

Exploiting Trade-offs* in Symbolic Execution for Identifying Security Bugs

SAS Workshop

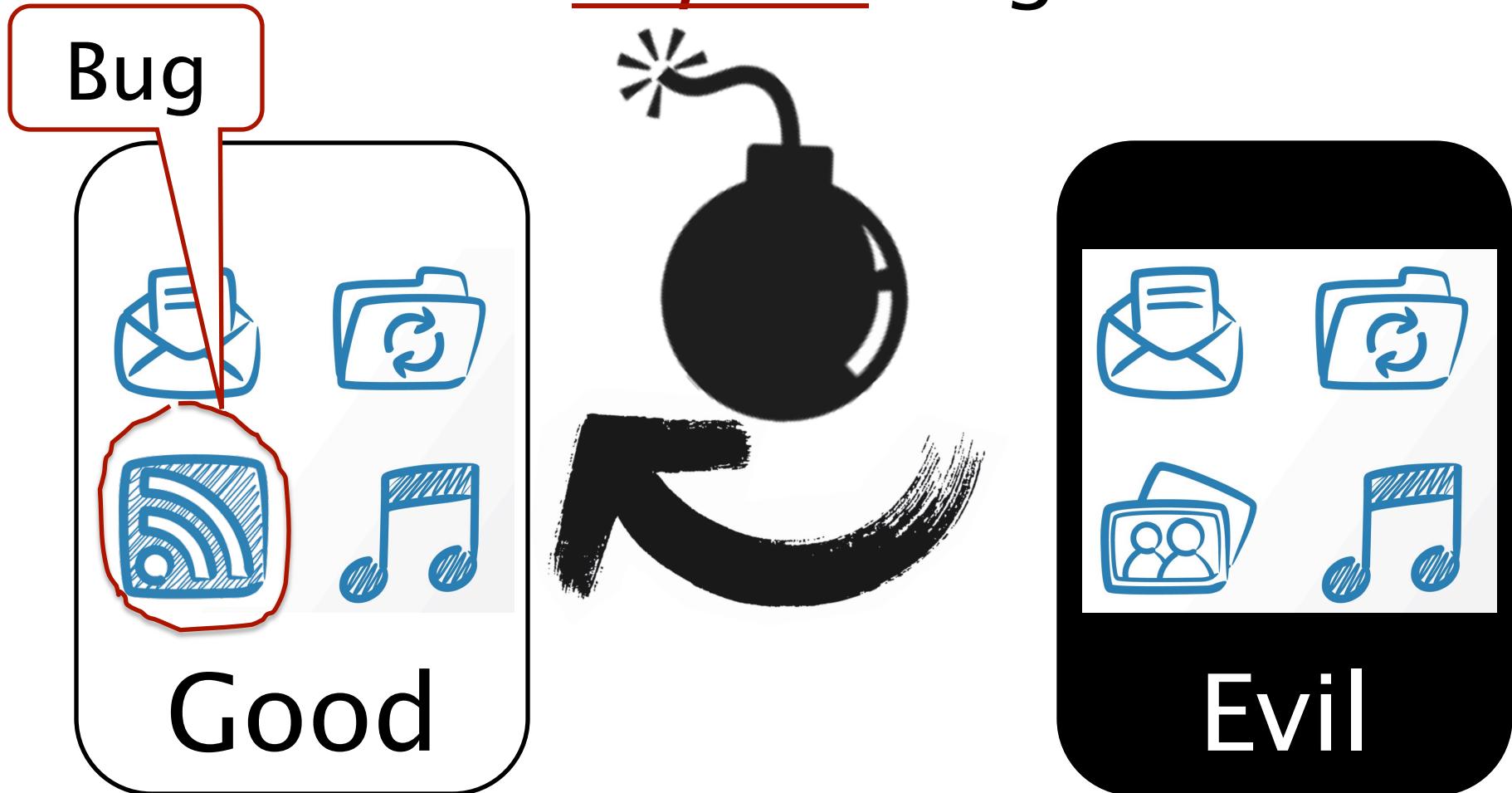
Thanassis Avgerinos
thanassis@forallsecure.co
m
September 8, 2015

***trade-off** [def. from *Merriam-Webster*]

noun

a balance achieved between two desirable but incompatible features; a compromise : *a trade-off between objectivity and relevance.*

The Security Battle to *Exploit* Bugs



OK

\$ iwconfig accesspoint

\$ iwconfig

#

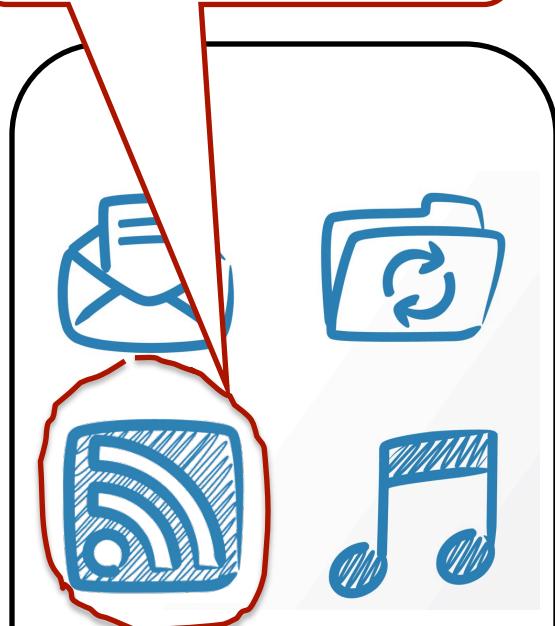
01ad 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 fce8 bfff 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 3101 50c0
2f68 732f 6868 622f 6e69
2380 5350 e189 d231

Exploit

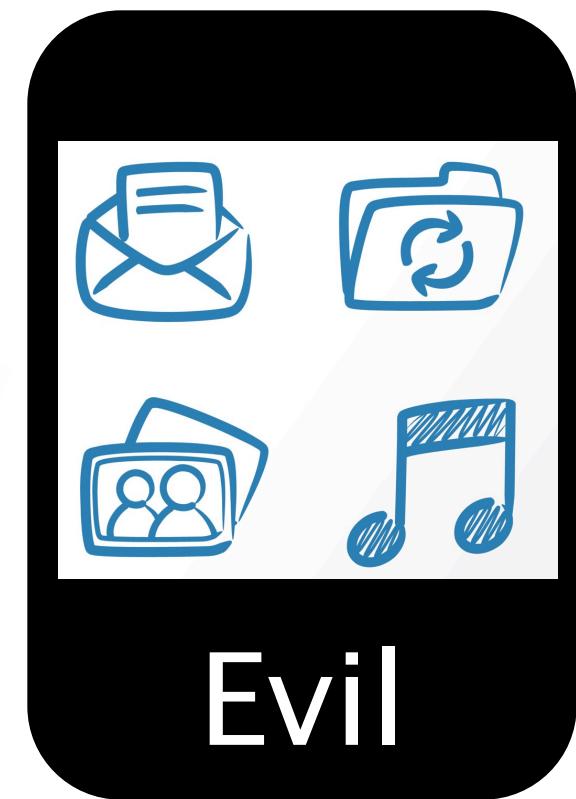
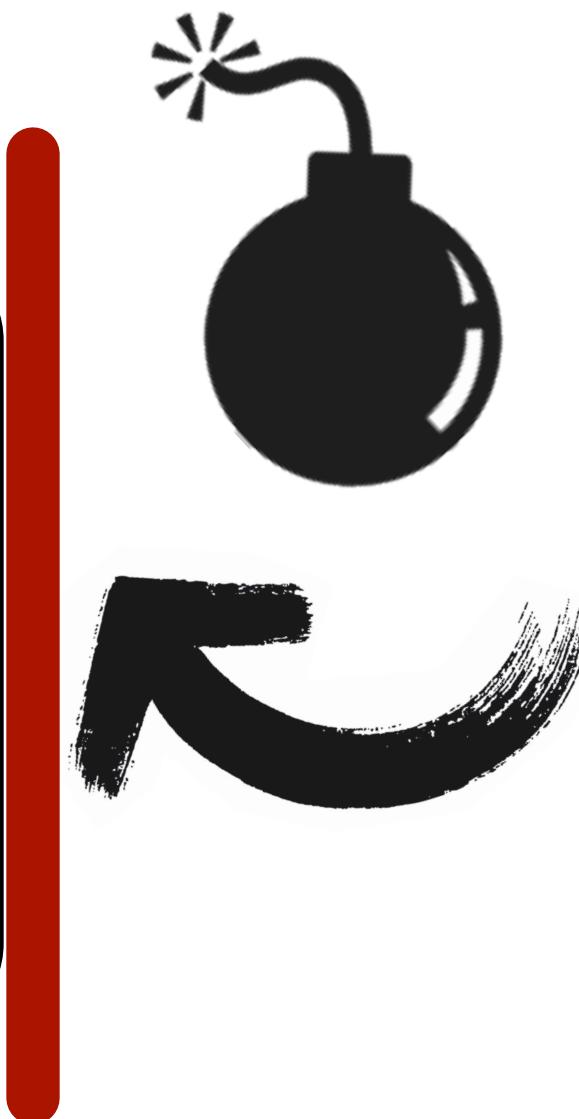
Superuser



~~Bug~~-Fixed!



Good



Evil

Fact:
Ubuntu
Linux has
over
119,000
known bugs

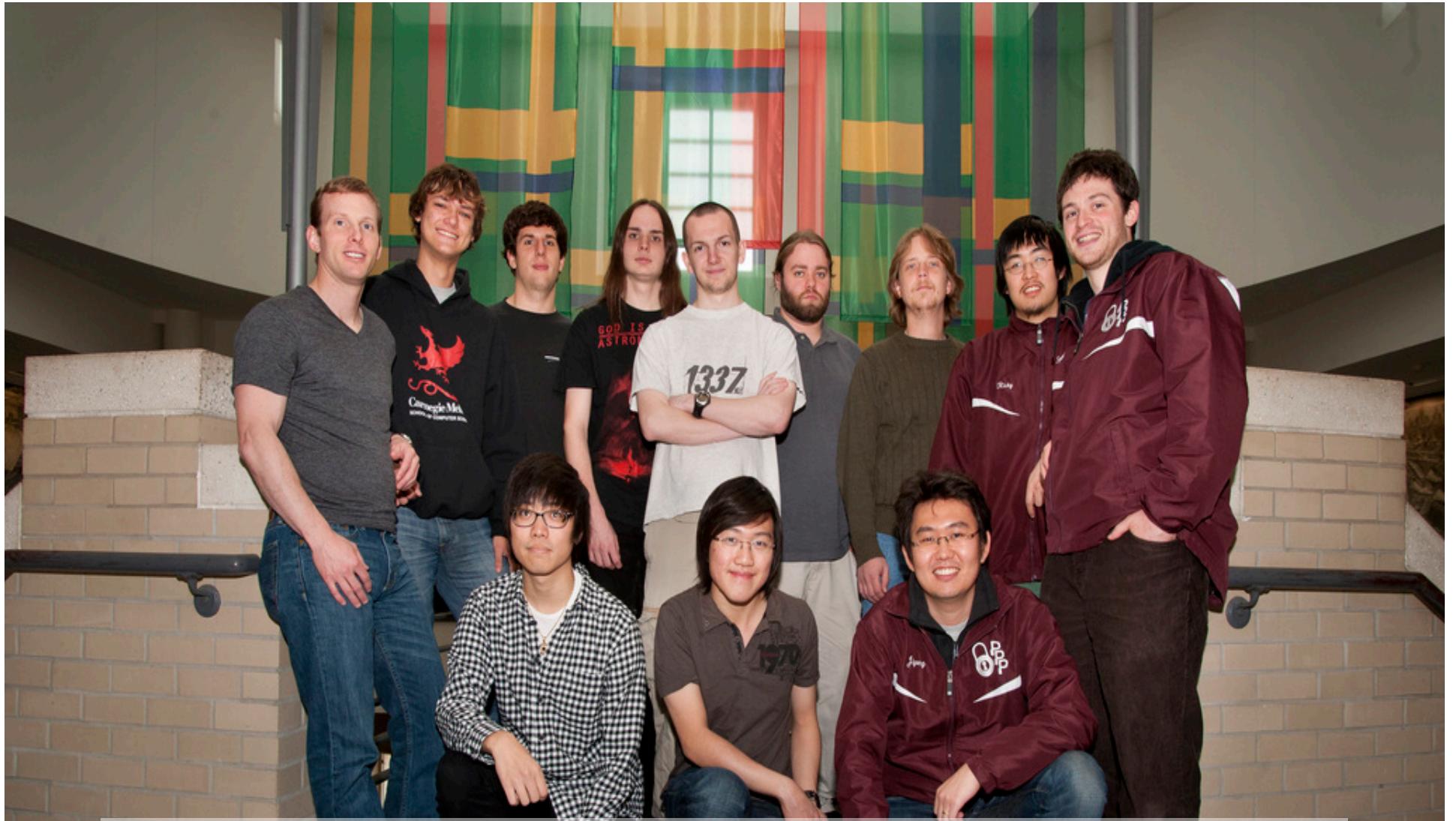


1. inp=`perl -e '{print "A"x8000}'`
2. for program in /usr/bin/*; do
3. for opt in {a..z} {A..Z}; do
4. timeout -s 9 1s
 \$program -\$opt \$inp
5. done
6. done

1009 Linux programs. 13
minutes. 52 *new* bugs in 29
programs.



Which bugs are
exploitable?



