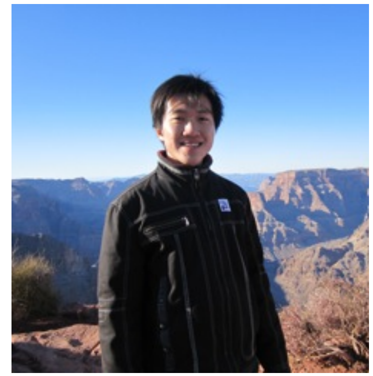
Symbolic Execution and Automated Exploit Generation

David Brumley
Carnegie Mellon University

Joint Work With

Thanassis Averginos

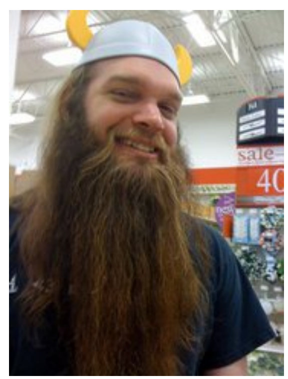
Sang Kil Cha



Brent Hao

Ed Schwartz

Pongsin Poosankam



Ivan Jager

Dawn Song

And Jiang Zheng, no picture available

Evil David

Find Bugs

↓

Exploit Systems

Automatic Exploit Generation

Given program, find bugs and demonstrate exploitability

My Research Agenda

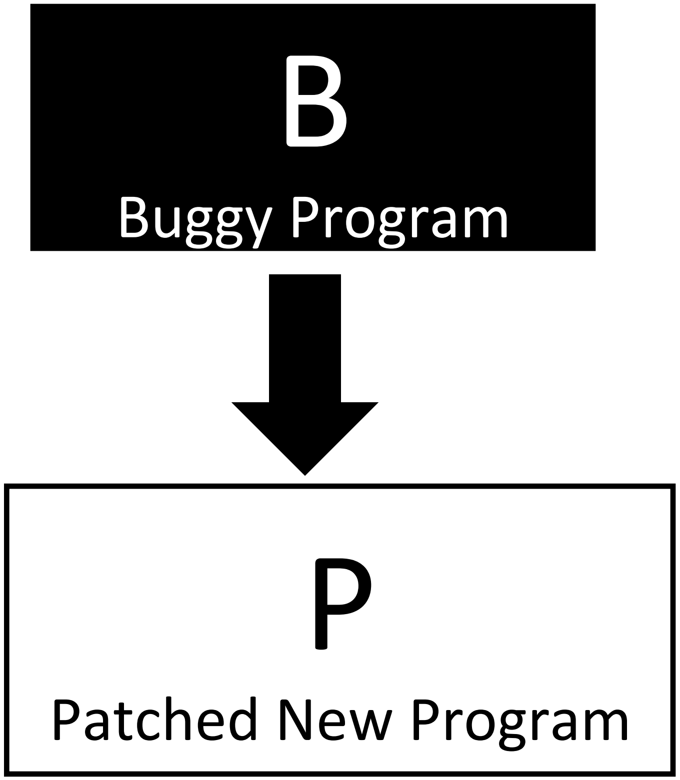
Tools and Systems for Real Code

Verification and Program Analysis

Formalize "Exploit"

Overview

- Automatic Patch-based Exploit Generation
  – IEEE Security and Privacy 2008
- Automatic Exploit Generation
  – NDSS 2011 and beyond
- Symbolic execution, taint analysis, and binary analysis lessons learned
  – 2003 - now

