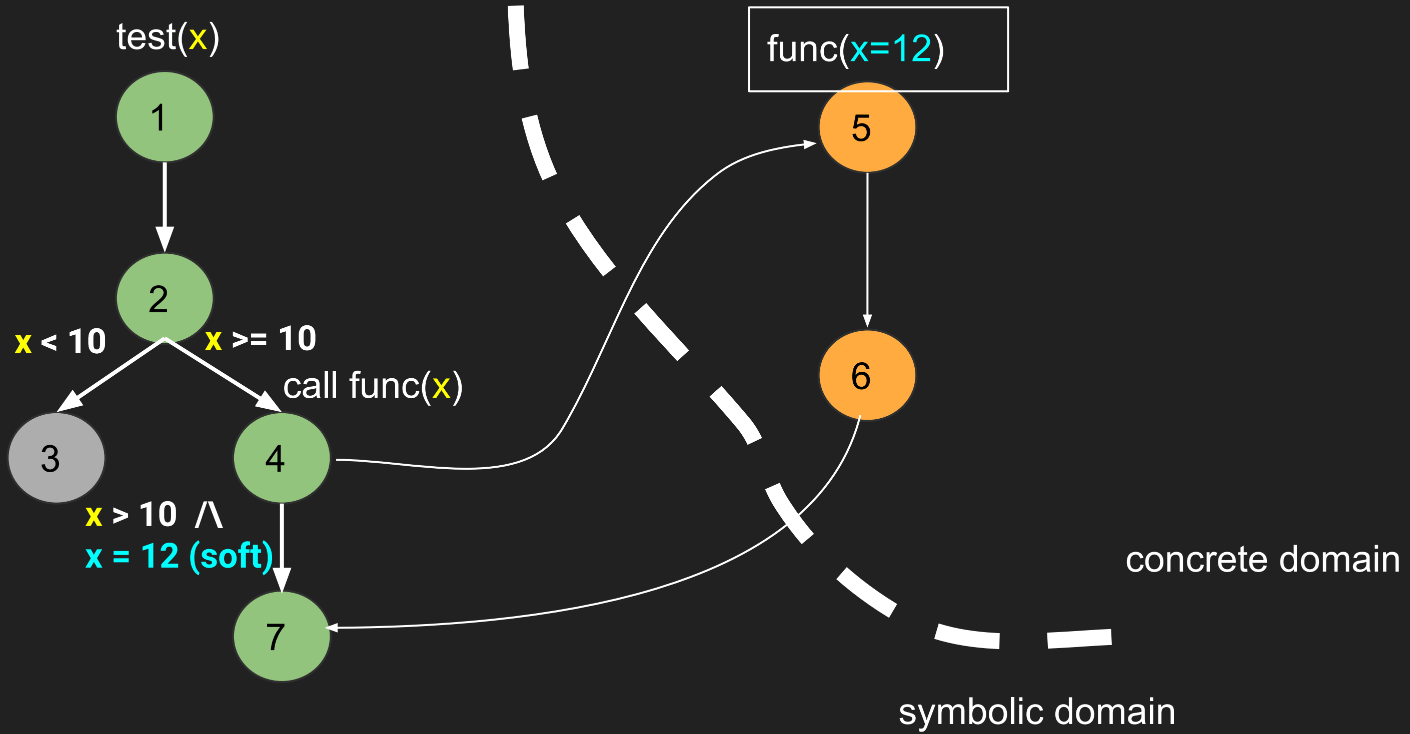
Extras



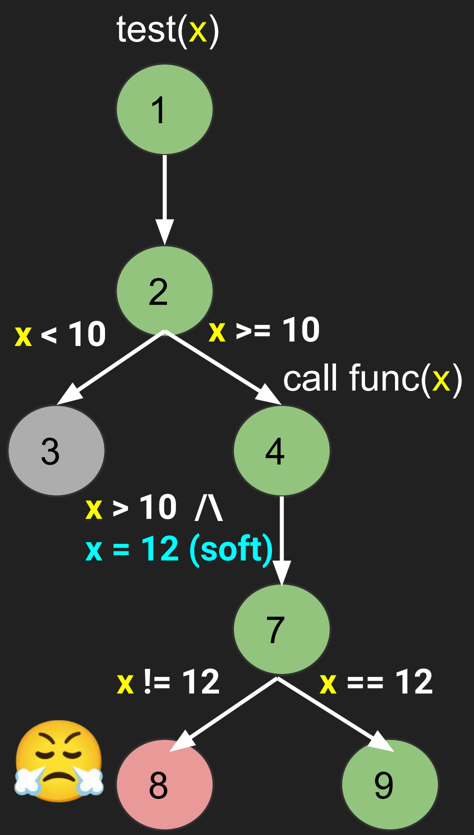
Selective symbolic execution



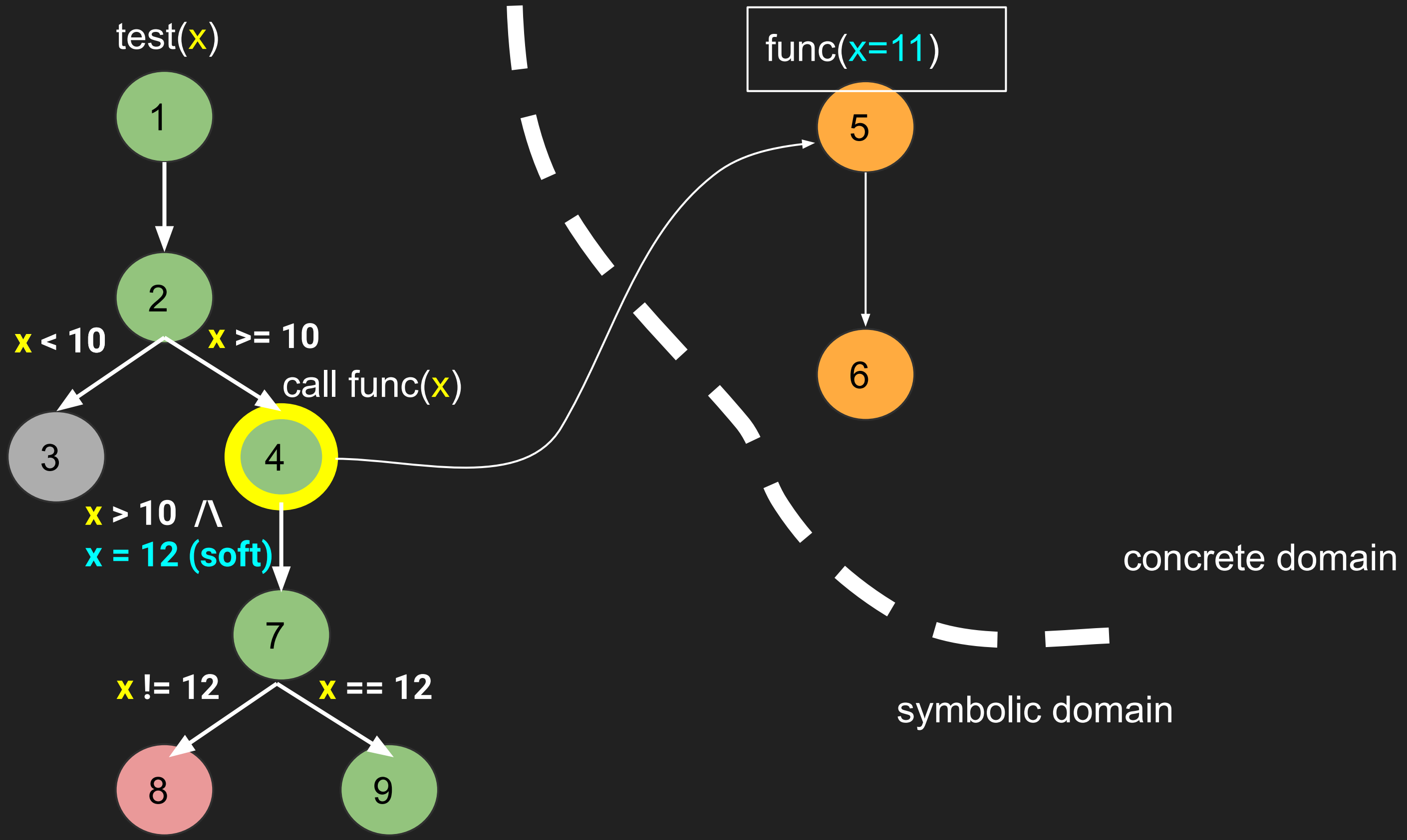
Selective symbolic execution