Plaid Parliament of Pwning CMU Hacking Team

Team rating

2013 2012 2011

Place Team

1 Plaid Parliament of Pwning

2 Leet More

3 Hates Irony

4 FluxFingers

5 sutegoma2

6 Eindbazen

7 disekt

8 int3pids

9 C.o.P

10 European Nopsle

Team rating

2013 2012 2011

Place Team

1 More Smoked Leet Chicken

2 Plaid Parliament of Pwning

3 Eindbazen

4 sutegoma2

Team rating

2013 2012 2011

Place	Team	Country	Rating
1	Plaid Parliament of Pwning	USA	1381.205
2	More Smoked Leet Chicken	Russia	900.705
3	Dragon Sector	Poland	617.515
4	Team ClevCode	Sweden	570.209
5	blue-lotus	China	538.325
6	Eindbazen	Netherlands	529.882
7	int3pids	Spain	502.743
8	ufologists	Russia	490.347
9	dcua	Ukraine	480.852
10	CLGT	China	472.345

DEF CON

Team	Score
PPP	15002
men in black hats	7924
raon_ASRT	7107
more smoked leet chicken	4160
routards	2503
sutegoma2	1540
shellphish	1223
Alternatives	1095
The European Nopsled Team	859
9447	506
blue lotus	441
Samurai	12
APT8	0
clgt	0
pwnies	0
pwnningyeti	0
Robot Mafia	0
shell corp	0
[Technopandas]	0
WOWHacker-BIOS	0

DEF CON

Final Scores	
Team	Score
Plaid Parliament of Pwning	11263
HITCON	7833
Dragon Sector	4421
Reckless Abandon	4020
blue-lotus	3233
(Mostly) Men in Black Hats	2594
raon_ASRT	2281
StratumAuhuur	1529
[CBA]9447	1519

DEF CON

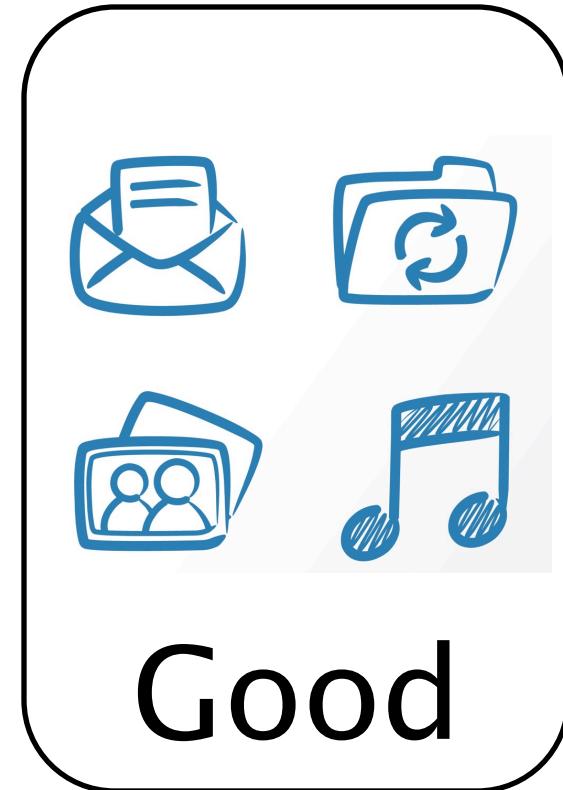
Team Name	Final Score
DEFKOR	23949
Plaid Parliament of Pwning	19896
Odaysober	17943
HITCON	13560
blue-lotus	12442
Oops	11306
Dragon Sector	11288
Samurai	10742
Shellphish	10591
LC&BC	9941
!SpamAndHex	9461
Gallopsled	8608
9447	8410
CORNDUMP	7508
Bushwhackers	7447

Unlimited size

Limited-size teams



Our Vision:
Automatically
Check the
World's
Software for
Exploitable Bugs



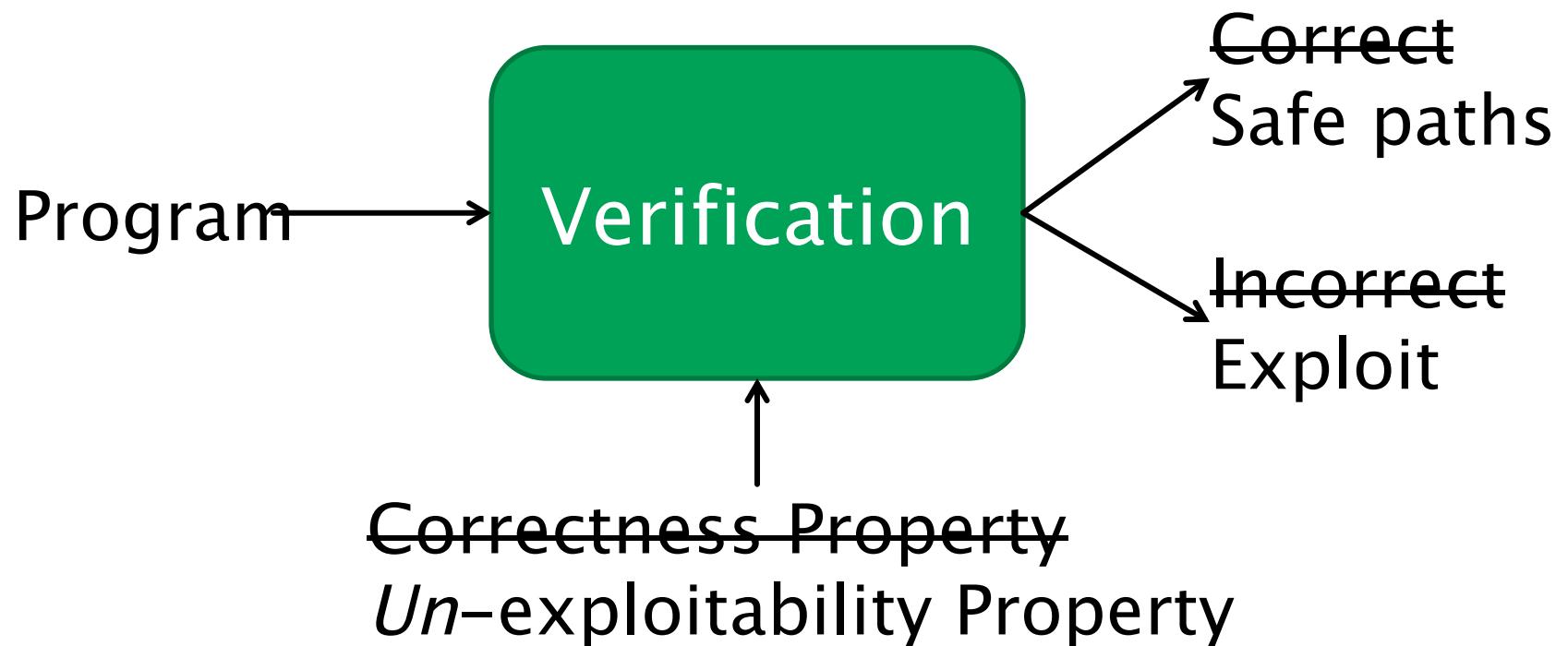
Automatic Exploit Generation with Mayhem

March 7, 2012



We owned the
machine in seconds

Verification, but with a twist



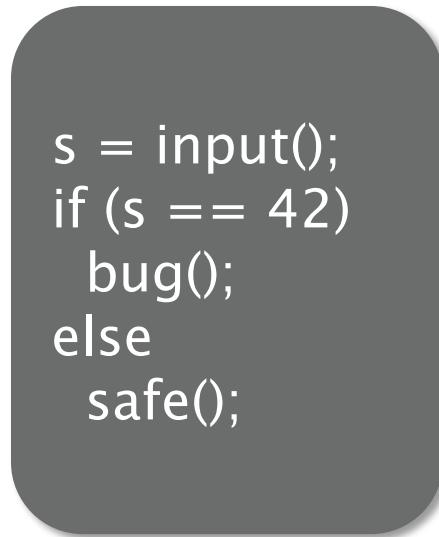
33,248 programs
152 new *exploitable* bugs

Talk Outline

- Basics of Dynamic Symbolic Execution (DSE)
- DSE for exploit generation and 3 tradeoffs:
 1. Preconditioned symbolic execution (Pruning)
 2. Memory modeling (Reduction)
 3. Veritesting (Segmentation)
- Current & Future Work

Automatically and Effectively Finding Exploitable Bugs

[Today's Talk]



Program

e.g., C/x86 assembly]



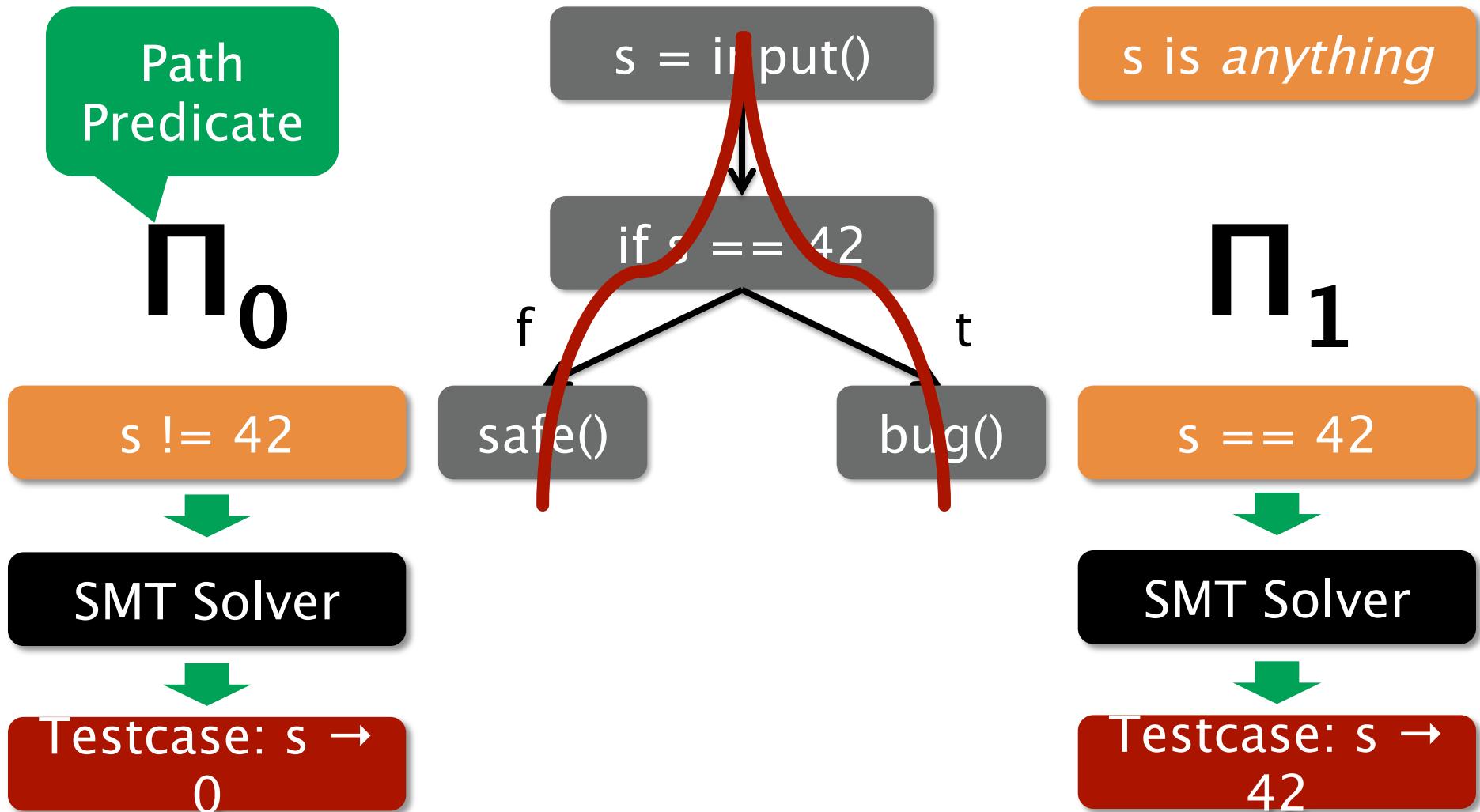
Symbolic
Execution



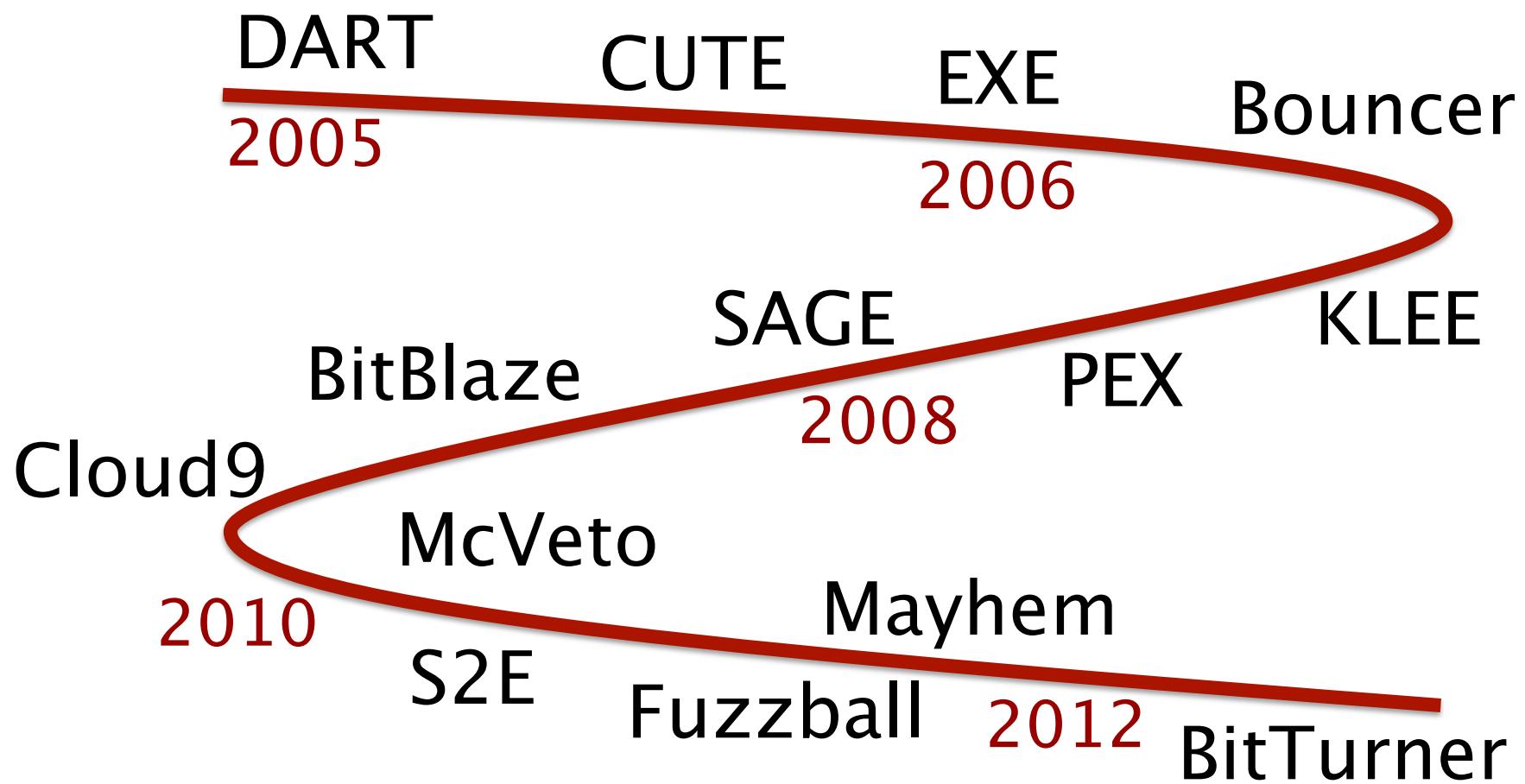
Bugs

[e.g., memory corruption]