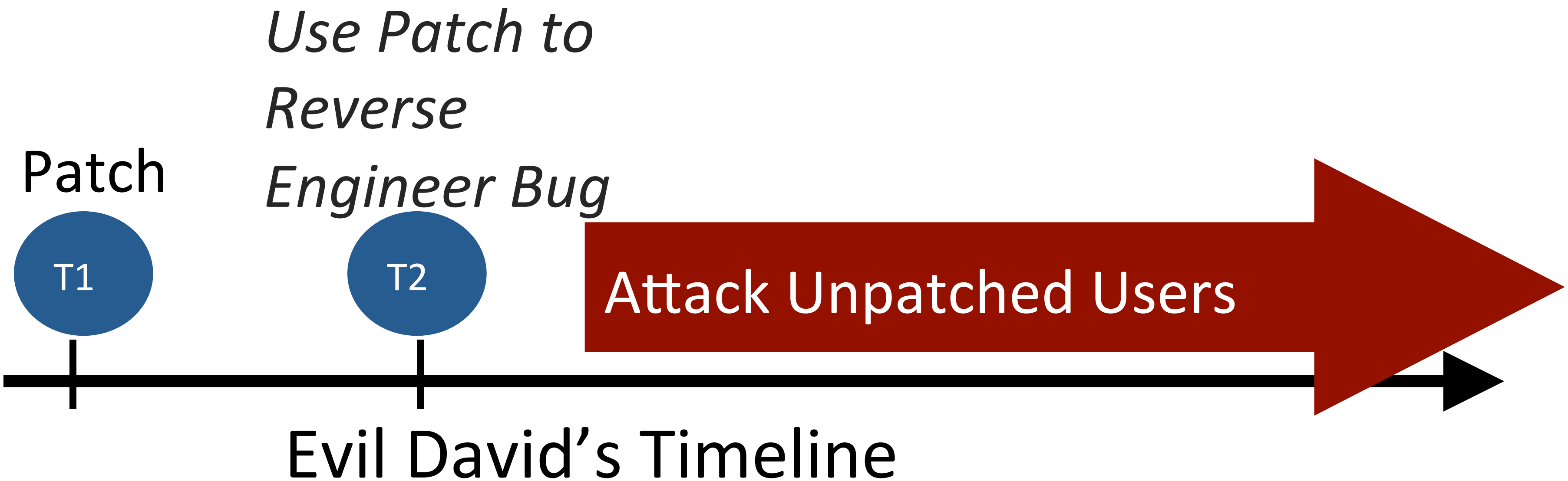


"Regularly Install Patches"
– US Dept. of Homeland Security

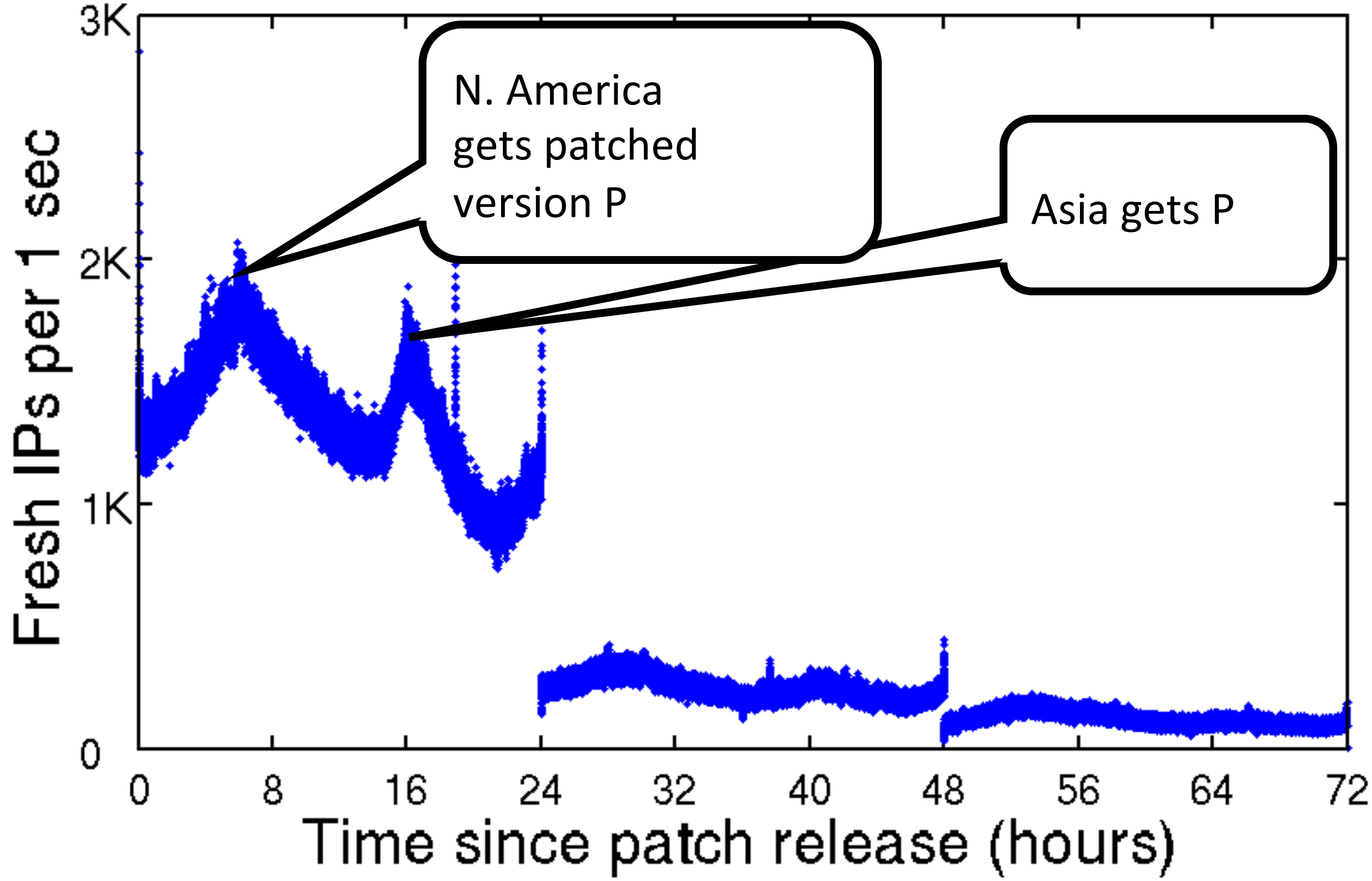
Patches can help attackers

Evil David

Delayed Patch Attack



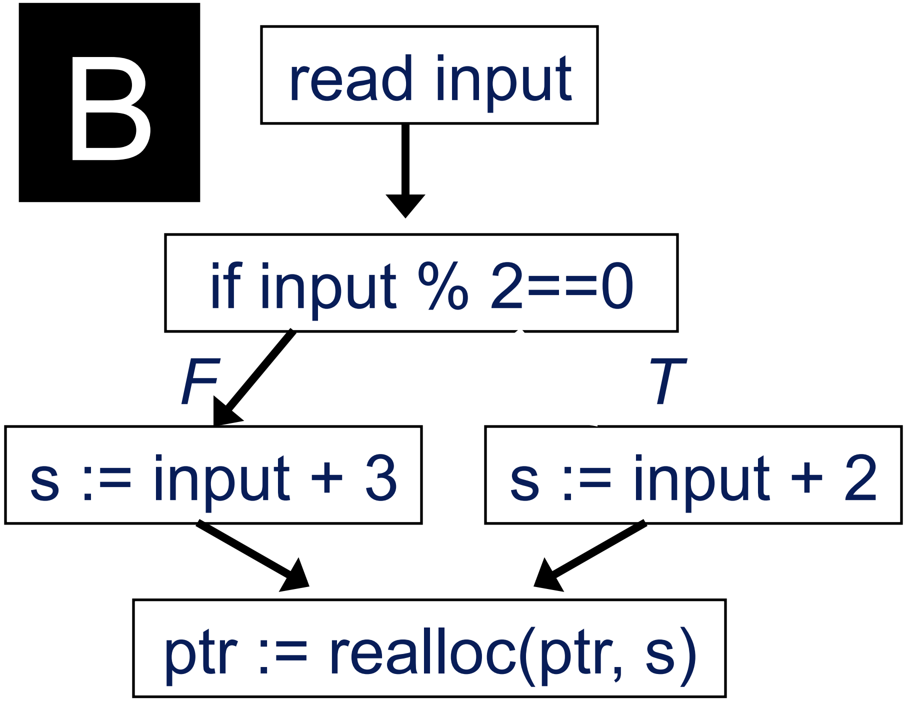
Patch Delay



[Gkantsidis et al 06]

Automatic Patch-Based Exploit Generation

APEG Example



- **All integers unsigned 32-bits**
- **All arithmetic mod 2^{32}**
- **B is binary code**

Understanding semantics is challenging:

- x86 is complex
- 100's of instructions

```
add a,b
shl x << a


goto L if carry
```

all control flow determined by flags

```
a = a+b
parity flag = …
carry flag = …
auxiliary carry flag = …
zero flag = …
signed flag = …
overflow flag = …
x = x << a
set carry flag if a <> 0


… Jump if carry set …
```

BAP: Binary Analysis Platform
Faithful Binary Code Analysis

- Faithful

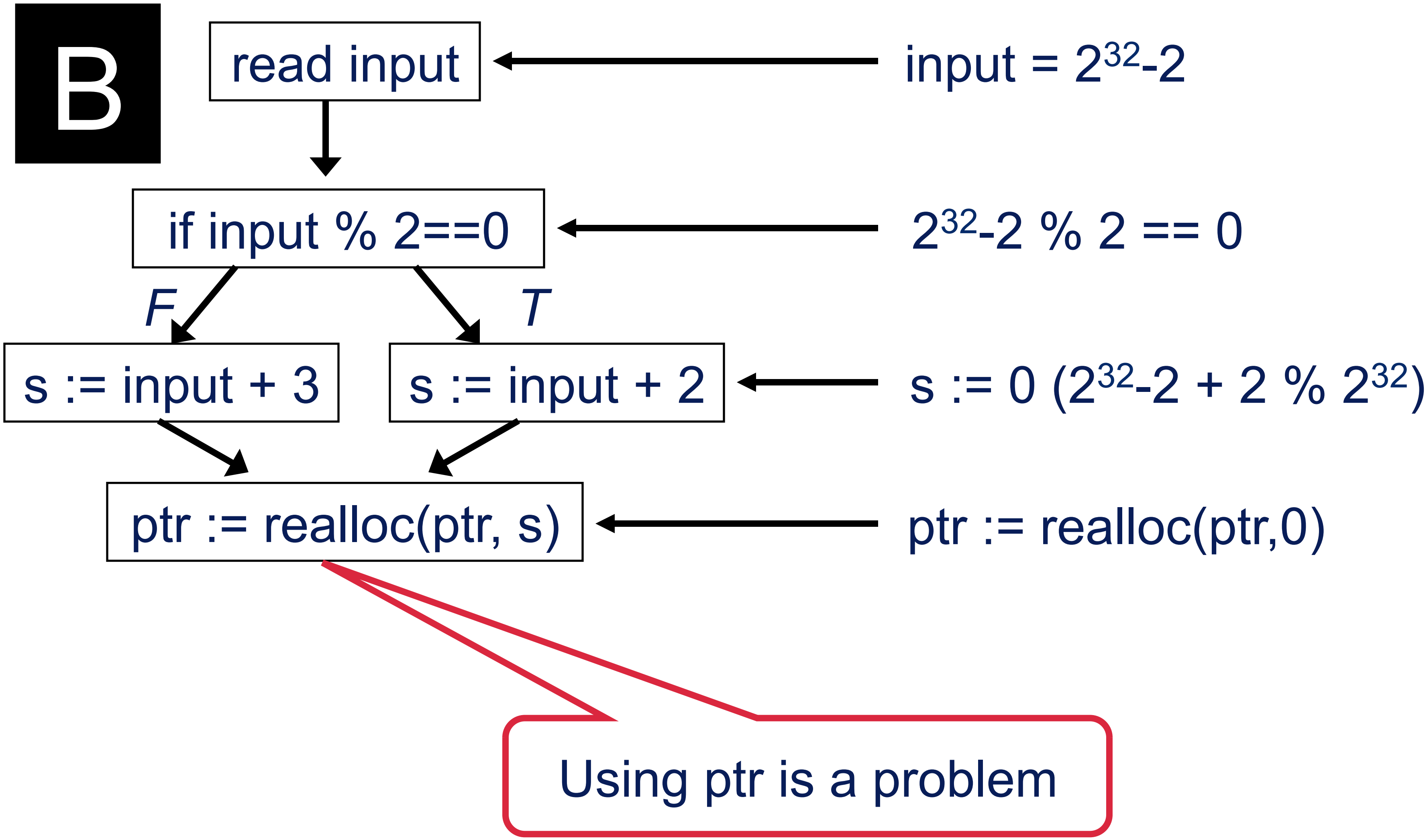
  Accurately model low-level semantics

- Simplified

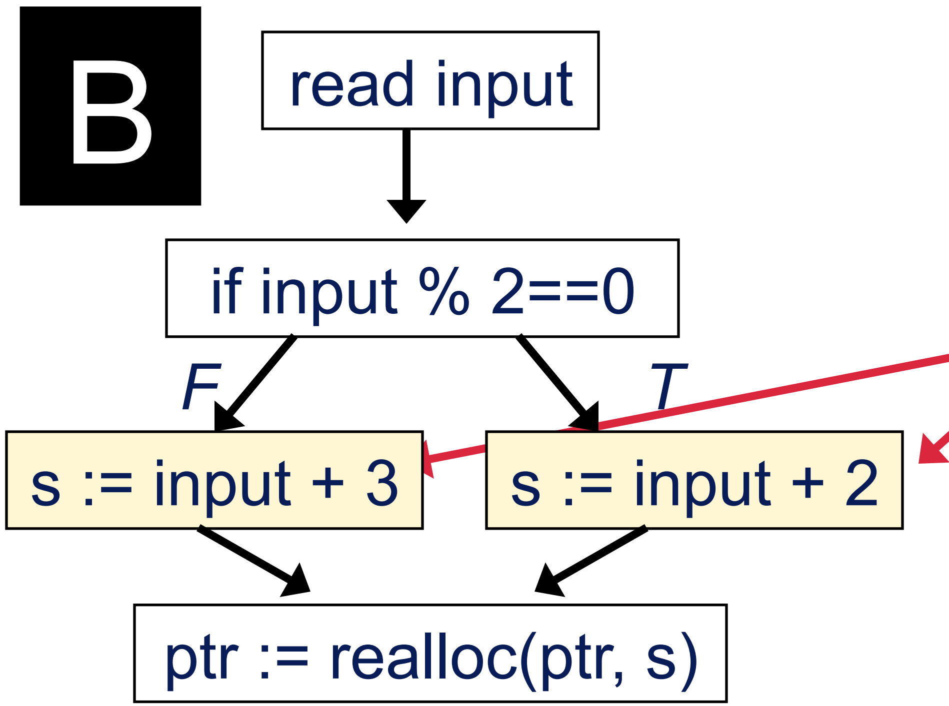
  Fewer cases

BAP Intermediate Language

```
  lval := exp
| goto exp
| if exp then goto exp1 else exp2
| return exp
| call exp
| assert exp
| special exp
| unknown (effects)
```