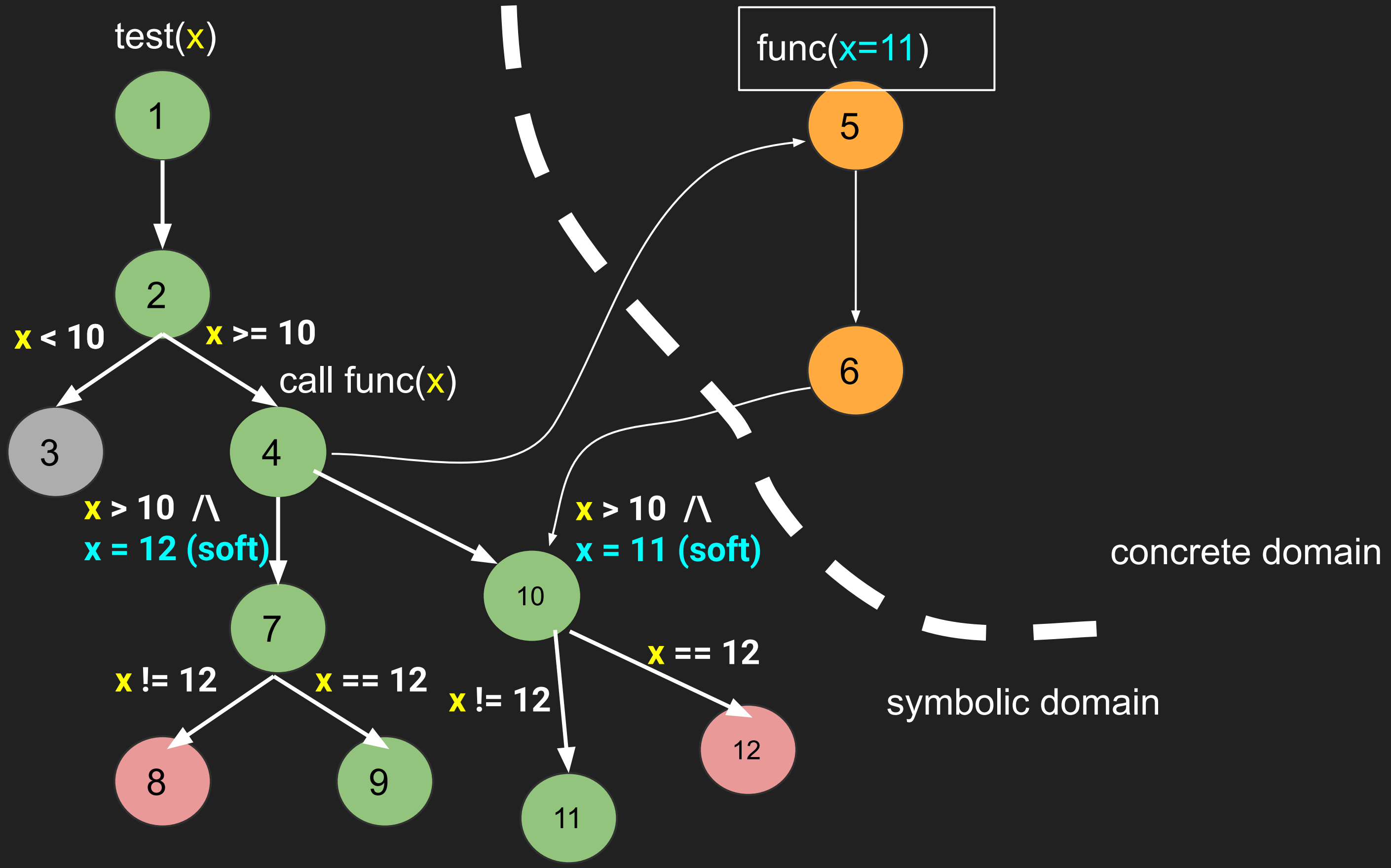
Selective symbolic execution



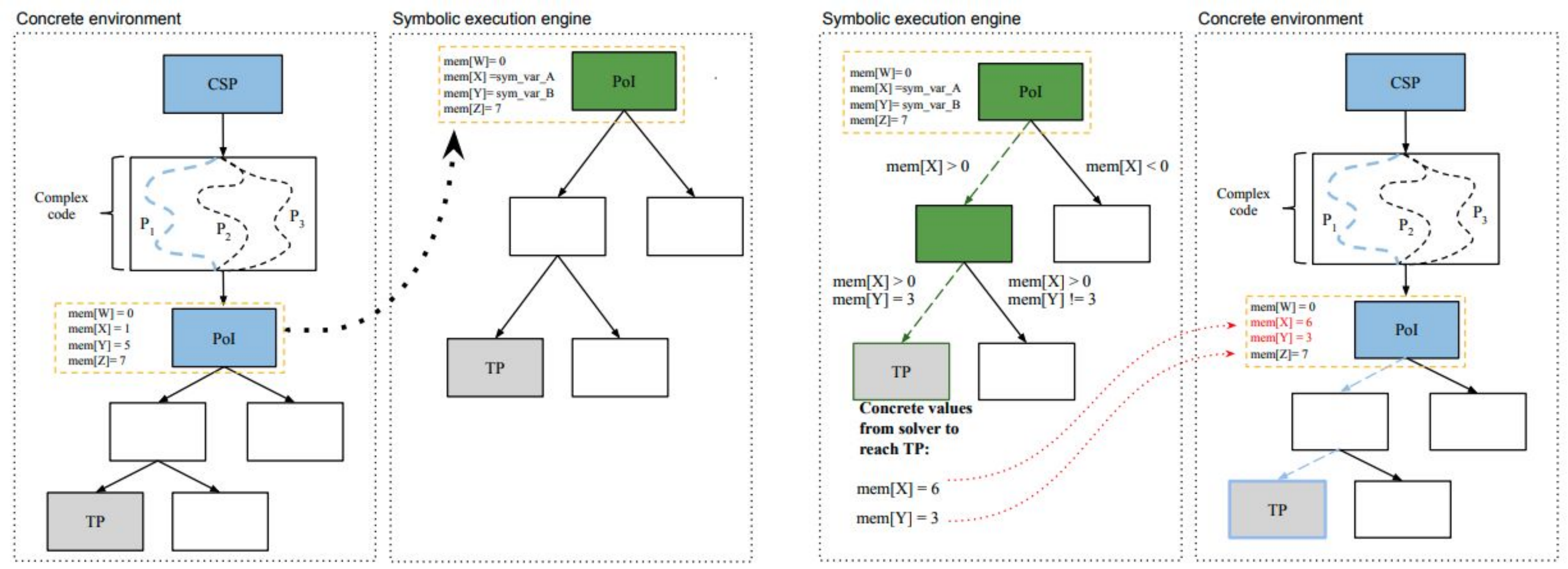
Selective symbolic execution



Selective symbolic execution



Selective symbolic execution



Interleaved symbolic execution

State-var →

State-dependent program path →

```
mode = 0
while (true){
   int curr_type = read_item_type()
   switch(curr_type){
         case 0: [...]
         case 1: [...]
         [...]
         case 23: mode = 1
         [...]
         Case 83: {
             if(mode == 1){
                memcpy(...)  // bug here
              }
         }
     }
}
```