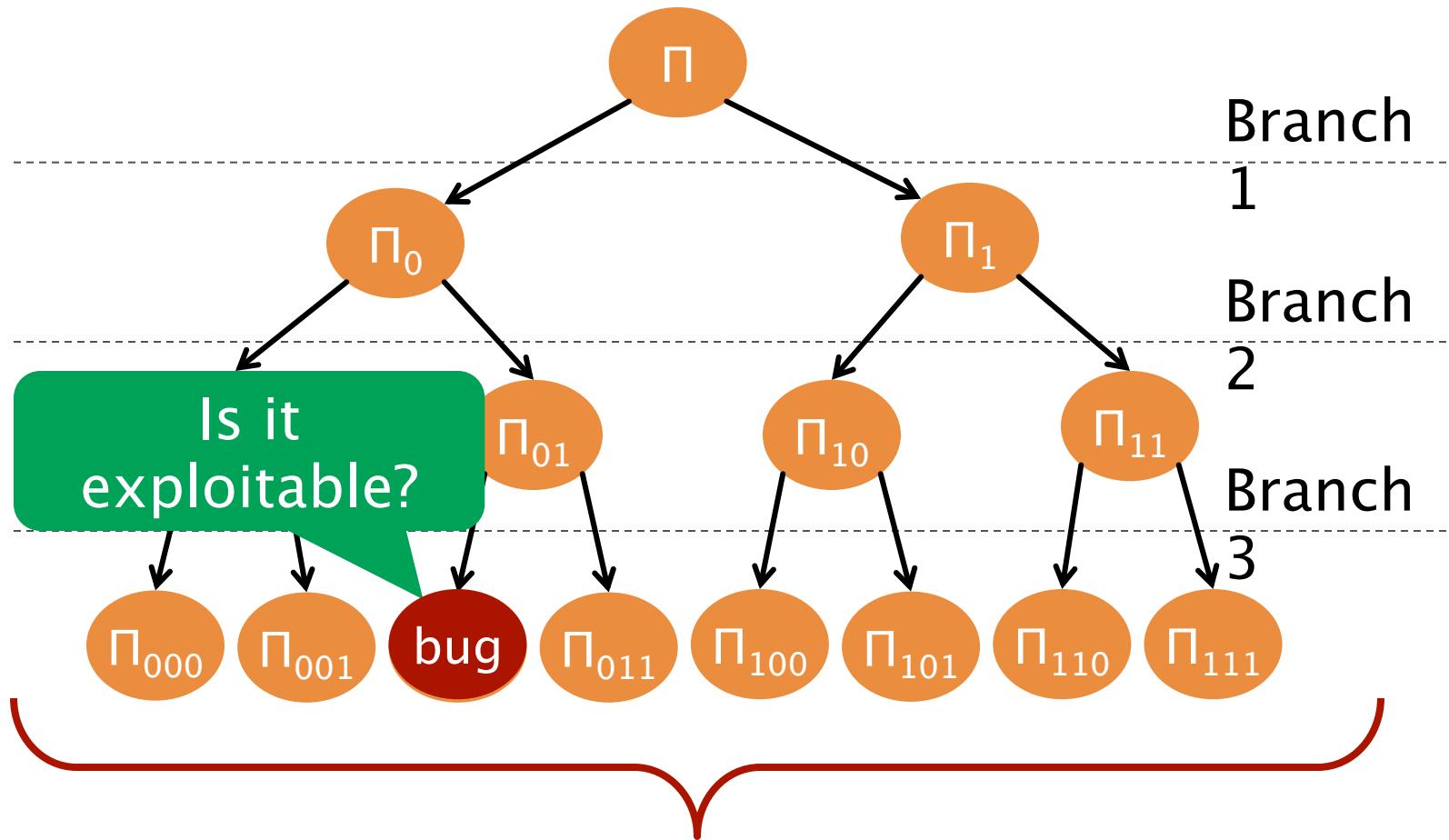
Dynamic Symbolic Execution (DSE)



Symbolic Execution Systems



Challenge: State Explosion



Every conditional branch potentially doubles
the number of states that should be checked

Finding Exploitable Bugs

- Basics of Exploitation
- Identifying Control Flow Hijacks [1, 2, 3]

- [1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao and David Brumley.
AEG: Automatic Exploit Generation. In Proceedings of the 2011 Network
and Distributed System Security Symposium (NDSS'11), Feb. 2011.
- [2] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley.
Unleashing Mayhem on Binary Code. In Proceedings of the 2012 IEEE
Symposium on Security and Privacy (Oakland'12), May 2012.
- [3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz,
Maverick Woo, and David Brumley, ***Automatic Exploit Generation.***
Communications of the ACM article (CACM'14), Feb 2014.

Security Policy: Control Flow Hijacks

Processor

EIP: 0x08048420

Effective Instruction Pointer points to next instruction to execute

Control Flow Hijack:
*EIP = Attacker Code

iwconfig: setuid wireless config

```
1 int get_info(int skfd, char ...
2 ...
3     if(iw_get_ext(skfd, ifname...
4 {
5     struct ifreq ifr;
6     strcpy(ifr.ifr_name, ifname);
7 }
```

Inputs triggering bug:
`strlen(argv[1]) > sizeof(ifr_name)`

```
8 print_info(int skfd, char *ifname,...){
9 ...
10    get_info(skfd, ifname, ...);
11 }
```

```
struct ifreq {
    char ifr_name[32]
    ...
}
```

```
12 main(int argc, char *argv[]){
13 ...
14    print_info(skfd, argv[1], NULL, 0);
15 }
```

```

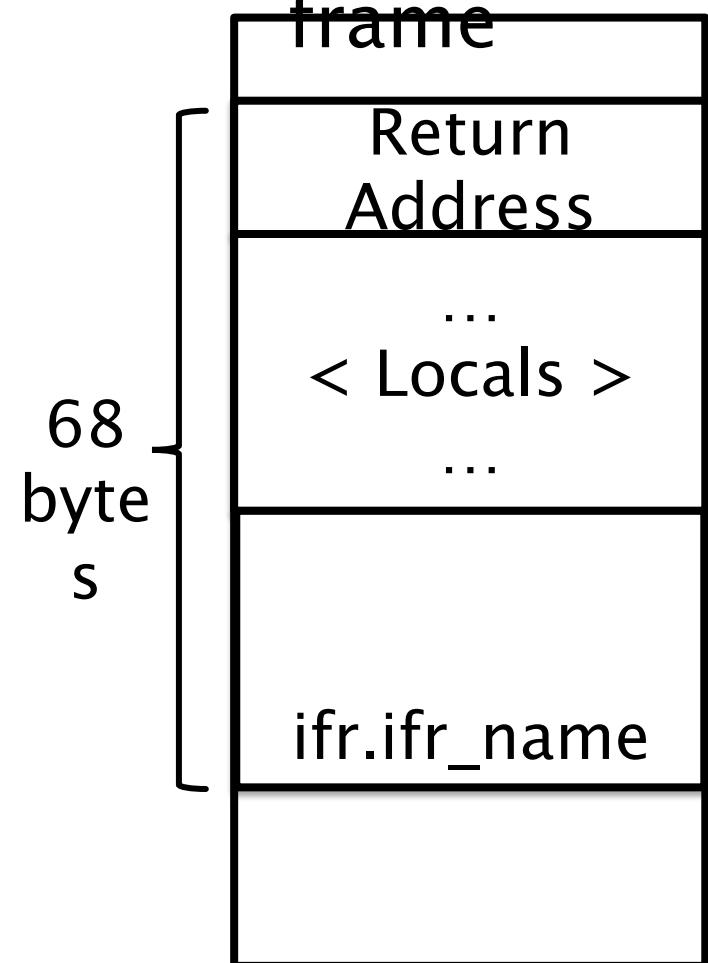
1 int get_info(int skfd, char * ifname)
2 ...
3 if(iw_get_ext(skfd, ifname, SIOCGI
4 {
5     struct ifreq ifr;
6     strcpy(ifr.ifr_name, ifname);
7 }

8 print_info(int skfd, char *ifname,...)
9 ...
10 get_info(skfd, ifname, ...);
11 }

12 main(int argc, char *argv[]){
13 ...
14 print_info(skfd, argv[1], NULL, 0)
15 }

```

get_info stack frame



Memory Layout

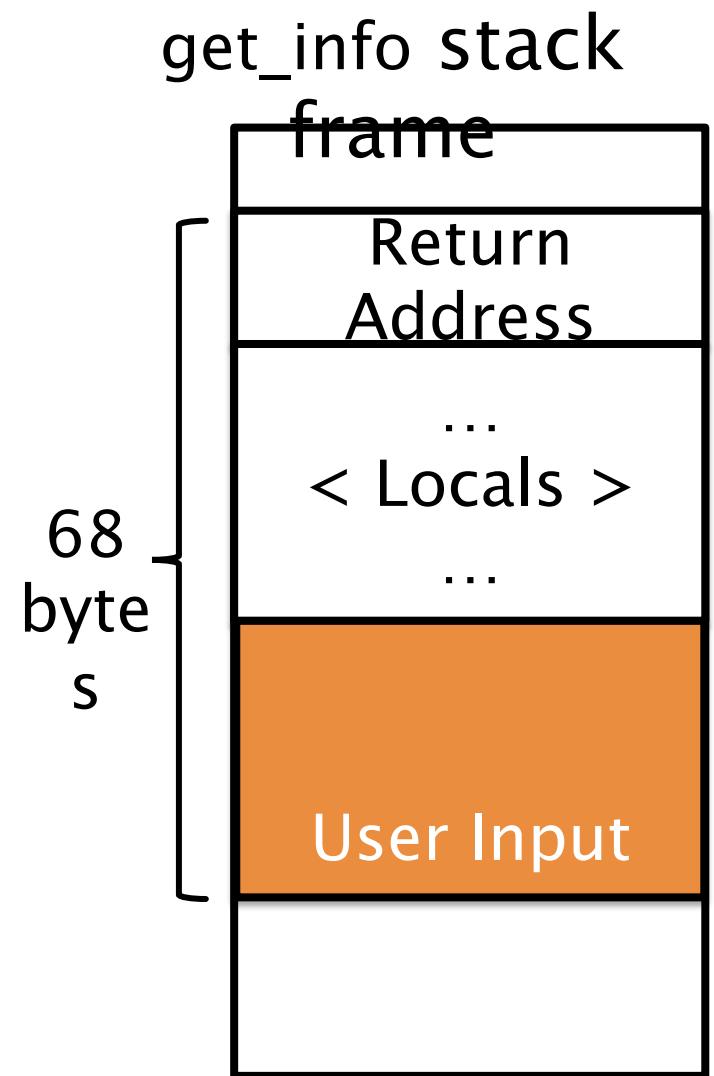
```

1 int get_info(int skfd, char * ifname)
2 ...
3 if(iw_get_ext(skfd, ifname, SIOCGI
4 {
5     struct ifreq ifr;
6     strcpy(ifr.ifr_name, ifname);
7 }

8 print_info(int skfd, char *ifname,...)
9 ...
10 get_info(skfd, ifname, ...);
11 }

12 main(int argc, char *argv[]){
13 ...
14 print_info(skfd, argv[1], NULL, 0)
15 }

```



Memory Layout

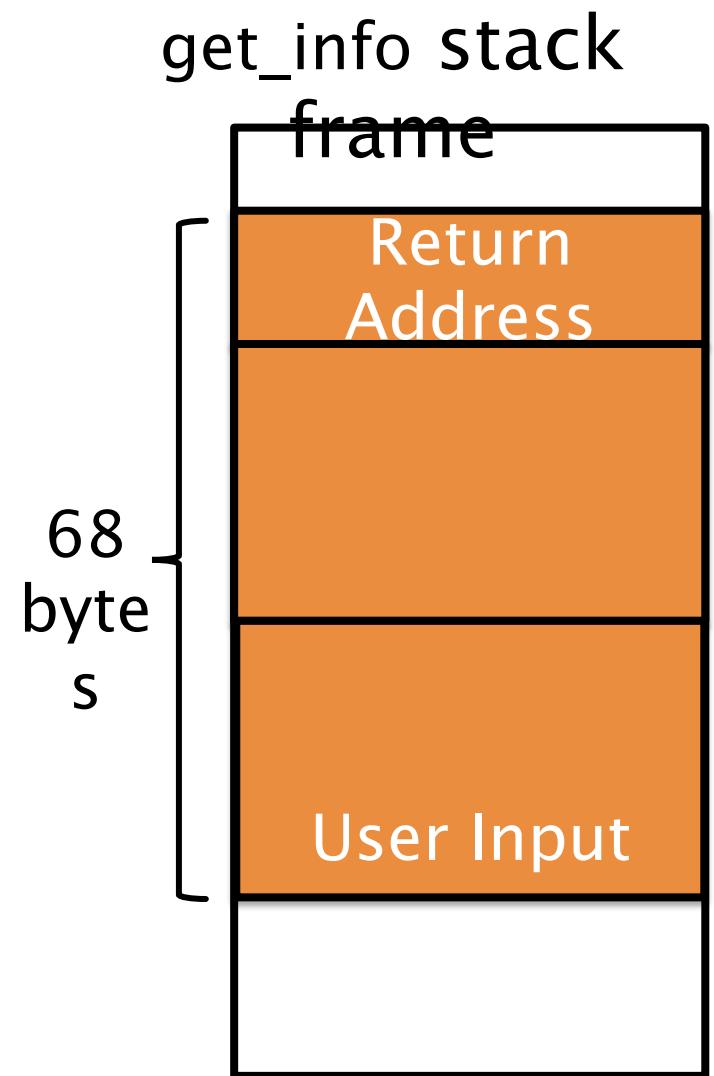
```

1 int get_info(int skfd, char * ifname)
2 ...
3 if(iw_get_ext(skfd, ifname, SIOCGI
4 {
5     struct ifreq ifr;
6     strcpy(ifr.ifr_name, ifname);
7 }

8 print_info(int skfd, char *ifname,...)
9 ...
10 get_info(skfd, ifname, ...);
11 }

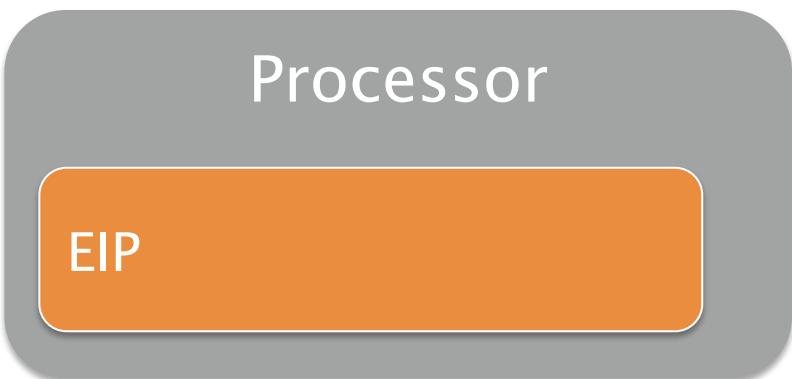
12 main(int argc, char *argv[]){
13 ...
14 print_info(skfd, argv[1], NULL, 0)
15 }