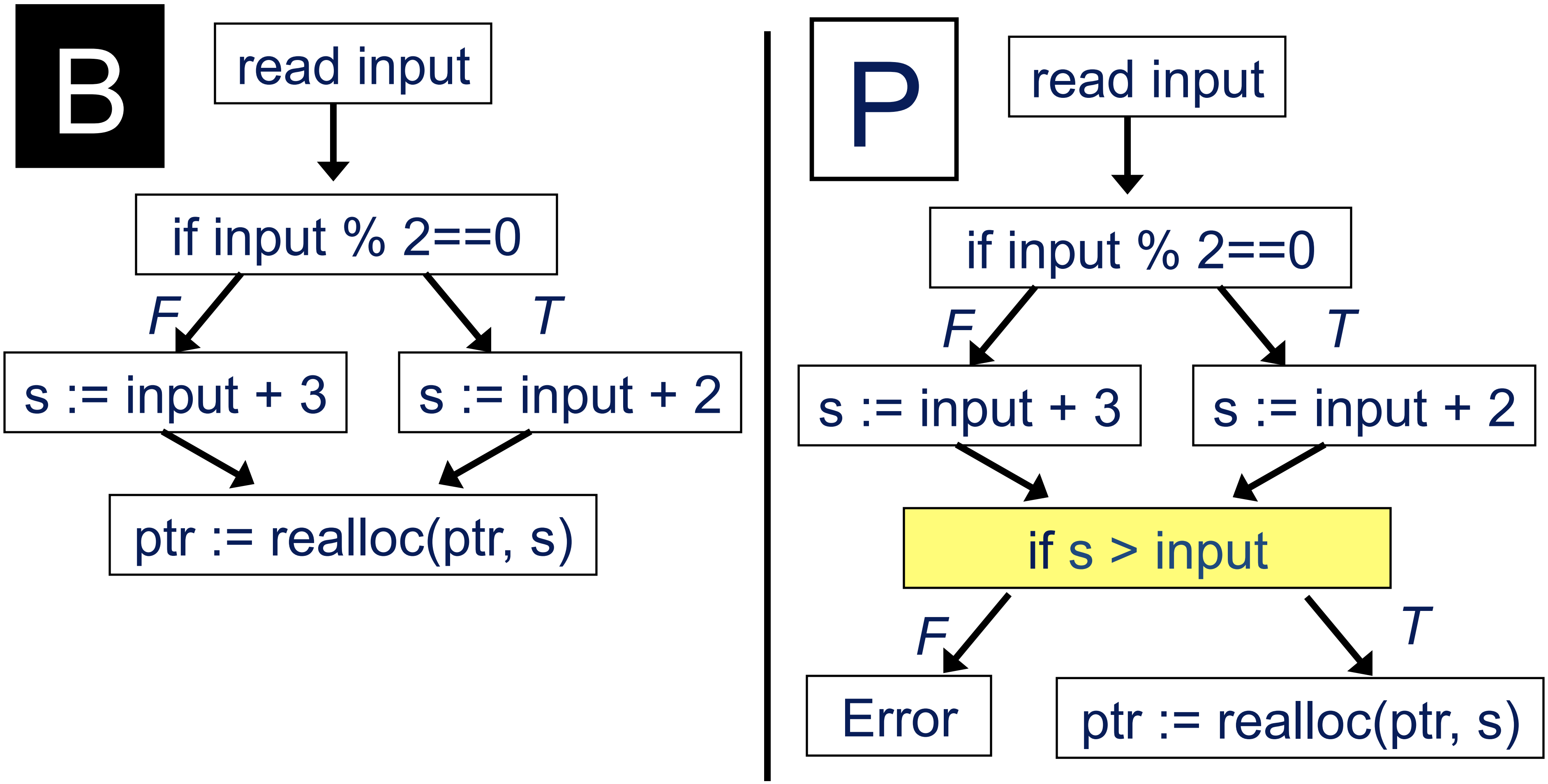


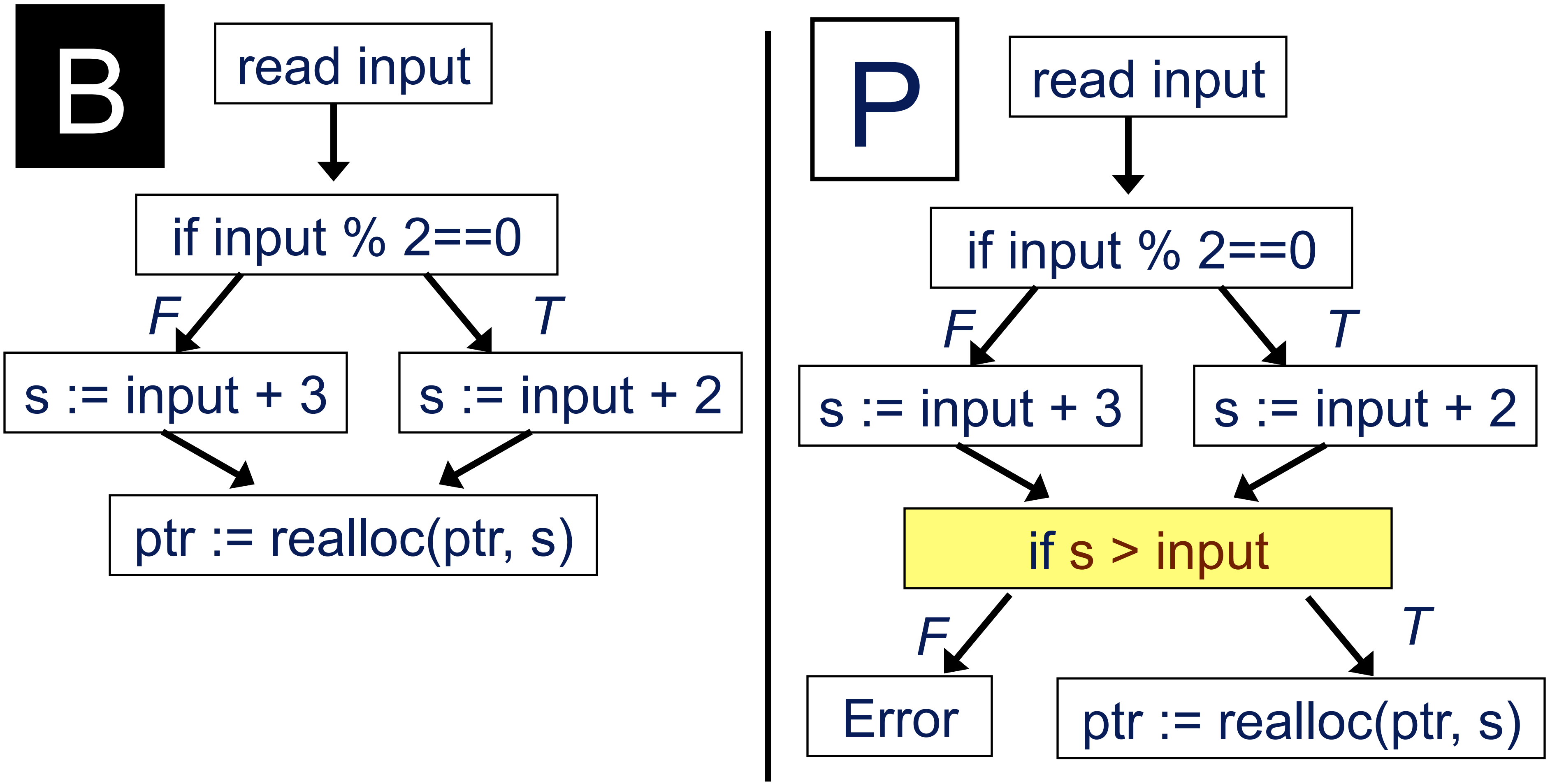
B
read input
if input % 2==0
F
T
s := input + 3
s := input + 2
ptr := realloc(ptr, s)
input = 2^{32}-2
2^{32}-2 % 2 == 0
s := 0 (2^{32}-2 + 2 % 2^{32})
ptr := realloc(ptr,0)
Using ptr is a problem



Wanted:
s > input

Integer Overflow when:
¬(s > input)


B
read input
if input % 2==0
F
T
s := input + 3
s := input + 2
ptr := realloc(ptr, s)
P
read input
if input % 2==0
F
T
s := input + 3
s := input + 2
if s > input
F
T
Error
ptr := realloc(ptr, s)

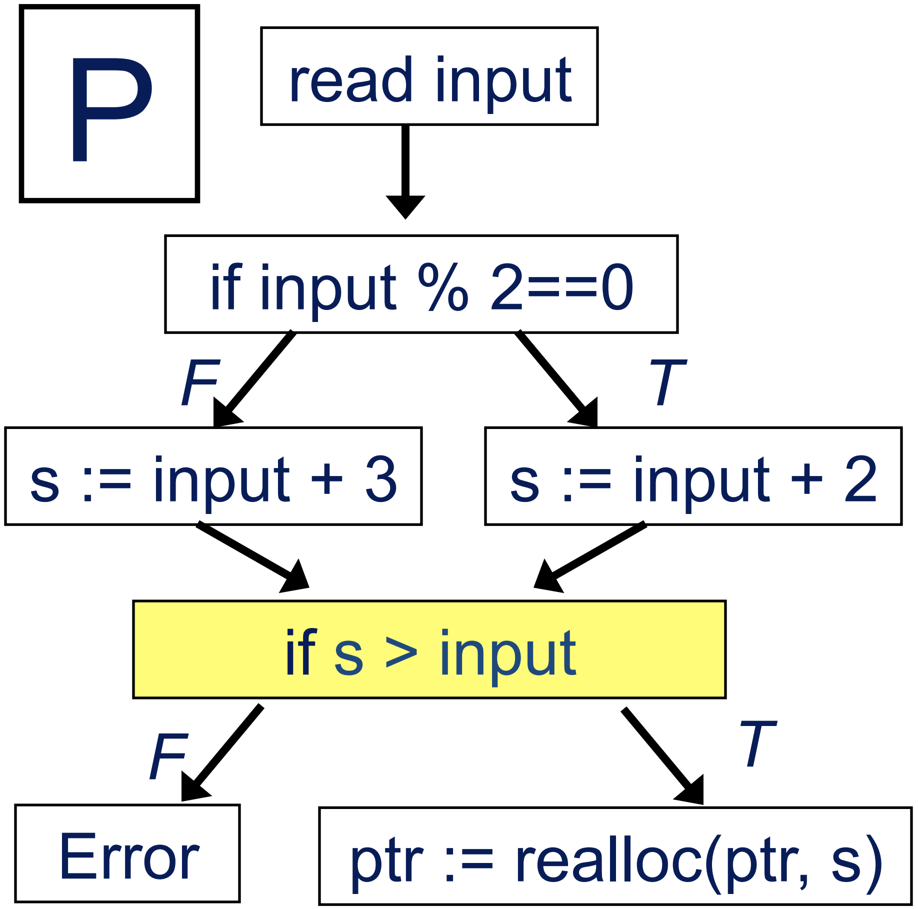


Exploits for B are inputs that fail new safety condition check in P

(s > input) = false

APEG

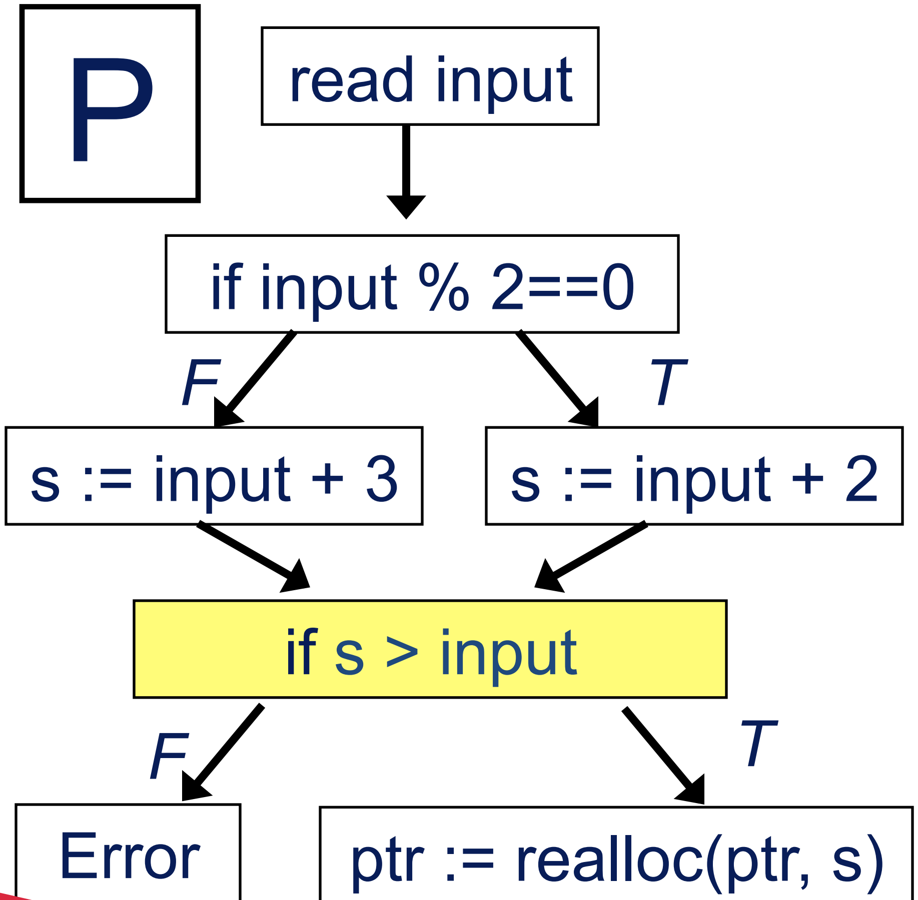
1. Diff B and P to identify location of new safety check
2. Create input that fails safety condition in P using Vine
3. Verify input is exploit on original buggy program B



1 & 3 performed using off the shelf tools

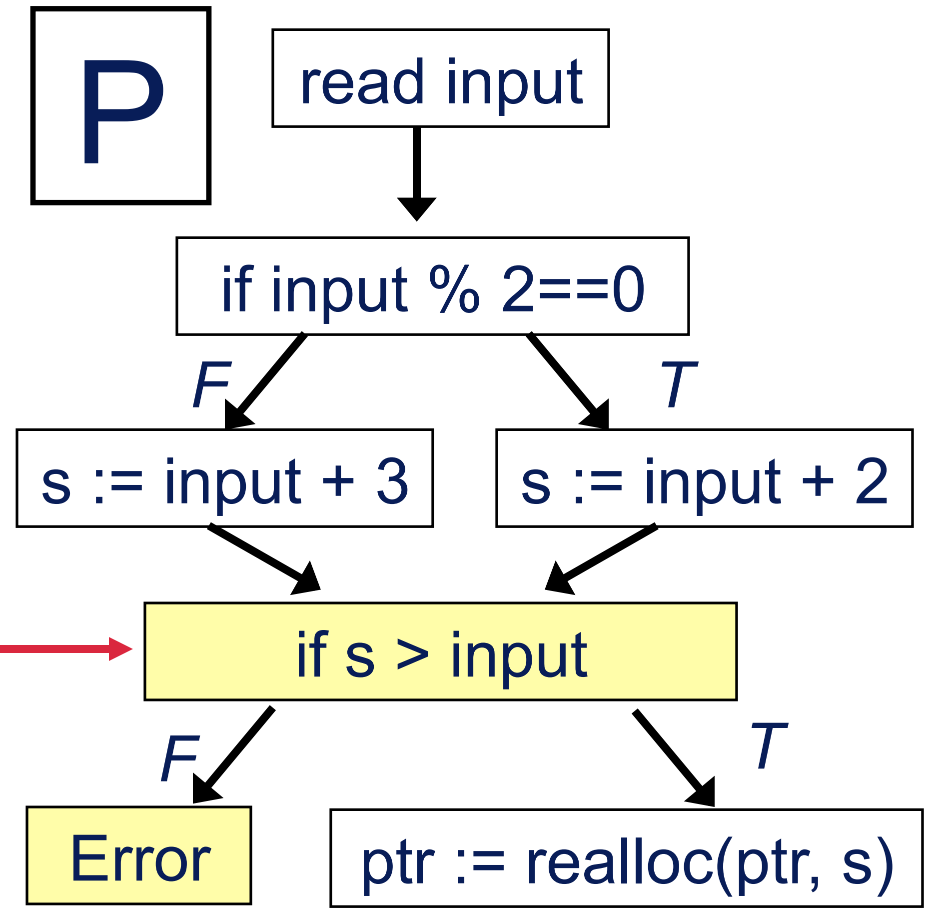
APEG

1. Diff B and P to identify location of new safety check
2. Create input that fails new safety check in P
3. Verify input is exploit on original buggy program B