```



Memory Layout

Control Flow Hijack:
*EIP = Attacker
Code



The next instruction will
execute attacker code

get_info stack
frame

&ifr.ifr_name

\x31\xc9\xf7
\xe1\x51\x6
8\x02\x02\x
73\x68\x68\x
2f :

Memory
Layout

Identifying Control Hijack Exploits

- Checking exploitability on every statement

Path predicate `||` ensures execution can reach the current state

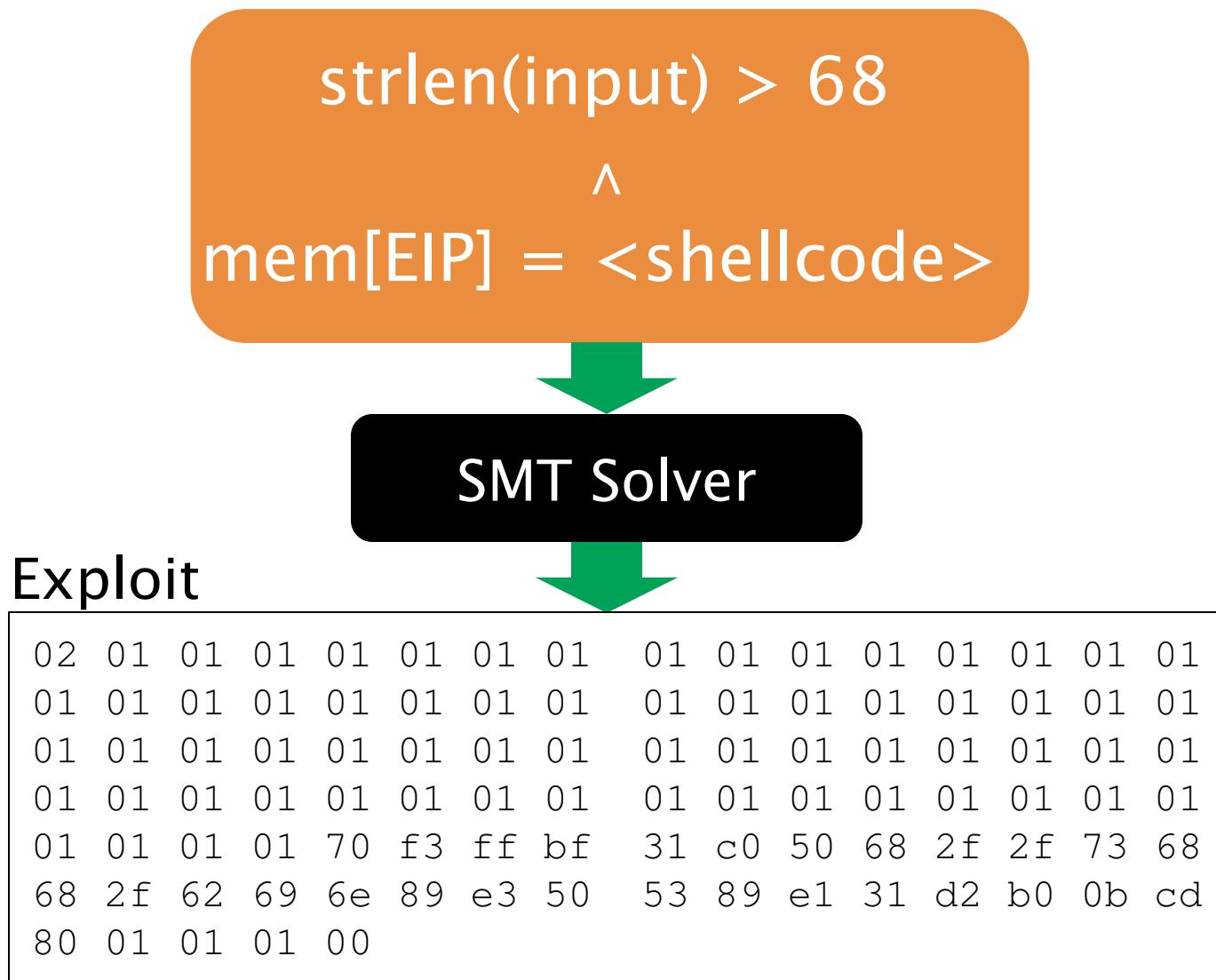
`strlen(input) > 68`

`^`

`mem[EIP] = <shellcode>`

Exploitability condition checks if
`*EIP = Attacker Code`

Generating Exploits



Note: Shellcode is parameterizable

mem[EIP] = <shellcode>

- Allows for immediate exploit hardening [1]
 - Return-Oriented Programming (ROP) shellcode can bypass common OS defenses:
 - Data Execution Prevention (DEP)
 - Address Space Layout Randomization (ASLR)
 - Q [1] is a system for automatic ROP shellcode generation with minimal code requirements

[1] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley.
Q: Exploit Hardening Made Easy. In Proceedings of the 2011 USENIX Security Symposium (USENIX'11), Aug. 2011.

First Prototype [2010]

- Built on top of KLEE¹
 - Required source (C/C++ programs)
- Checked exploitability

$\Pi \wedge \text{mem[EIP]} = \langle \text{shellcode} \rangle$

- Analyzed tens of known buggy applications
 - Found **one** exploit – iwconfig in ~5 minutes

[1] Cadar et al., KLEE: Unassisted and automatic generation of high-coverage tests [OSDI'08]

Traditional Symbolic Execution

```
strcpy(ifr_name,  
ifname);
```



```
for (i = 0 ; ifname[i] != 0 ;  
i++)  
    ifr_name[i] = ifname[i];  
ifr_name[i] = 0;
```



```
if (ifname[0] !=
```

0)

t

f

```
if (ifname[1] !=
```

0)

t

f

...

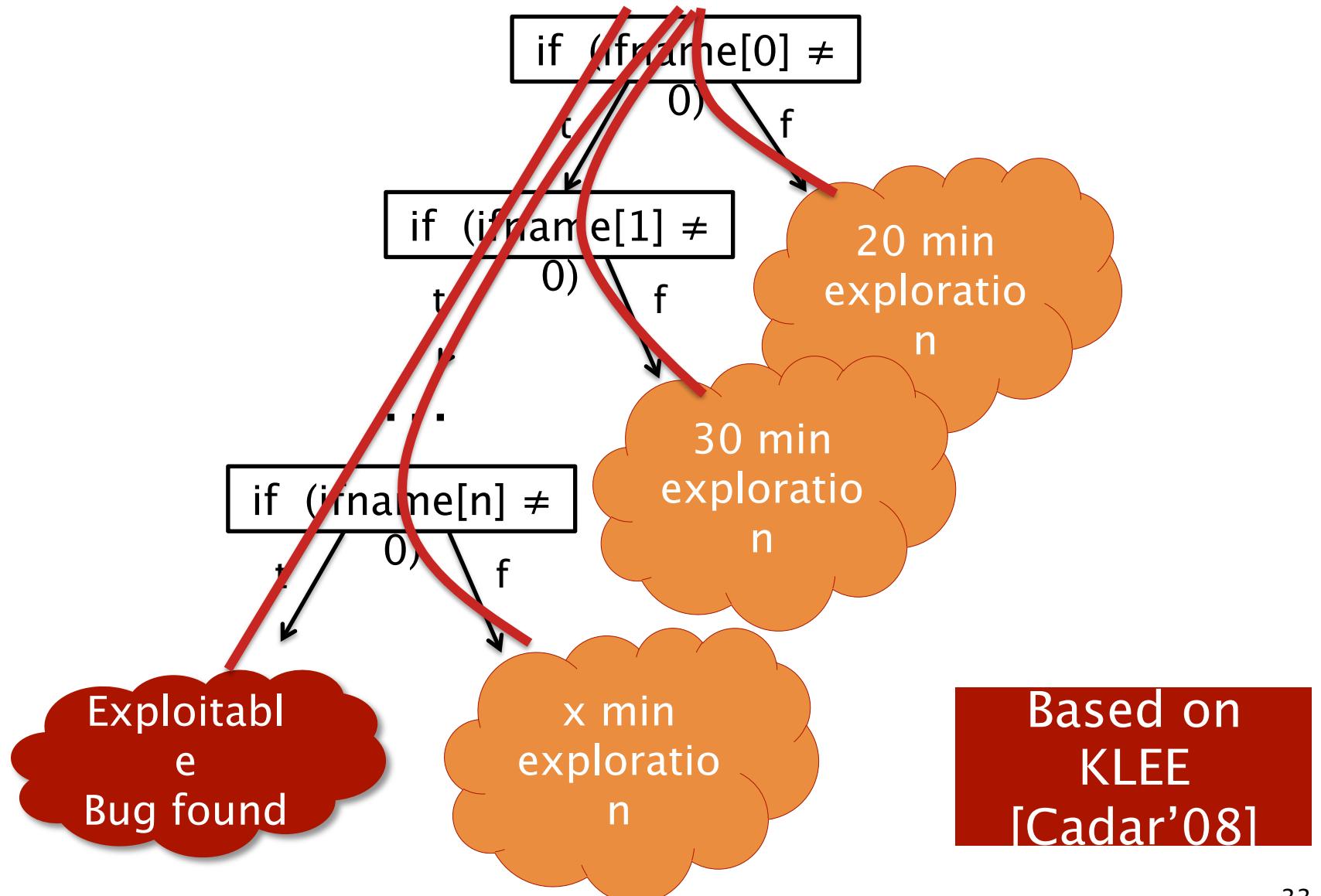
```
if (ifname[n] !=
```

0)

t

f

Traditional Symbolic Execution



Trade-off #1

DSE

- ✓ Checks all paths
- ✗ Exploits

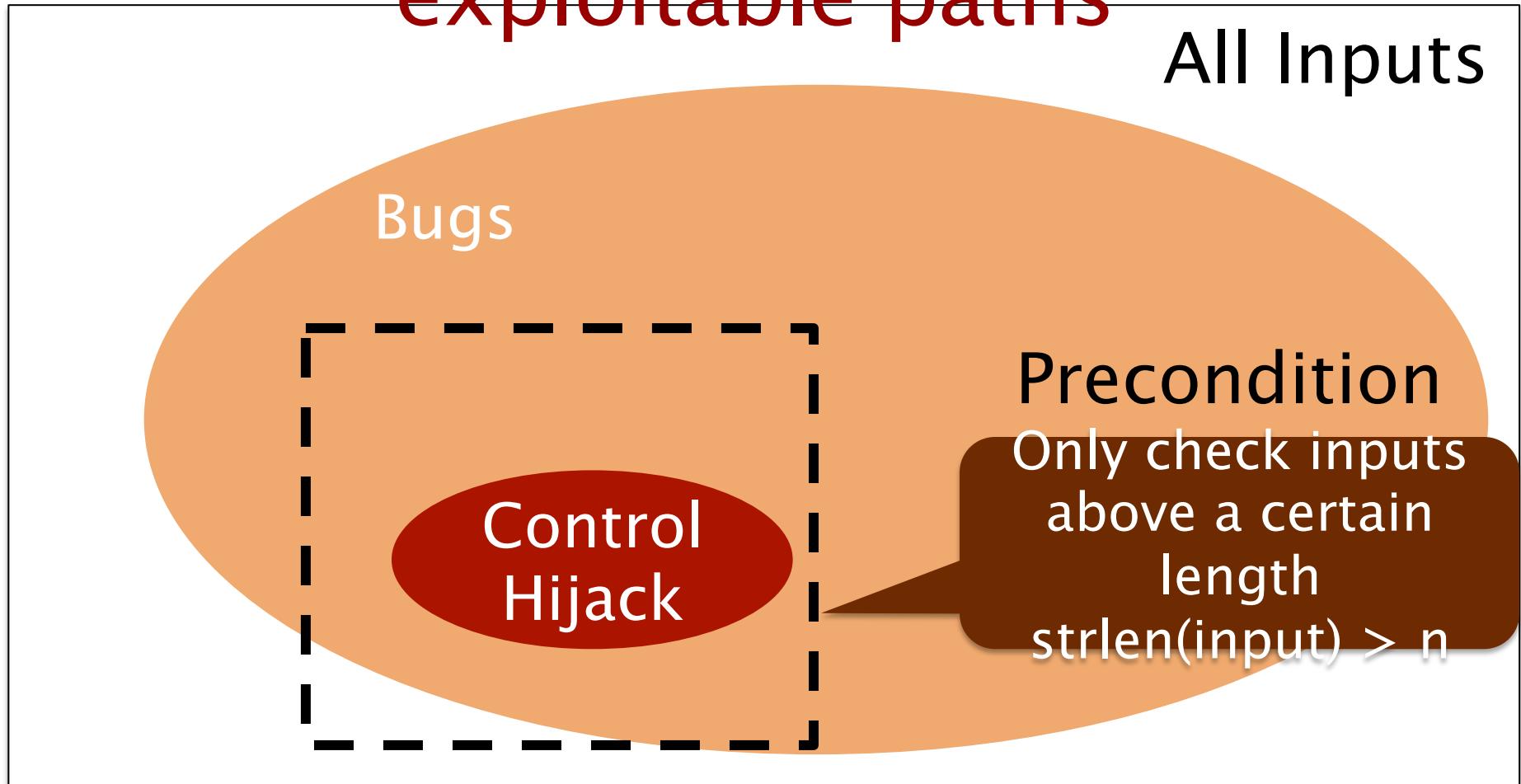
Preconditioned DSE [1]

- ✗ Checks all paths
- ✓ Exploits

Pruning: only check part of the state space

[1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao and David Brumley. **AEG: Automatic Exploit Generation**. In Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS'11), Feb. 2011.

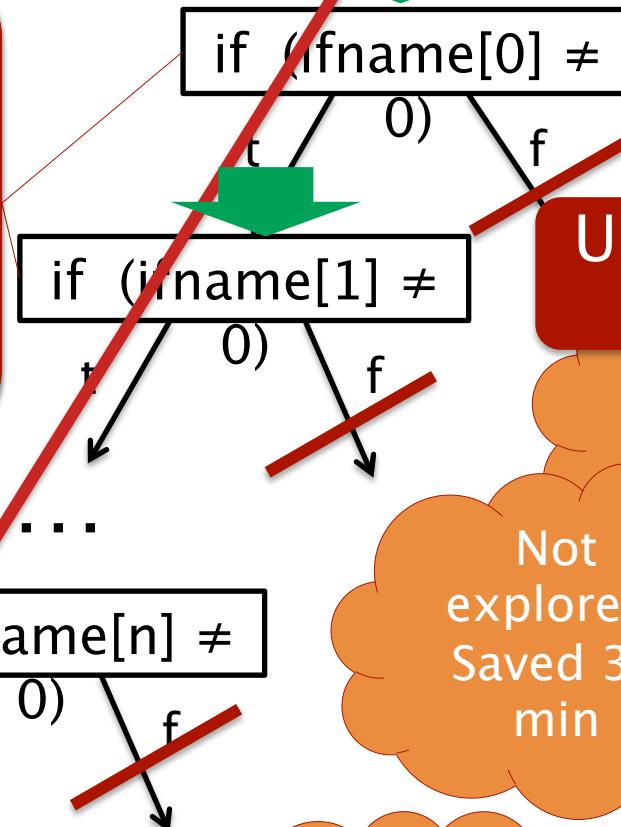
Insight: *Precondition Symbolic Execution* to focus on (likely) exploitable paths



AEG: Preconditioned Symbolic Execution

Precondition Check:

$\text{strlen}(\text{input}) > n$
^
 $\text{ifname}[0] = 0$



Unsatisfiable

Exploitabe
Bug found

Unsatisfiable

explored.
Saved 20
min

Not
explored.
Saved 30
min

Not
explored.
Saved x
min

Generating Exploits

Length precondition + heuristic  10
exploits

Name	Advisory ID	Time	Exploit Class
Iwconfig	CVE-2003-0947	1.5s	Buffer Overflow
Htget	CVE-2004-0852	< 1min	Buffer Overflow
Htget	-	1.2s	Buffer Overflow
Ncompress	CVE-2001-1413	12.3s	Buffer Overflow
Aeon	CVE-2005-1019	3.8s	Buffer Overflow
Tipxd	OSVDB-ID#12346	1.5s	Format String
Glftpd	OSVDB-ID#16373	2.3s	Buffer Overflow
Socat	CVE-2004-1484	3.2s	Format String
Expect	OSVDB-ID#60979	< 4min	Buffer Overflow
Expect	-	19.7s	Buffer Overflow

Finding Exploitable Bugs

010101010110
111010101010
101010101010
101010101010
101011111100
001101010101
010001010110
111001111000
01

SonarEye



**Symbolic
Execution**



Control Hijack

Second Prototype: Mayhem [2011]

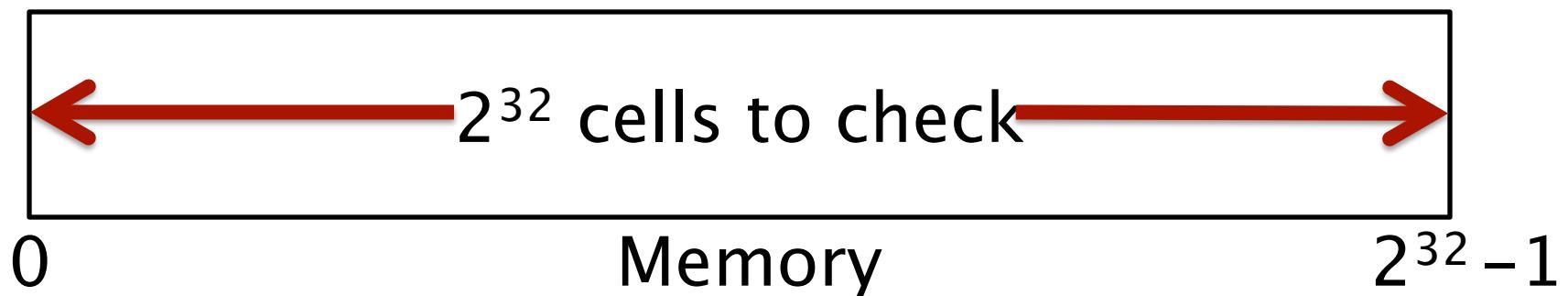
- Binary-only symbolic executor
- Checks exploitability
 - $\Pi \wedge \text{mem[EIP]} = \langle \text{shellcode} \rangle$
- No source code abstractions
 - Types, buffers, datastructures
 - Indirect jumps, partial control flow graph

One Challenge: Symbolic Indices

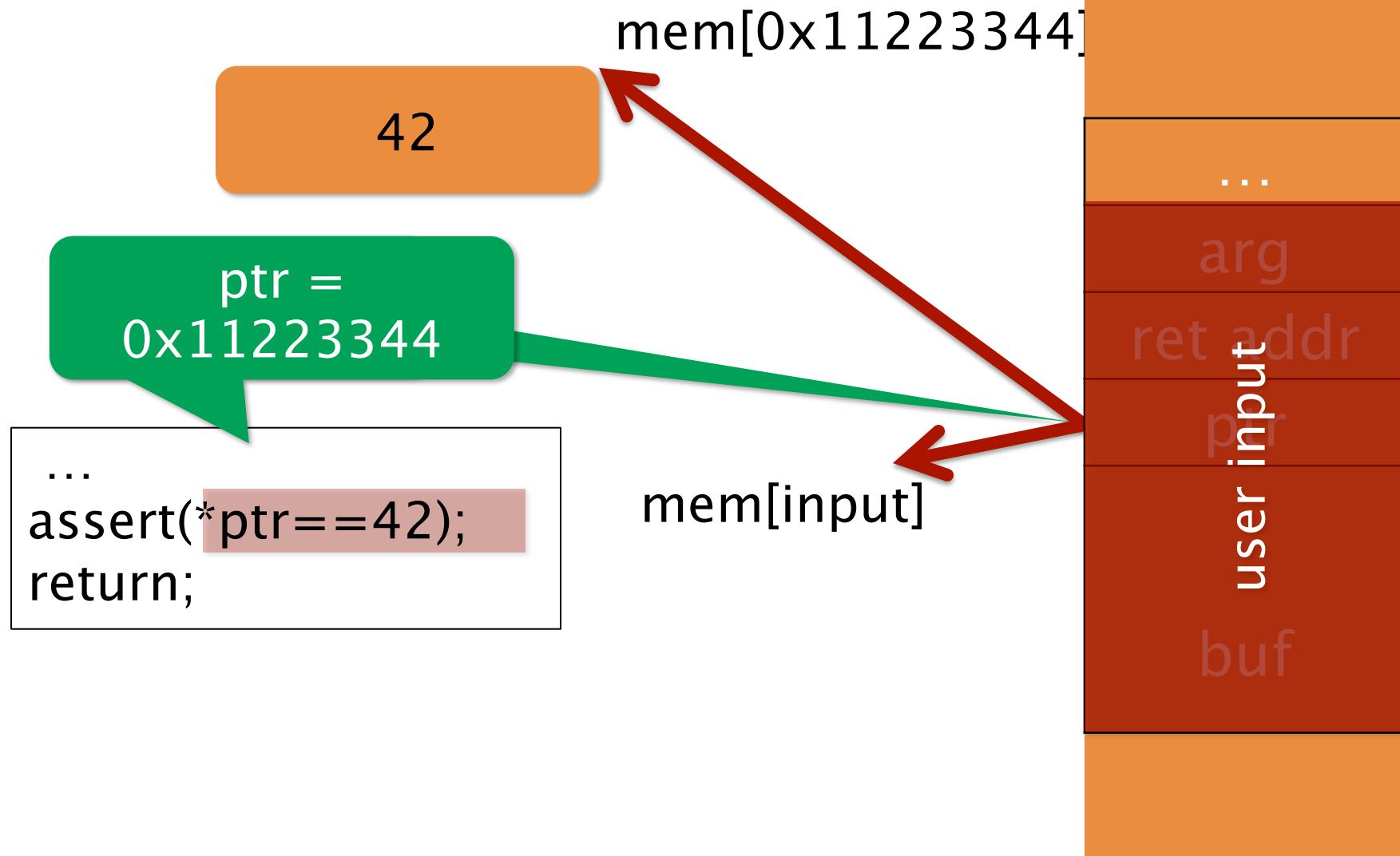
```
x := get_input();  
...  
y := mem[x];  
assert (y == 42);
```

x can be anything

Which memory cell
contains 42?



Symbolic Indices: Overwritten Pointers



Symbolic Indices: Translation Tables

```
c = get_char();
```

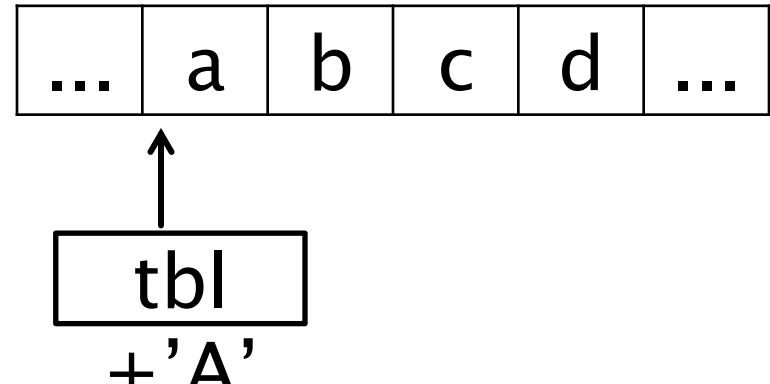
```
...
```

```
c = tolower(c);
```

```
tolower(char c){
```

```
    return c >= -128 && c < 256 ? tbl[c] : c;
```

```
}
```



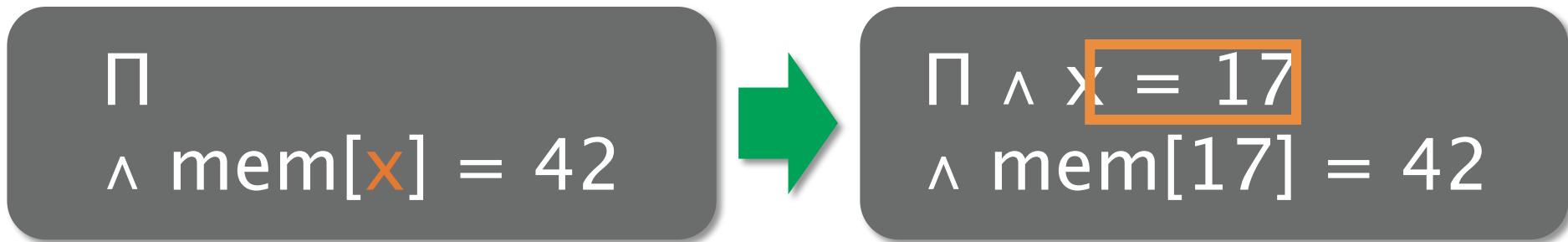
Other causes

- Parsing: sscanf, vfprintf, etc.
- Character test: isspace, isalpha, etc.
- Conversion: toupper, tolower, mbtowc, etc.
- ...

Address is
symbolic

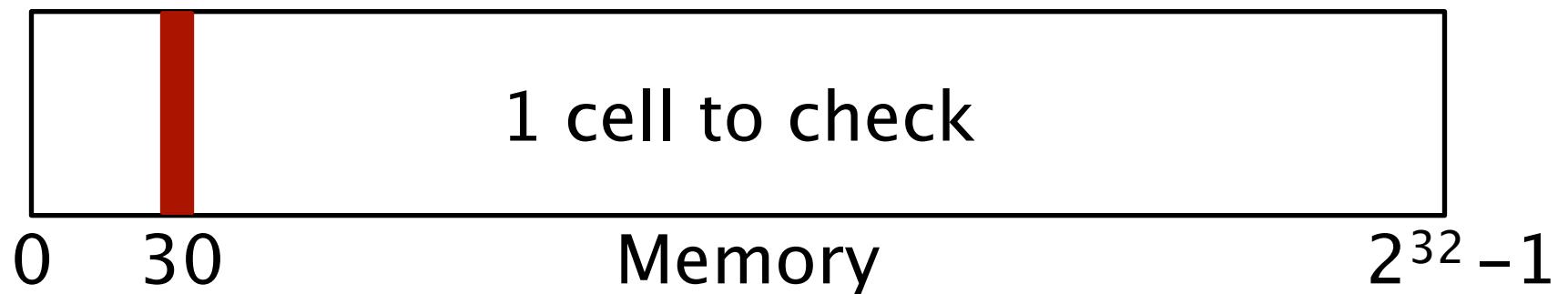
Method 1: Concretization

e.g., SAGE, DART, CUTE



- ✓ Solvable
- ✗ Exploits

Misses over 40% of
exploits



Method 2: Fully Symbolic

$$\Pi \wedge \text{mem}[x] = 42$$



$$\Pi \wedge \text{mem}[x] = 42$$

$$\wedge \text{mem}[0] = v_0 \wedge \dots \wedge \text{mem}[2^{32}-1] =$$

$$v_{2^{32}-1}$$

- ✗ Solvable
- ✓ Exploits

Trade-off #2

Concretization

- ✓ Solvable
- ✗ Exploits

Fully symbolic

- ✗ Solvable
- ✓ Exploits

Partial Memory Modeling [1]

- ✓ Solvable
- ✓ Exploits

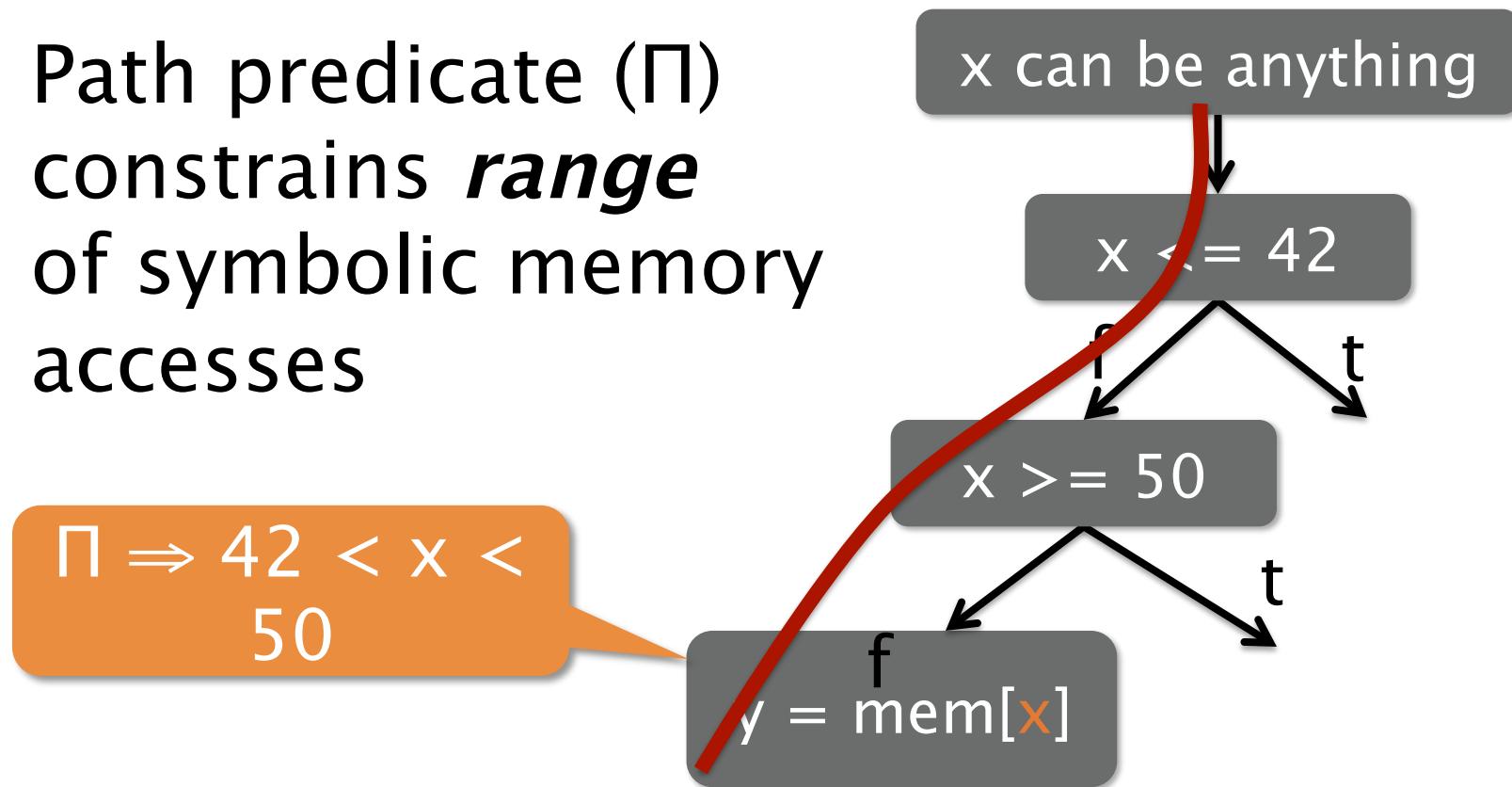
Reduce the size of memory formulas and concretize when necessary

[1] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley.