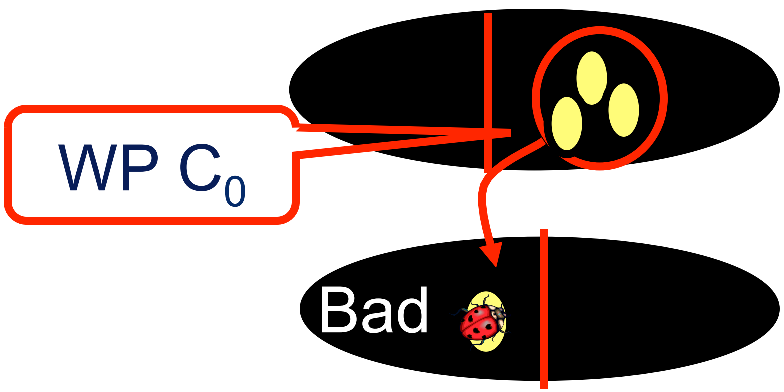


Weakest Precondition: Backwards computation of condition to fail check

WP:
Derive condition at step i-1 to execute line i



$C_0 = \neg(s > input)$



$C_3 =$

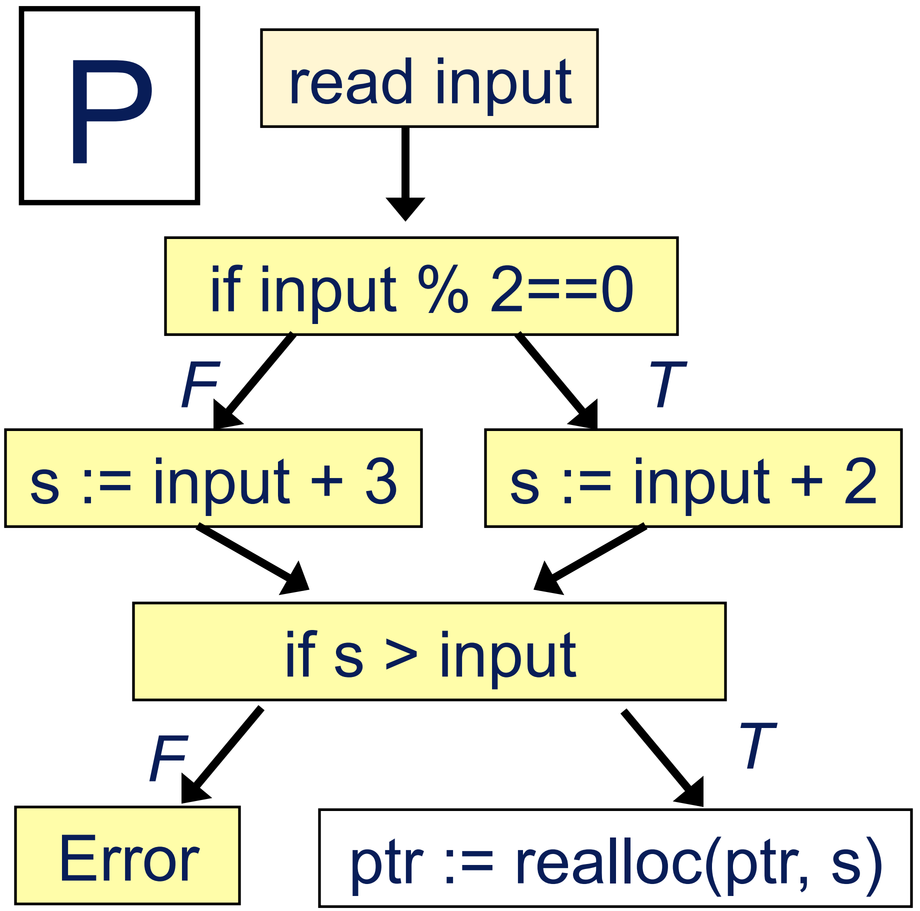
$(input \% 2 == 0) \rightarrow$

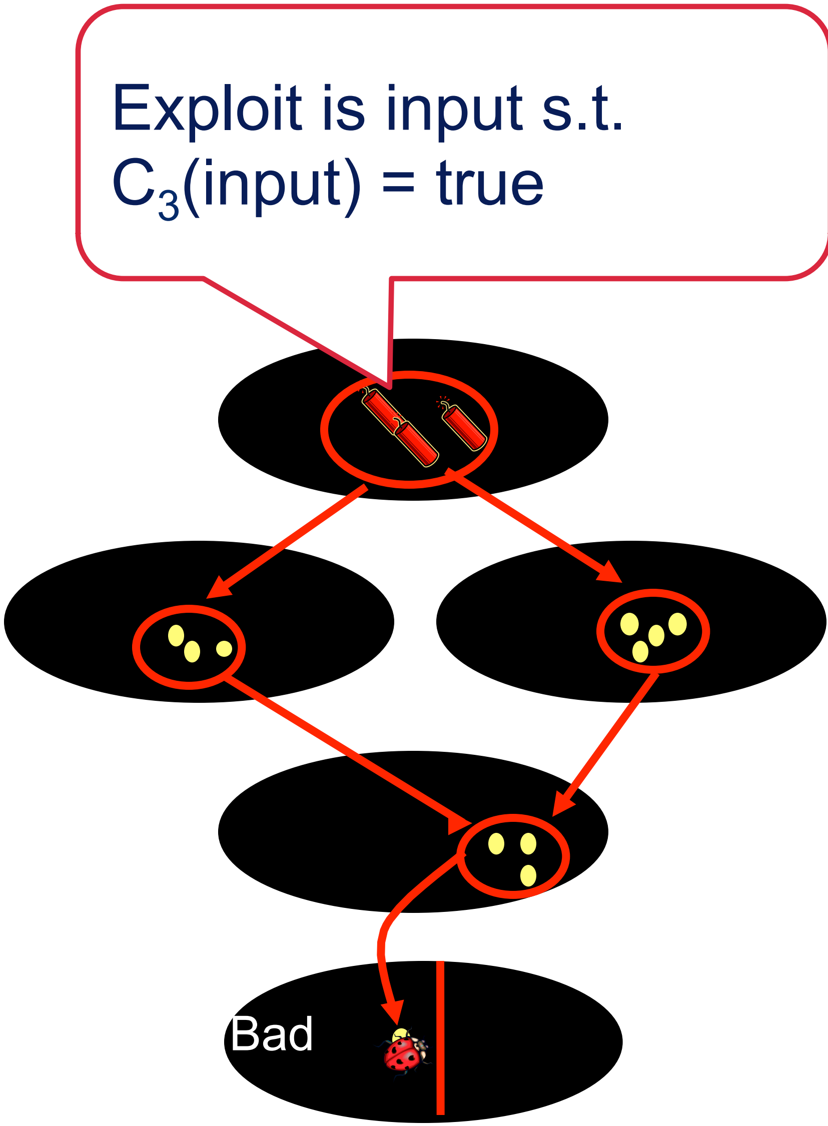
$\neg(input + 2 \% 2^{32} > input)$

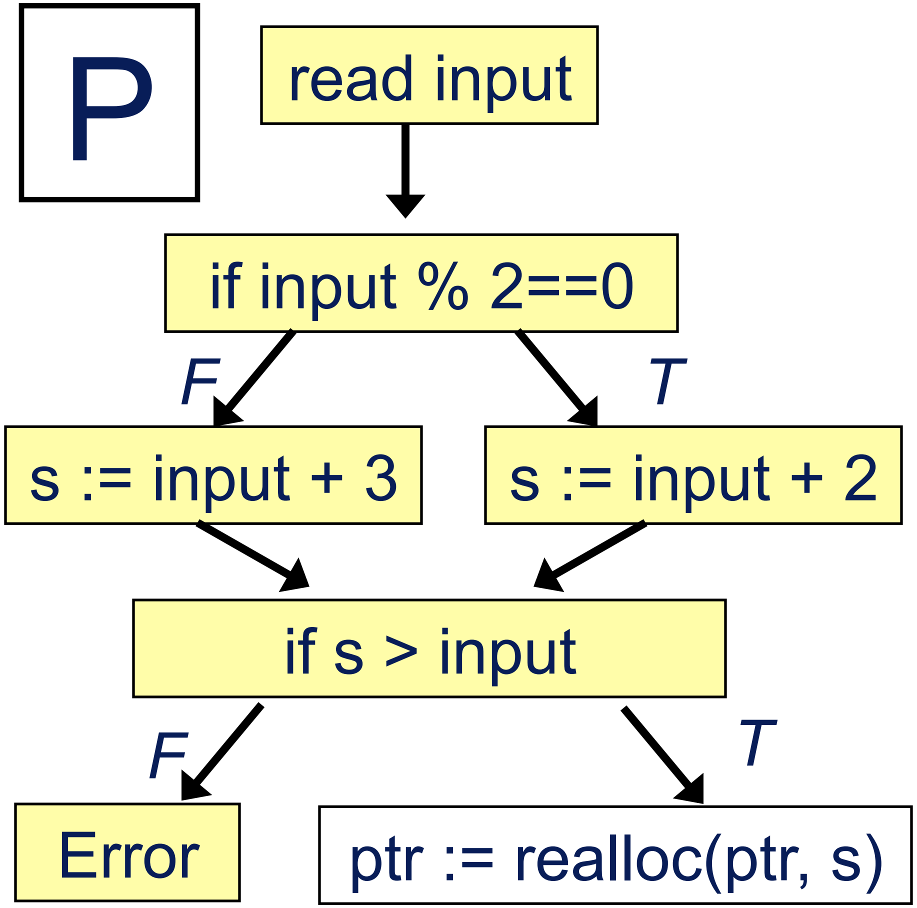
&&

$\neg(input \% 2 == 0) \rightarrow$

$\neg (input + 3 \% 2^{32} > input)$

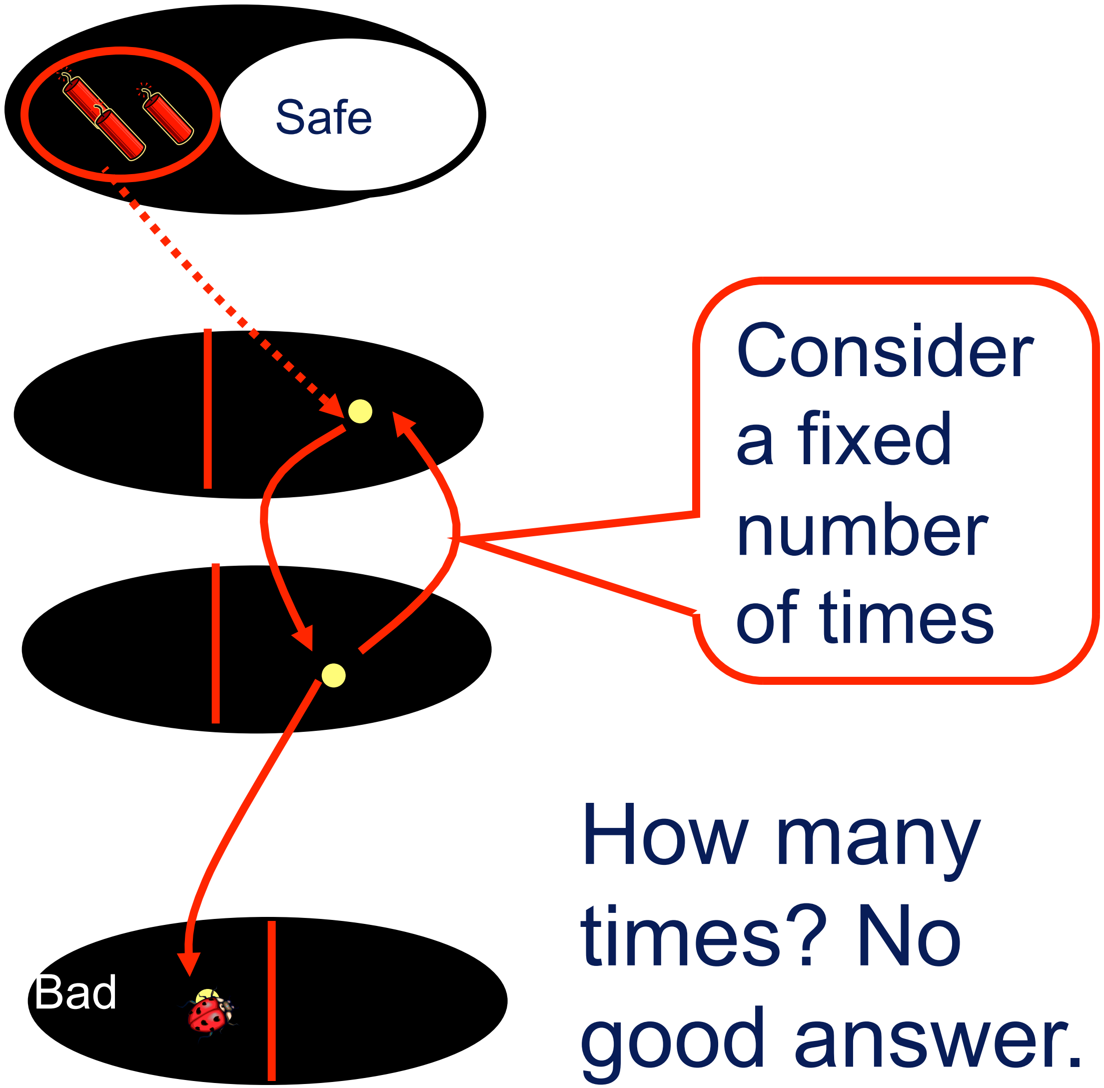






Use STP or other solver to find exploit

Problem 2: Loops



How many times? No good answer.