Unleashing Mayhem on Binary Code. In Proceedings of the 2012 IEEE Symposium on Security and Privacy (Oakland'12), May 2012.

Our Observation

Path predicate (Π)
constrains *range*
of symbolic memory
accesses



Use symbolic execution state to:
Step 1: Bound memory addresses referenced
Step 2: Make search tree for memory address
values

Step 1 — Find Bounds

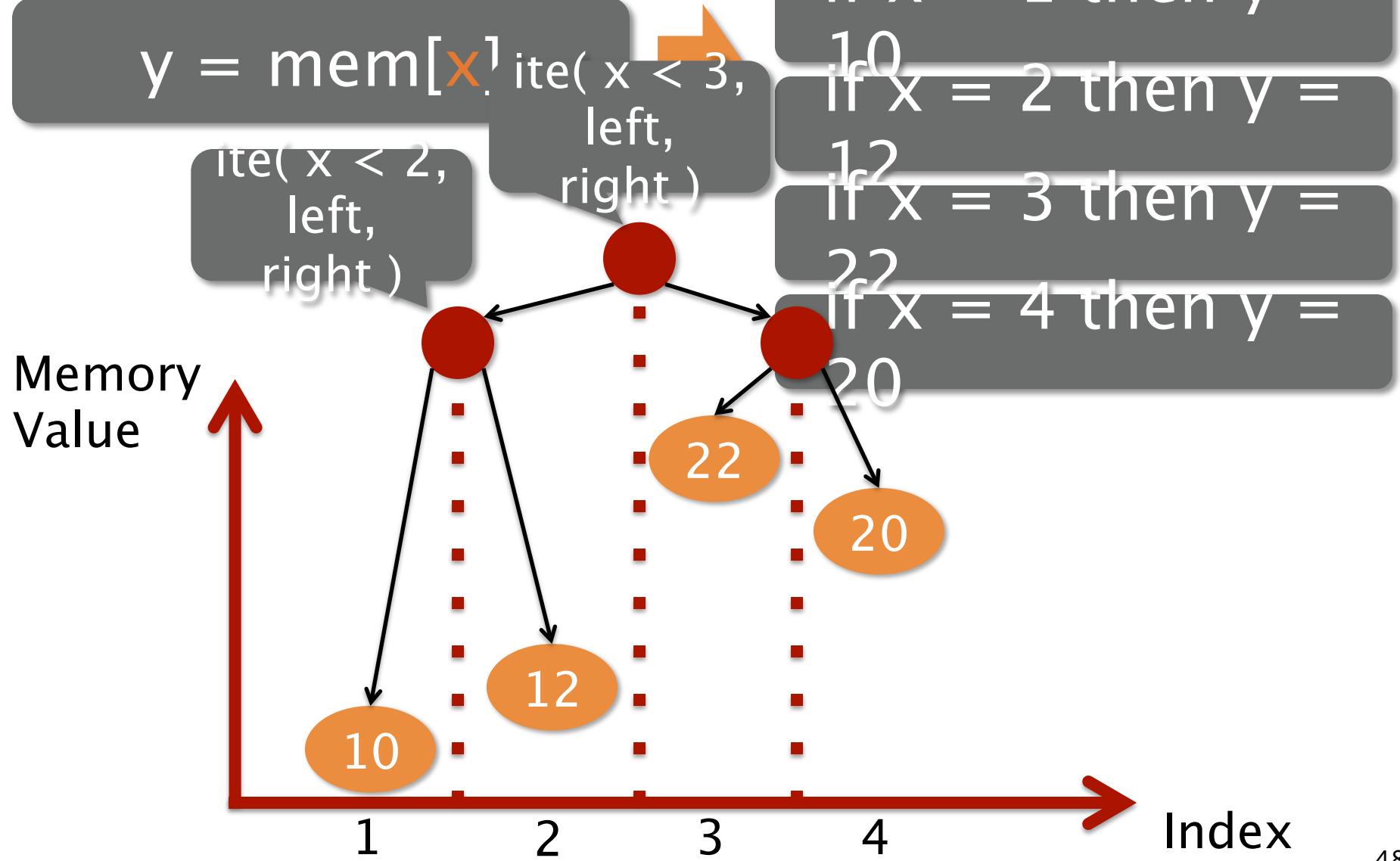
mem[x & 0xff]



Lowerbound = 0, Upperbound = 0xff

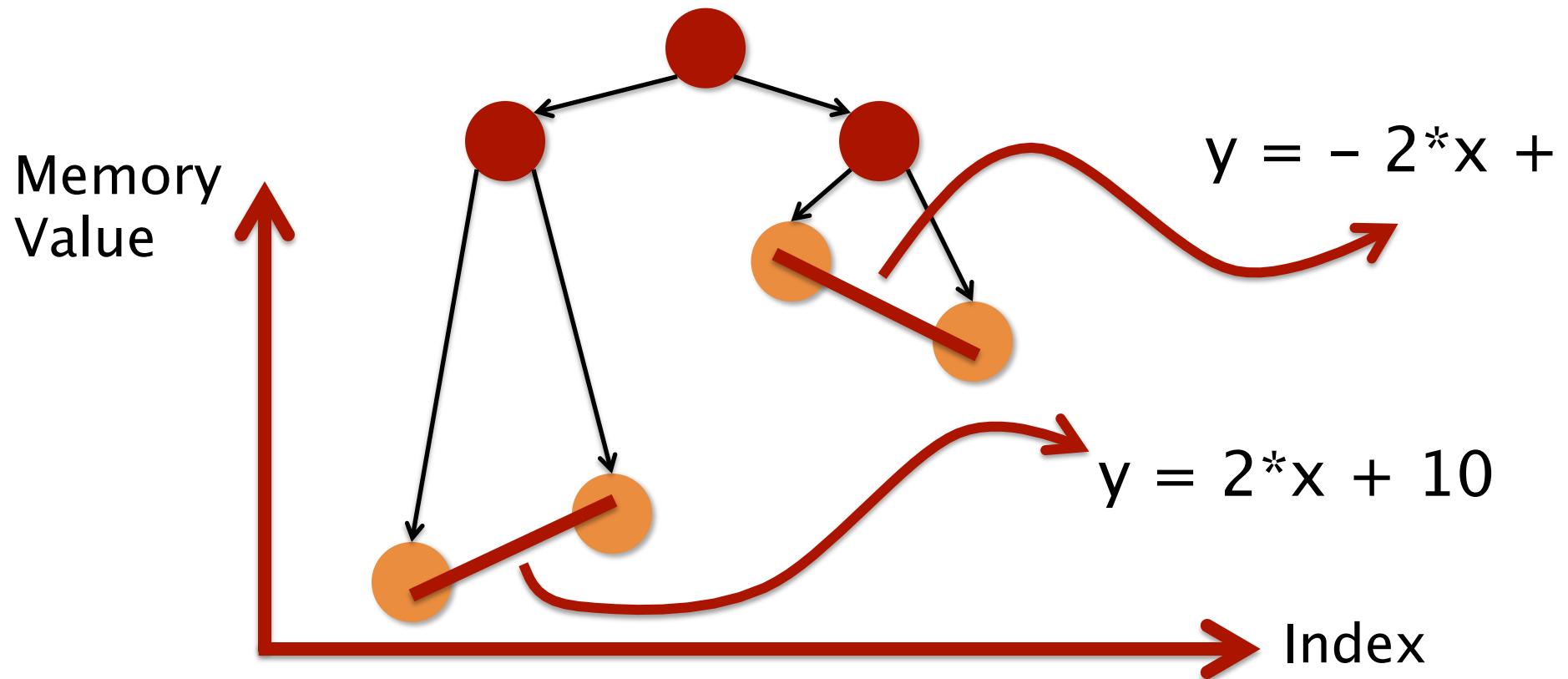
1. Value Set Analysis¹ provides initial bounds
 - Over-approximation
2. Query solver to refine bounds

Step 2 – Index Search Tree Construction



Index Search Tree Optimization (reads):