Problem 1:

1. WP not suitable for programs with gotos (unstructured programs)
2. Final C is exponential in size
   - Due to substitution for assignment
   - Duplication of condition in branches

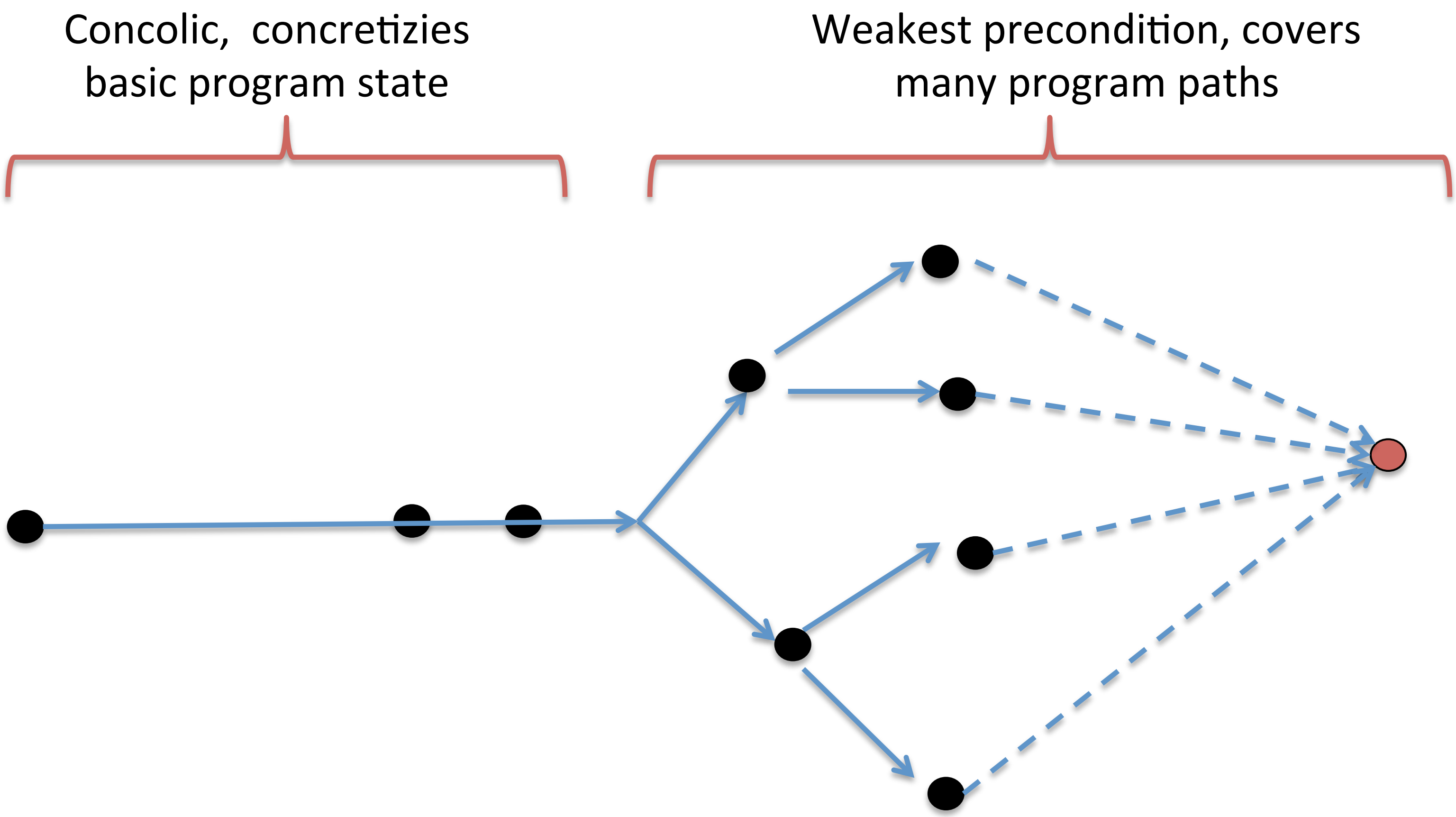
Substitution rule

Condition Rule

Developed variant of Flanagan and Saxe appropriate for unstructured code with $O(n^2)$ VC guarantee where n is size of program
- Leino also has nice work on this.

Problem: State Space Still too Huge

Our solution:
Mixed dynamic + static approach

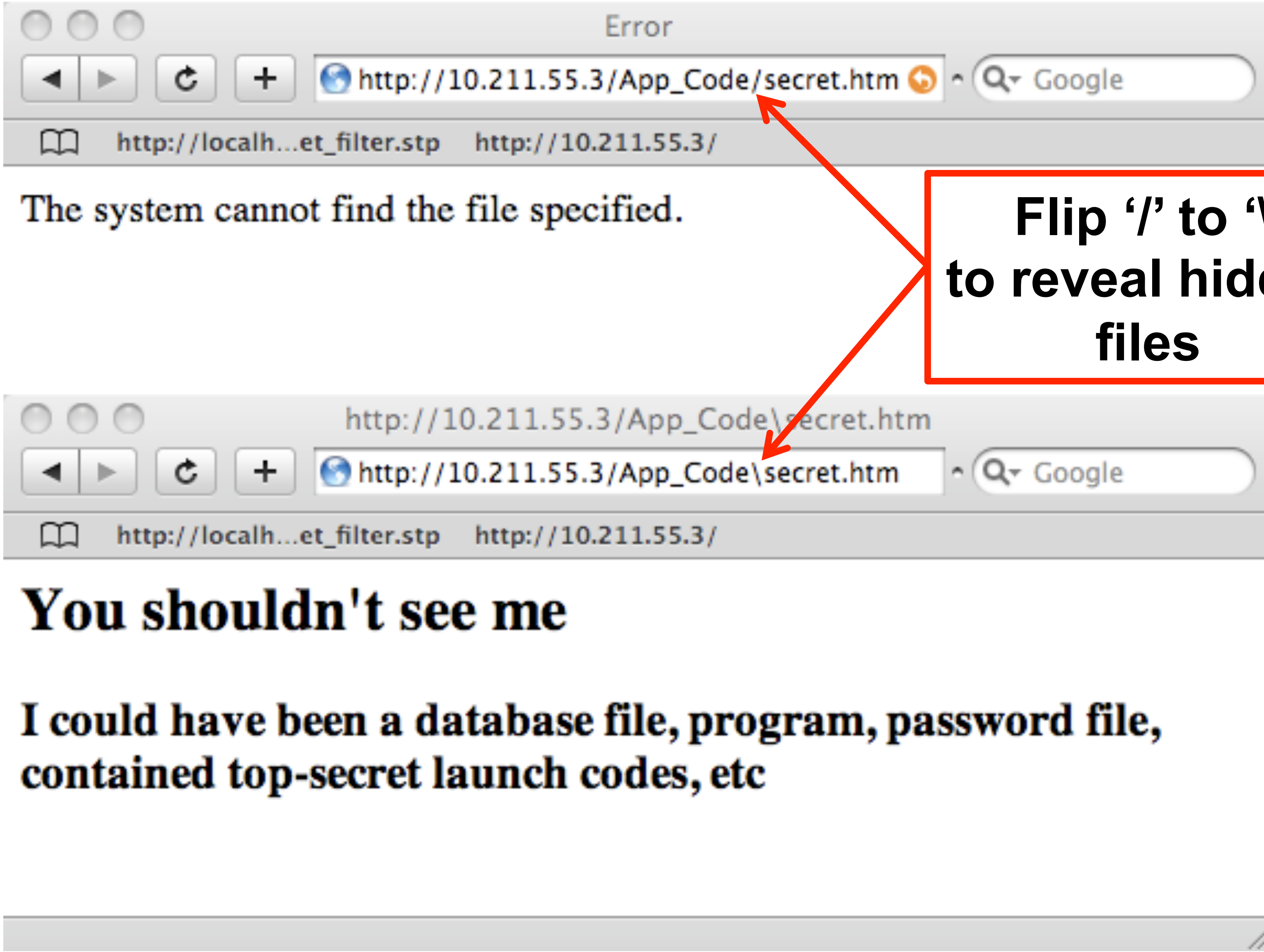

Concolic, concretizies
basic program state
Weakest precondition, covers
many program paths

APEG Results

ASPNet_Filter	Information Disclosure	29 sec
GDI	Hijack Control Possible	135 sec
PNG	Hijack Control Possible	131 sec
IE COMCTL32 (B)	Hijack Control Possible	456 sec
IGMP	Denial of Service	186 sec

- No public exploit for 3 out of 5
- Exploit unique for other 2
- STP optimizations implemented by Vijay reduced solve time by 1/2

Demo



Flip '/' to '\' to reveal hidden files

New Research Problem:
Prevent Patches From Helping Attackers

Research Ideas:

- Code Analysis: Obfuscate patches
  - Prevents diffing in our approach, no changes to current update schemes
  - Con: May slow down program, may be insufficient

- Crypto: Encrypt patch initially, broadcast decryption key
  - Fair: Everyone applies patch simultaneously
  - Con: Which patches to encrypt? Requires changes to current update schemes, offline hosts?