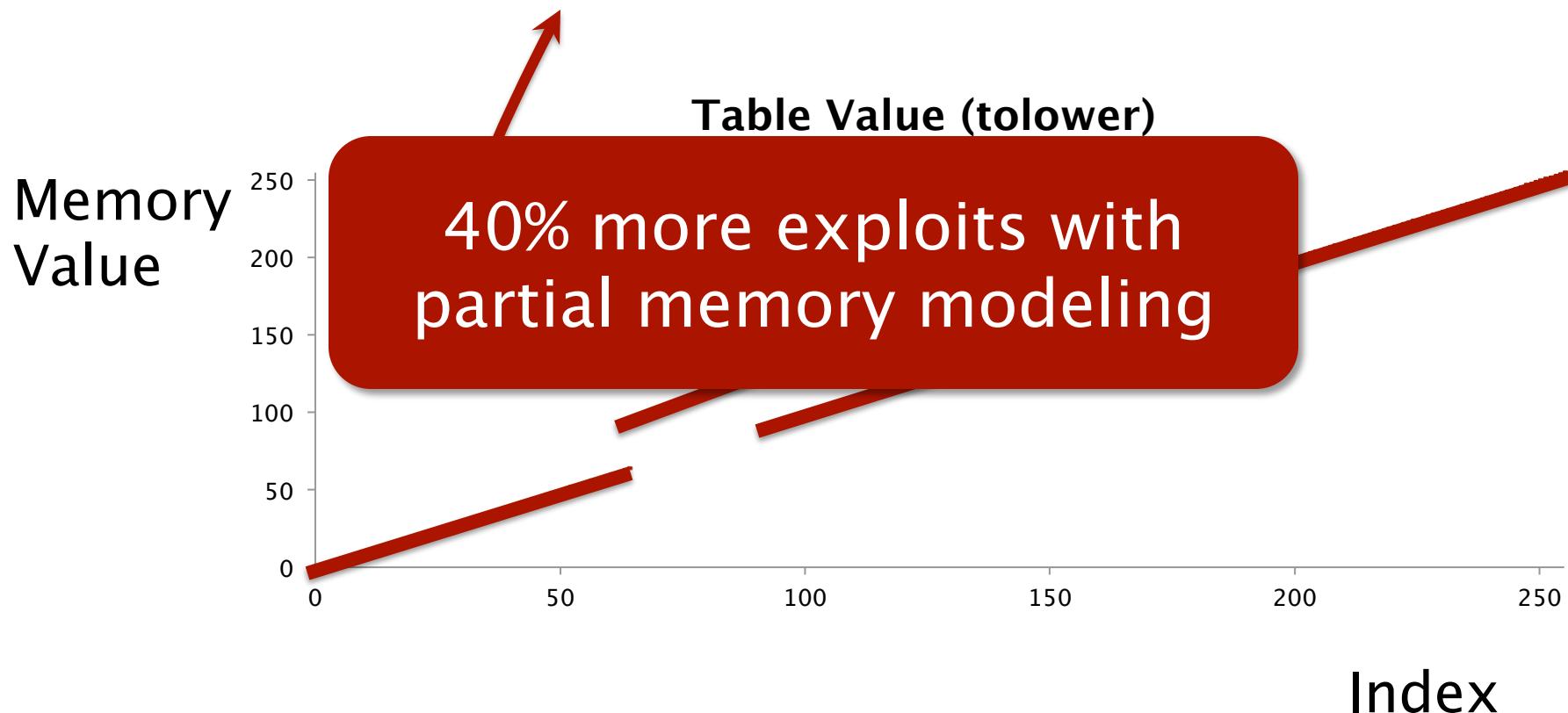
Piecewise Linear Reduction

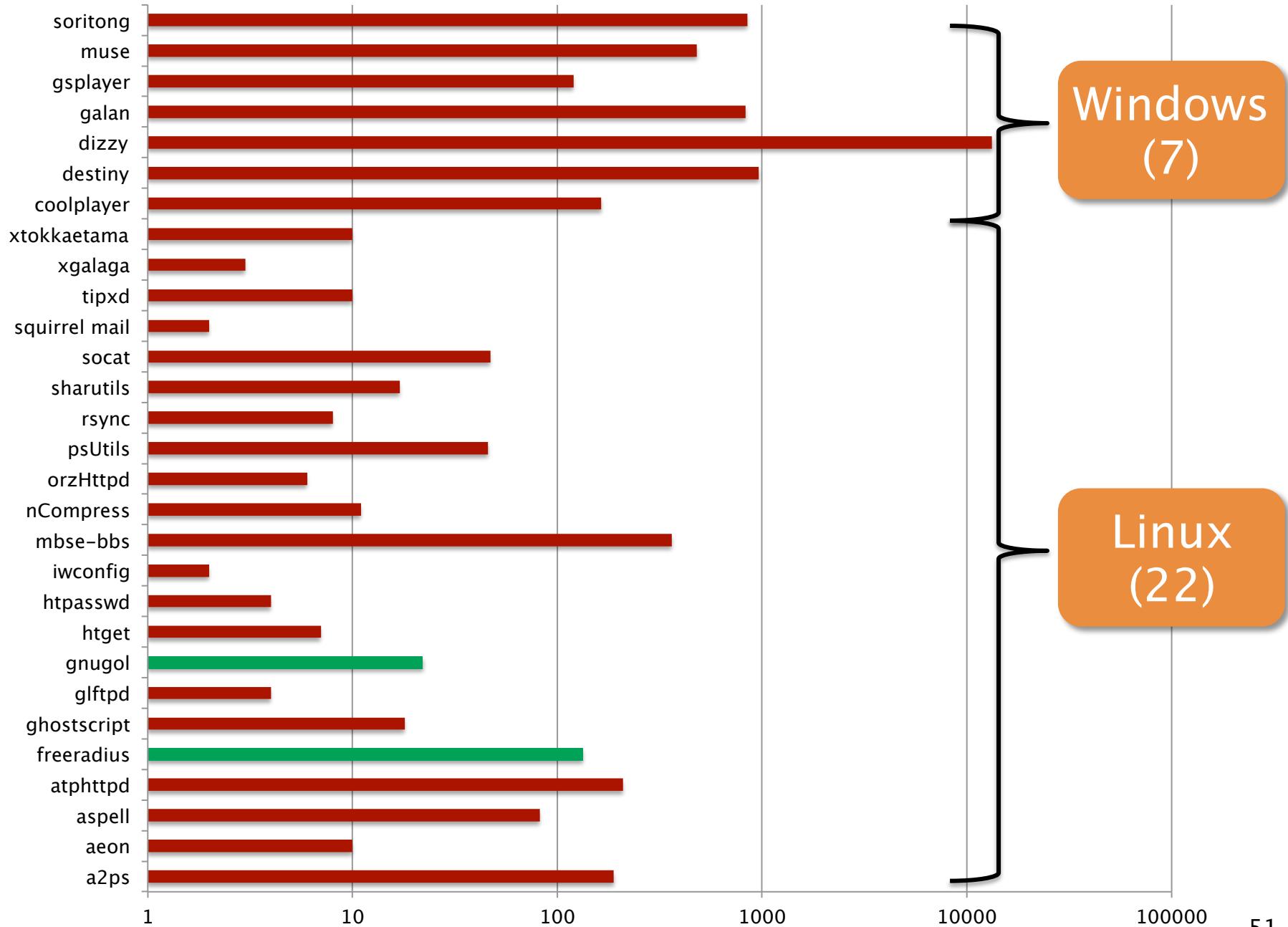


Index Search Tree Optimization (reads):

Piecewise Linear Reduction

`ite(n < 91, ite(n < 64, n, n+32), n)`





State Explosion: An Example in C

```
1. int counter = 0;  
2. for ( i = 0 ; i < 100 ; i ++ ) {  
3.   if (input[i] == 0x42) // 'B'  
4.     counter ++;  
5. }  
6. if (counter == 75) bug ();  
7. ...
```

- 100 consecutive branches
- 2^{100} feasible paths

State Explosion: An Example in C

```
1. int counter = 0;  
2. for ( i = 0 ; i < 100 ; i ++ ) {  
3.   if (input[i] == 0x42) // 'B'  
4.     counter ++;
```

Can we check *all* states in a reasonable amount of time?

- Time to check 2^{100} states:

DSE executing @ 1state/ns: $\sim 10^{14}$ years

Age of Universe < 10^{12} years

Yes, but *not* if we check
one state at a time

Static Symbolic Execution (SSE)¹

- SSE input:

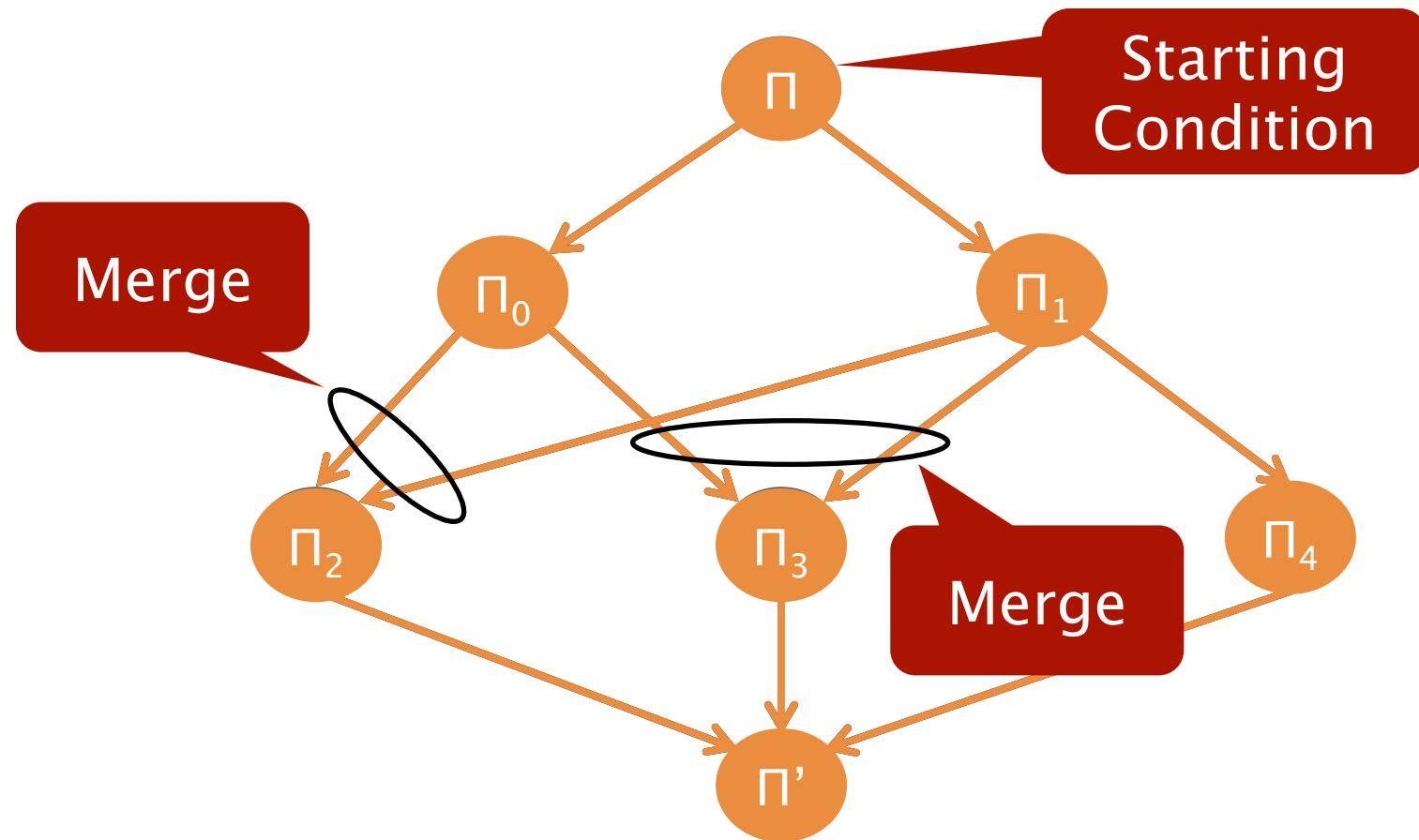
- Starting condition for the execution
- An **acyclic** control flow graph (CFG)

- SSE output:

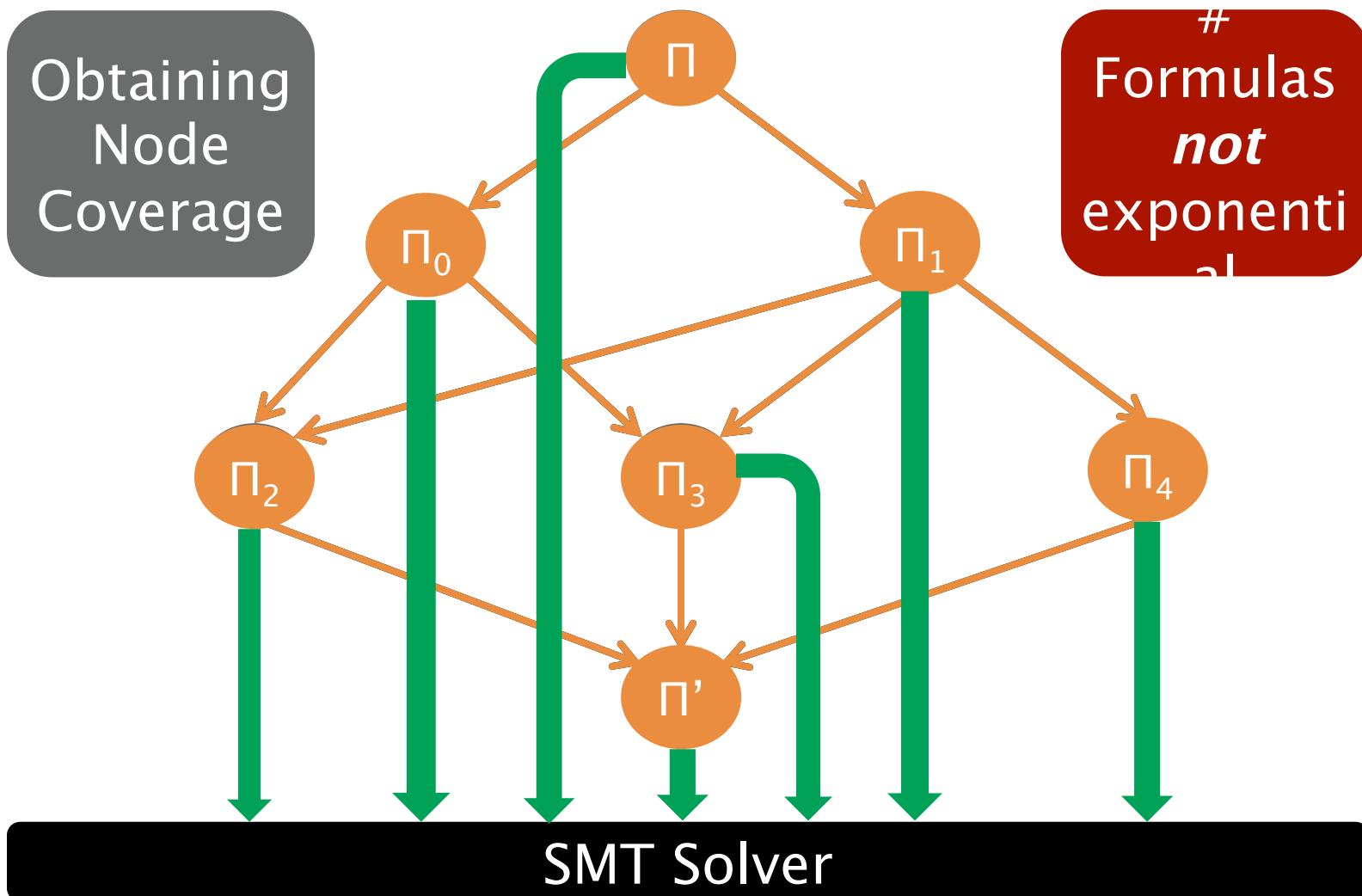
- One formula per CFG node
 - encompasses **all paths** reaching the node

[1] Variants by Koelbl et al. [IJPP'05], Xie et al. [POPL'05], Babic et al.

Static Symbolic Execution (SSE)



Static Symbolic Execution (SSE)



Static Symbolic Execution (SSE)

- SSE input:

- Starting condition for the execution
- An acyclic control flow graph (CFG)

What about features that
cannot be recovered
statically?

- SSE output:

- One formula per CFG node
 - encompasses all paths reaching the node

What about
programs with
loops?

Are formulas too
difficult to solve?

How expensive is formula solving?

- Solve time in DSE (25 million queries)
 - 99.9% solved in less than 1sec
 - 95% solved in less than 100ms
 - Mean solve time: 3.67ms
 - Variance: 0.34ms
 - SAGE¹ reports similar results (99% require less than 1sec)

[1] Bounimova et al, Billions and Billions of Constraints: Whitebox Fuzz Testing in Production [ICSE'13]

Quick Recap

DSE for Testing

✓ Dynamic execution

✓ Loops unrolled as the code executes

✓ Formula solving time acceptable

✗ Path explosion

SSE for Verification

✗ Missing dynamic features

✗ # Unrolls per loop unknown

✗ Formula solving worse than DSE

✓ No path explosion

Trade-off #3

DSE for ~~Testing~~ State explosion

- ✓ Formulas
- ✓ Dynamic features

SSE for Verification

- ✓ State explosion
- ✗ Formulas
- ✗ Dynamic Features

Veritesting [1]

- ✗ State explosion
- Formulas
- ✓ Dynamic features
- ✓ Bugs & testing

Segment the state space and
check *sets* of states
simultaneously

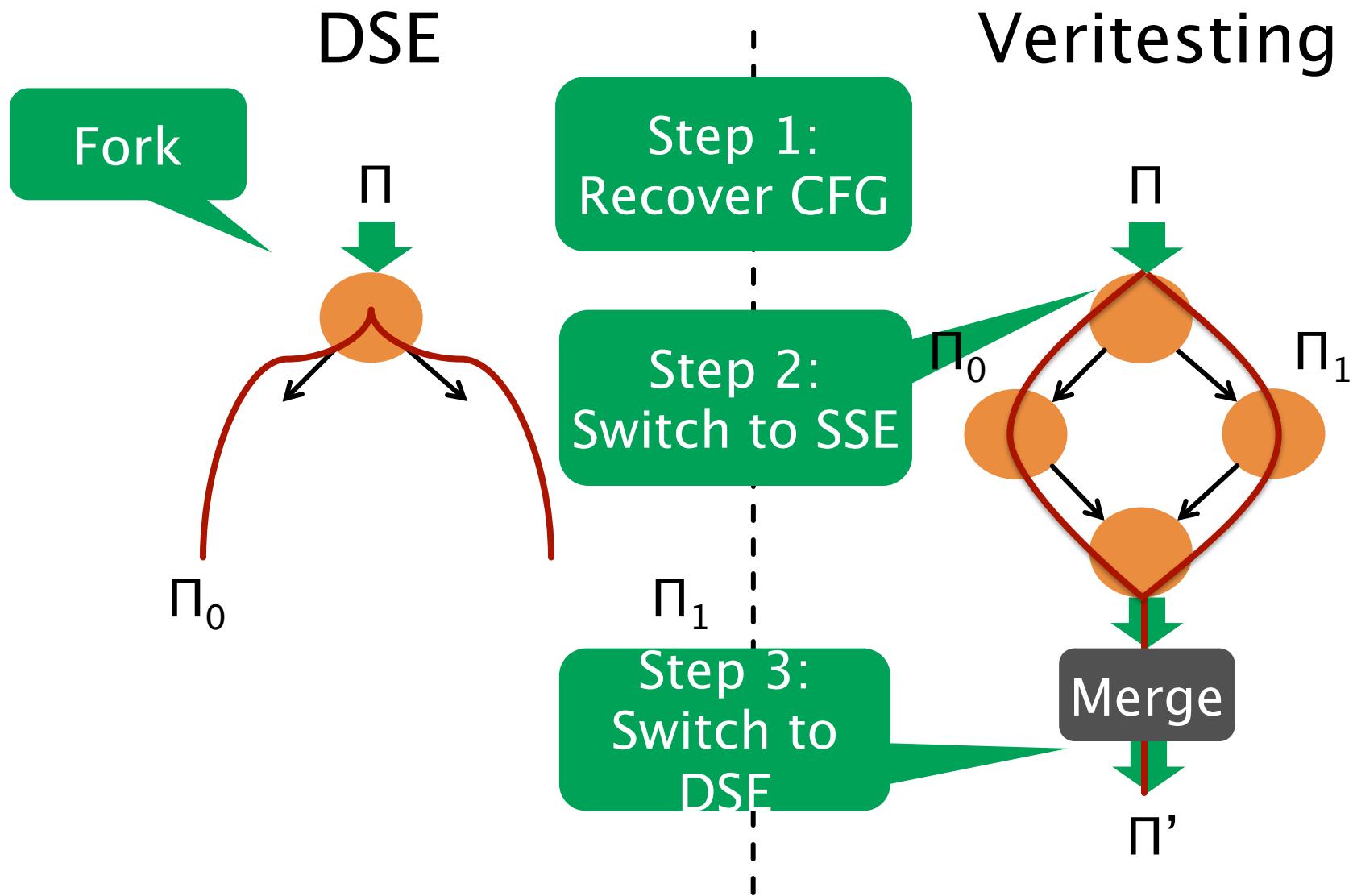
[1] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha and David Brumley, ***Enhancing Symbolic Execution with Veritesting***, In Proceedings of the International Conference on Software Engineering (ICSE'14), June 2014.

* ACM Distinguished Paper Award (to appear in CACM 2015 Research) 61

Core Idea: Alternate DSE + SSE

- Use DSE to:
 - Dynamically unroll loops
 - Have access to dynamic features
- Use SSE to:
 - Analyze multiple paths simultaneously

DSE vs Veritesting



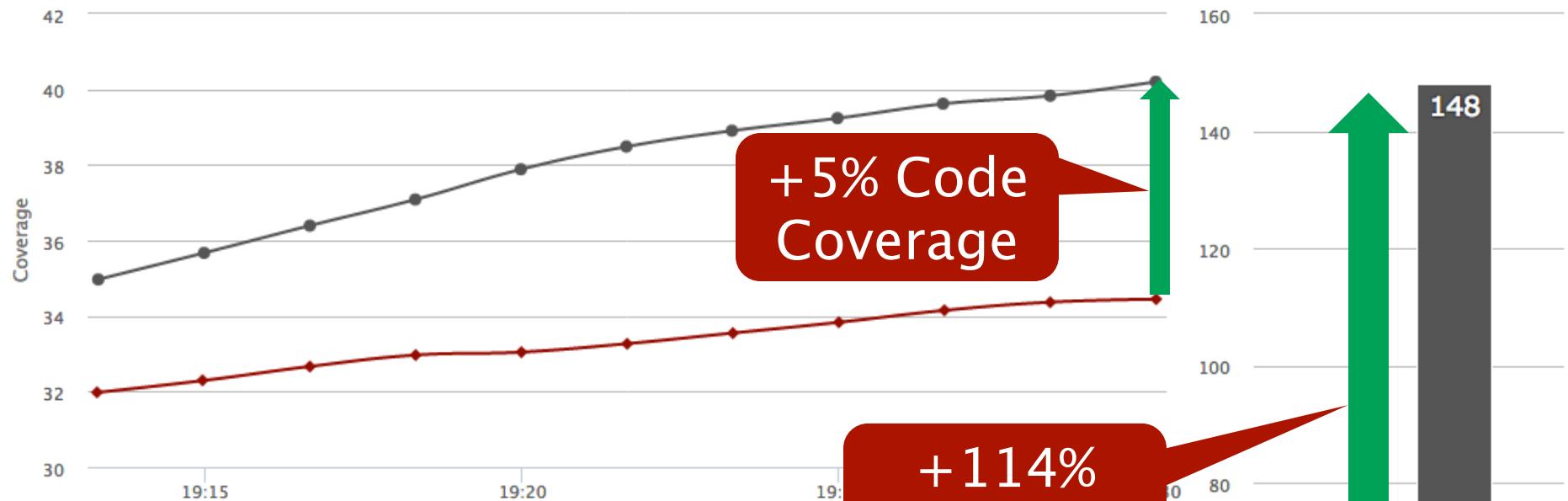
DSE vs Veritesting



Experiments on 1,023 Programs

- Source: Debian Squeeze (default install)
- All /bin, /usr/bin, /sbin ELF 32-bit binaries
- Time: 30 minutes each (DSE vs Veritesting)
- Measured:
 1. # of Bugs
 2. Node code coverage (reported by gcov)
 3. Test cases

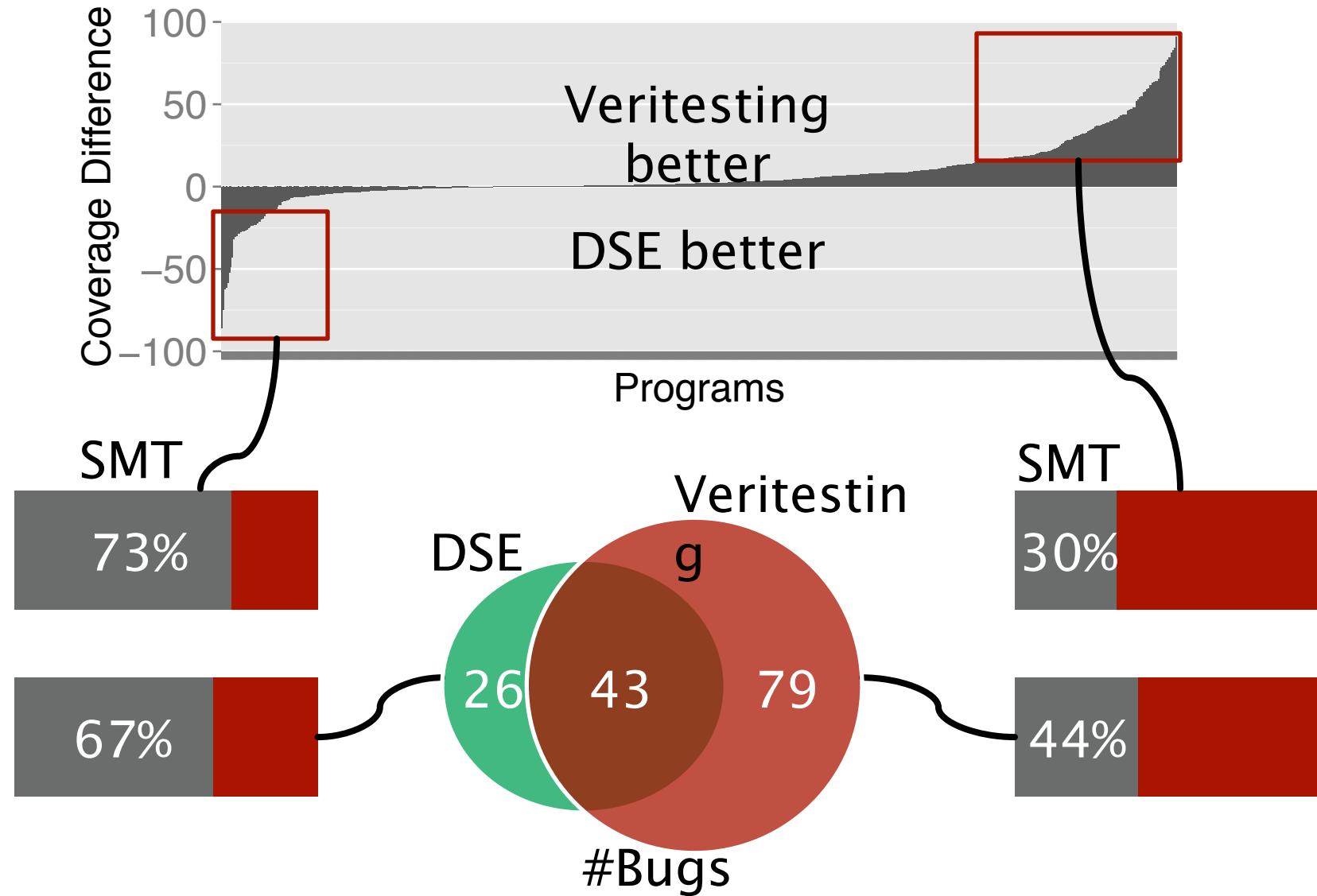
Code Coverage with Time



Test Cases with Time



Veritestng Profiles: a Trade-off



Statistics* from 7.7 Years CPU-time

- 37,391 programs / 16 billion SMT resolved

207 million test cases

2,606,506 crashes

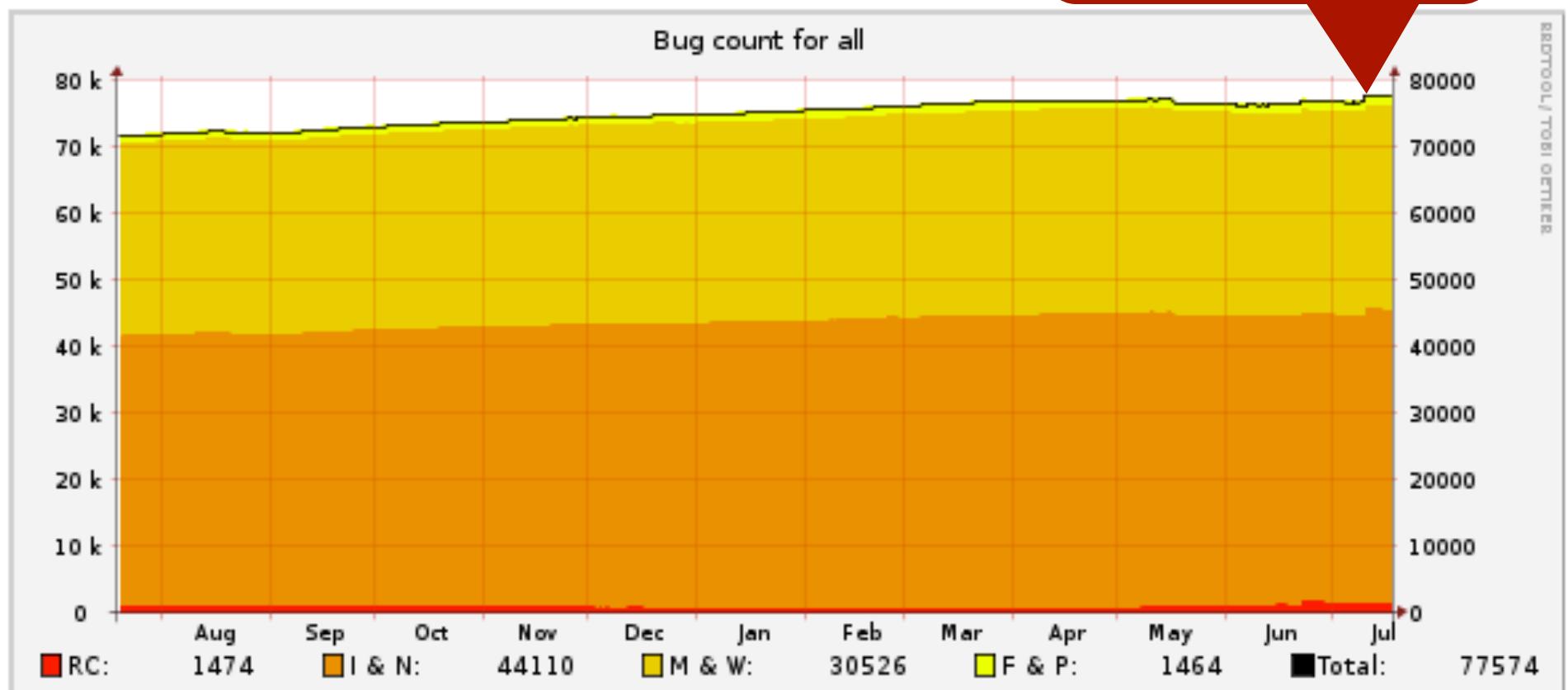
13,875 unique (stack hash)
bugs

152 control hijacks

[*] Statistics and data available at:
<http://forallsecure.com/debian>

Reporting 1.2K Crashes

MergePoint
bump



Feedback

“

Thanks for your extensive feedback, it's a pleasure to work with such

“

detailed material (and easy to pin the bug, BTW).

“

I have a lot of respect for the Mayhem tool now as a way to find corner cases in simple C parsers. I'm sure the team at CMU's project will find some very real bugs in

“

Debian.

“

I am sorry, but it is not a bug if jocamlrun segfaults when you feed

“

it garbage!

“

*No you ***did*** not! You might have found a bug in libc but it is not a bug in*

“

tart.

Bugs are getting fixed (slowly)

- [Outstanding bugs -- Normal bugs; Patch Available](#) (1 bug)
- [Outstanding bugs -- Normal bugs; Confirmed](#) (1 bug)
- [Outstanding bugs -- Normal bugs; Unclassified](#) (796 bugs)
- [Outstanding bugs -- Normal bugs; Will Not Fix](#) (1 bug)
- [Outstanding bugs -- Minor bugs; Confirmed](#) (1 bug)
- [Outstanding bugs -- Minor bugs; Unclassified](#) (7 bugs)
- [Forwarded bugs -- Normal bugs](#) (29 bugs)
- [Pending Upload bugs -- Normal bugs](#) (1 bug)
- [Resolved bugs -- Normal bugs](#) (28 bugs)

~300 bugs already fixed!

Acknowledgments



David Brumley



Sang Kil Cha



Alexandre Rebert



Edward J. Schwartz



Maverick Woo

- Brent Lim Tze Hao
- JongHyup Lee
- Ivan Jager
- Jonathan Foote
- David Warren
- Gustavo Grieco

Conclusion

- Automatically **finding** and **demonstrating exploitable** bugs is possible
- Exploiting tradeoffs such as state *pruning*, *reduction*, and *segmentation* can improve DSE as a testing/bug-finding tool

The future of binary program analysis should be exciting

Thank You!

Questions?