- Others

APEG Lessons

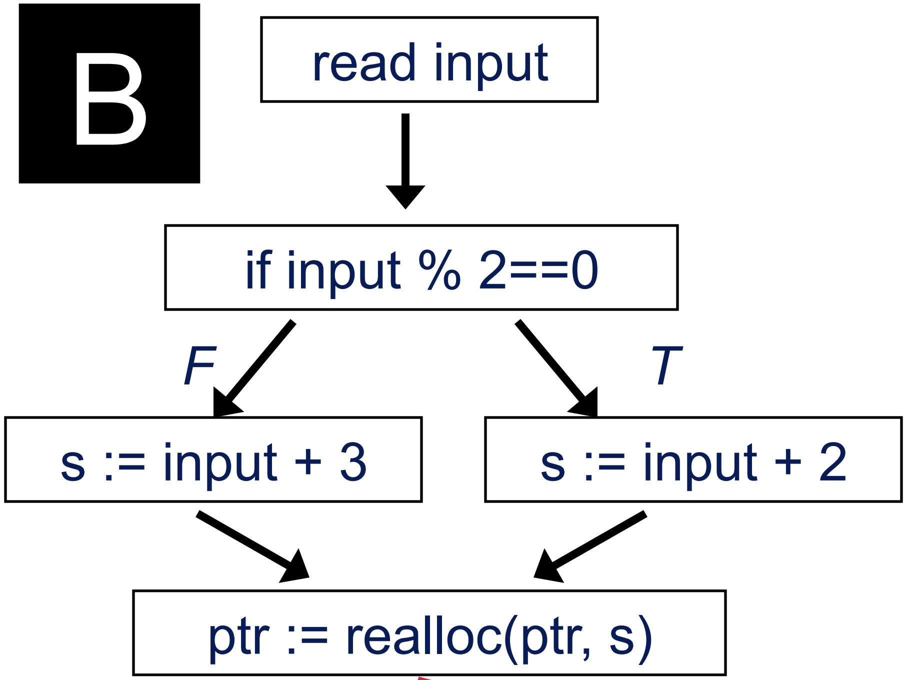
Pro

- Work with your SMT. STP optimizations cut cost of APEG by ½
- WP creates relatively small VC
- WP is goal driven from where we know there is a (potential) problem

Con

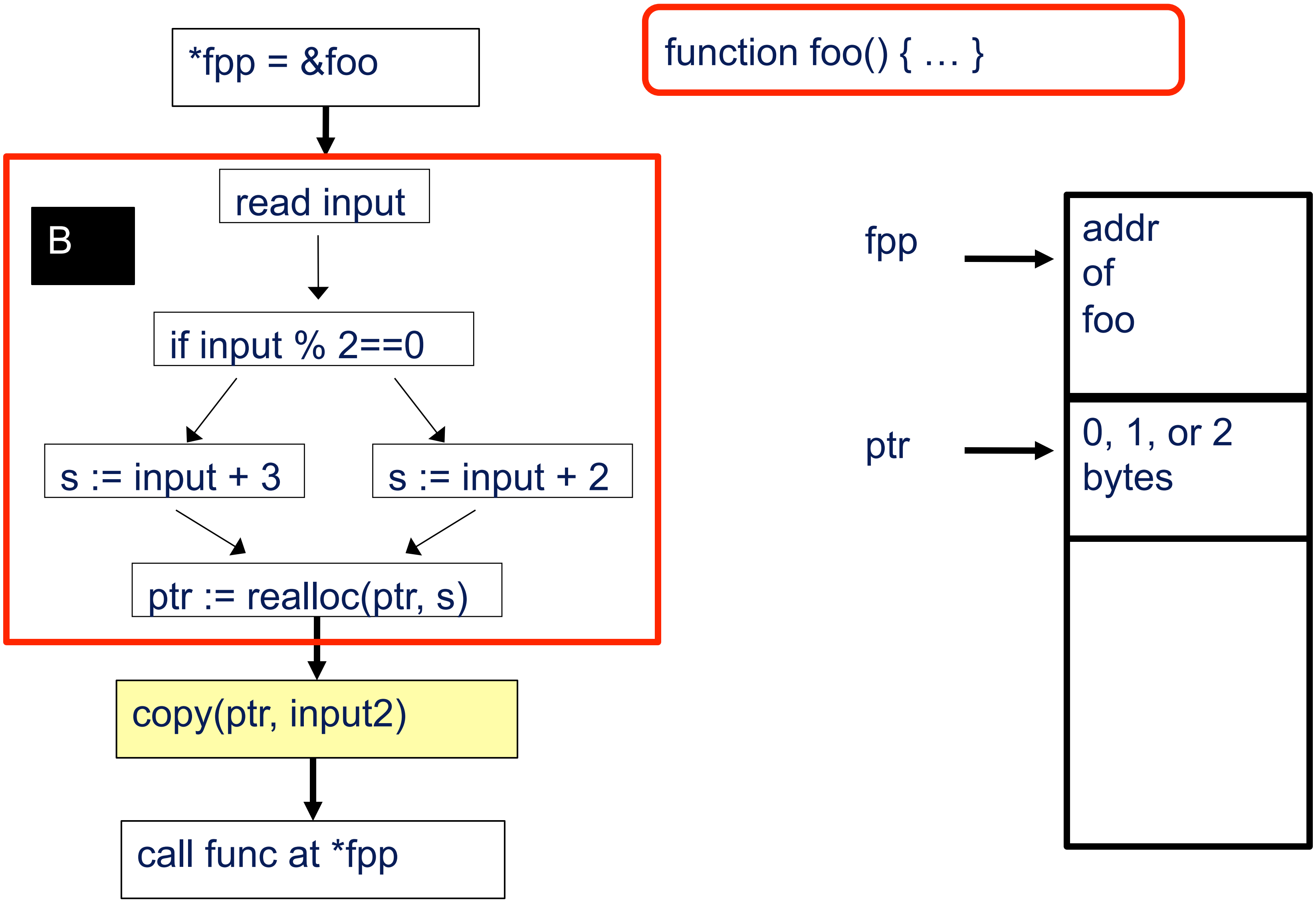
- Backward calculation makes it harder to concretize variables with values
  - E.G., system calls, external environment, configuration

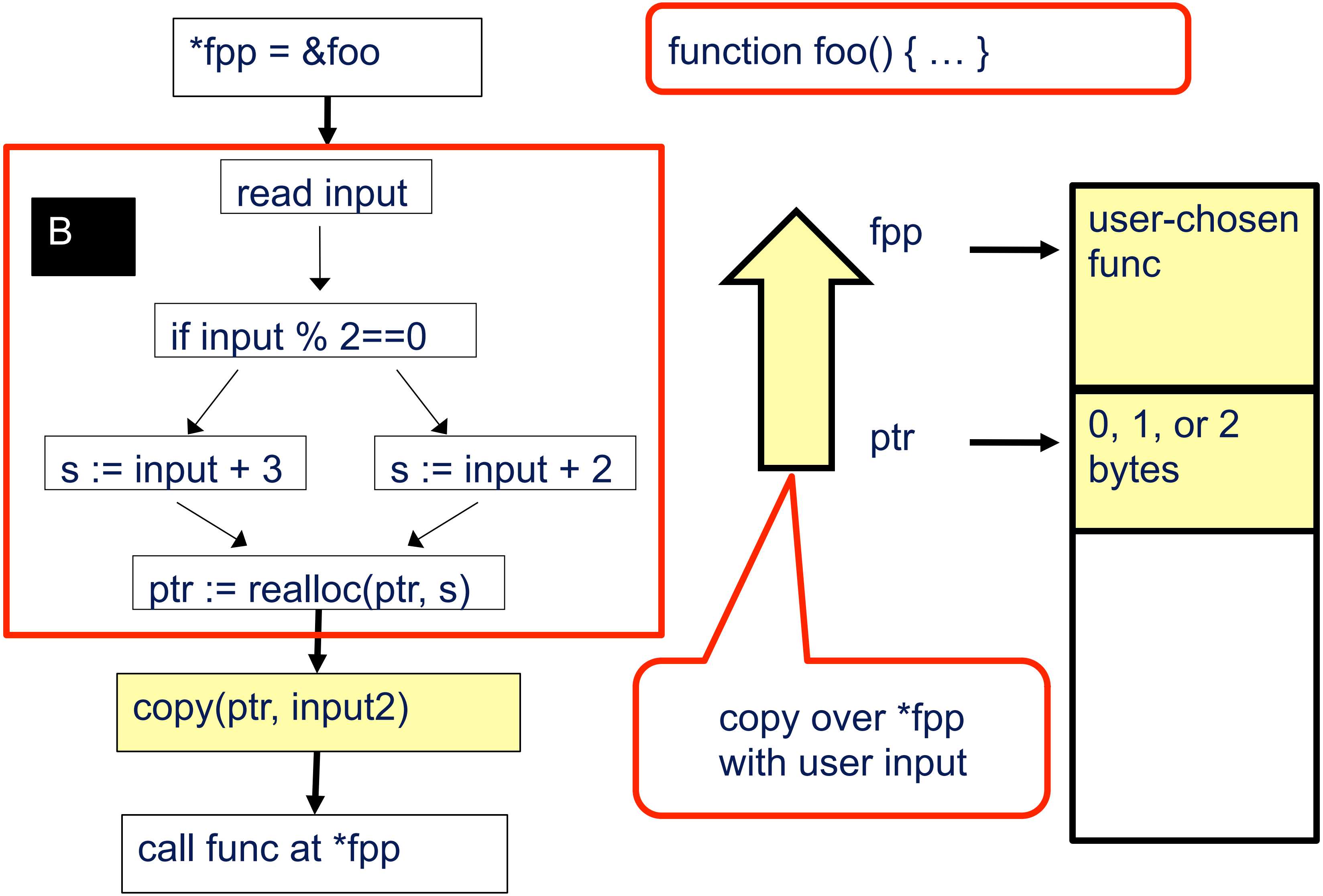
Recall

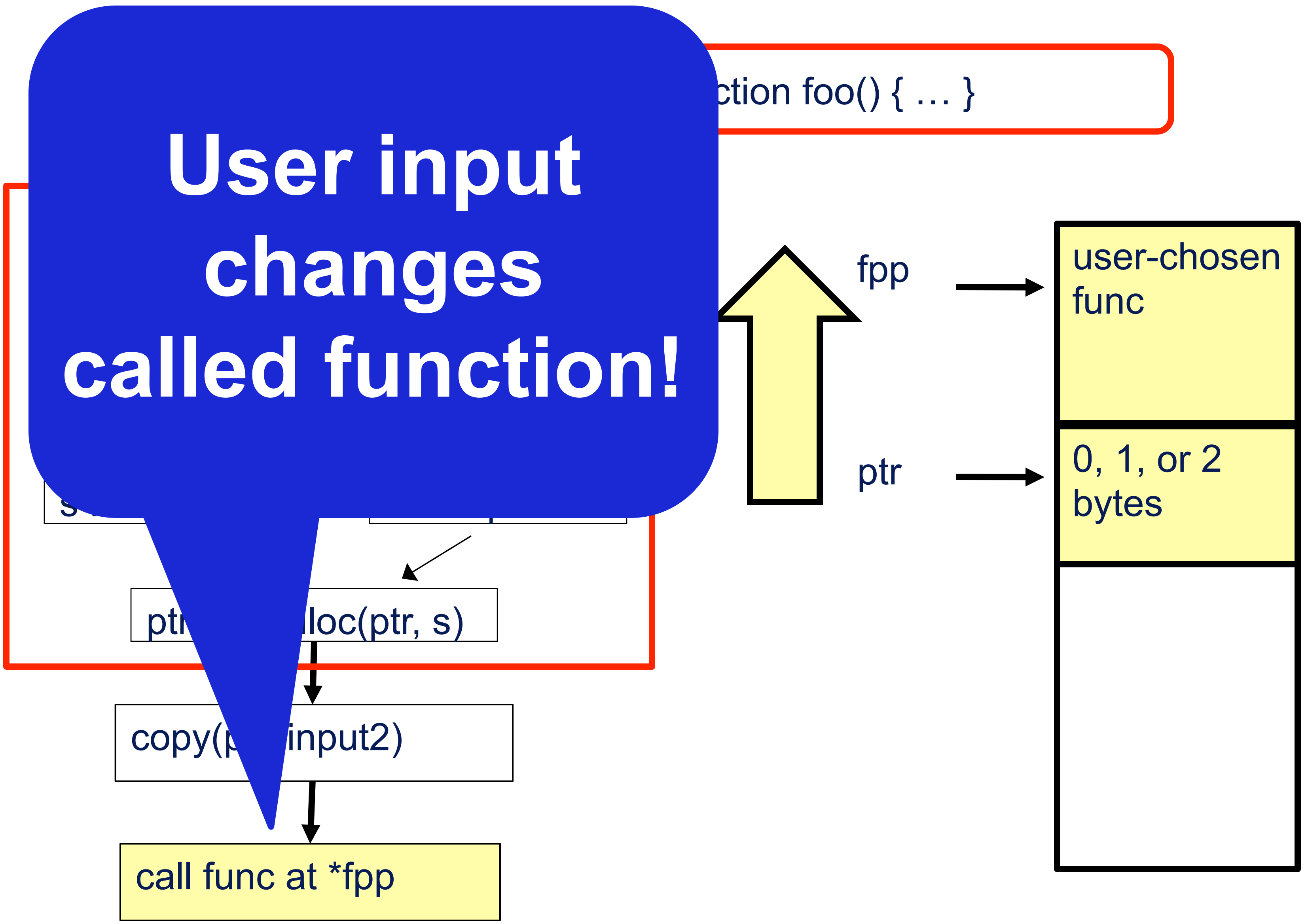


[Brumley07] only automatically generated inputs that violated new checks.
Not control flow hijacks

Using ptr is a problem


*fpp = &foo
function foo() { … }
B
read input
if input % 2==0
s := input + 3
s := input + 2
ptr := realloc(ptr, s)
copy(ptr, input2)
call func at *fpp
fpp
addr
of
foo
ptr
0, 1, or 2
bytes


*fpp = &foo
B
read input
if input % 2==0
s := input + 3
s := input + 2
ptr := realloc(ptr, s)
copy(ptr, input2)
call func at *fpp
function foo() { … }
fpp
ptr
user-chosen func
0, 1, or 2 bytes
copy over *fpp with user input



User input changes called function!
ction foo() { … }
fpp
ptr
user-chosen func
0, 1, or 2 bytes
ptr
loc(ptr, s)
copy(p
input2)
call func at *fpp

Automatic Exploit Generation

Given program, find bugs and demonstrate exploitability

APEG find bugs: P - B

AEG without patch?
Hijack control experiments? P

The iwconfig vulnerability

iwconfig: setuid wireless config

```
int get_info(int skfd, char * ifname ...){
  ...
  if(iw_get_ext(skfd, i
  {
    struct ifreq ifr;
    strcpy(ifr.ifr_name, ifname);
}

print_info(int skfd, char *ifname,…){
 ...
 get_info(skfd, ifname, …);
}

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

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

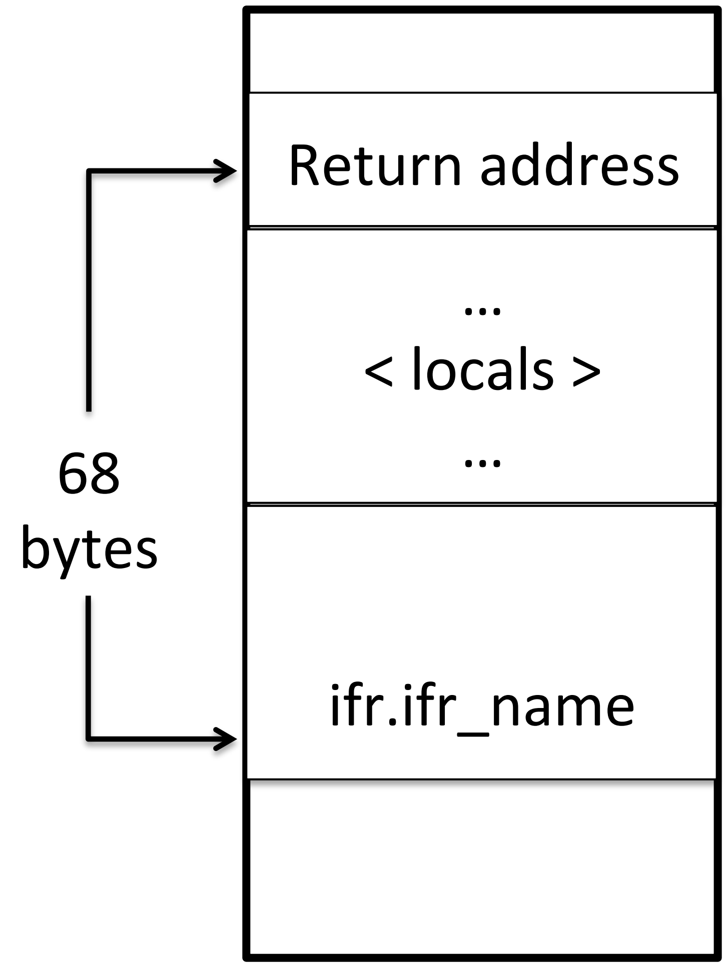
```
struct ifreq {
  char ifr_name[32]
  …
}
```

Can you spot the bug?

Is it exploitable?

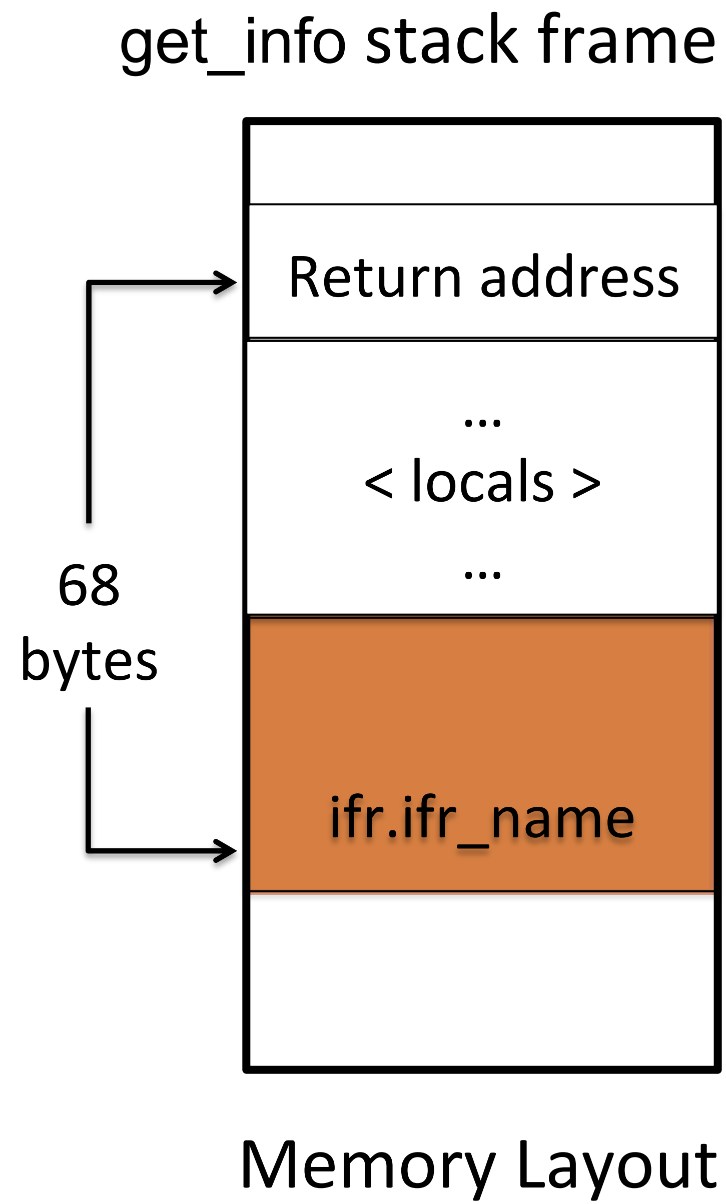
```c
int get_info(int skfd, char * ifname
  ...
  if(iw_get_ext(skfd, ifname, SIOCGI
  {
    struct ifreq ifr;
    strcpy(ifr.ifr_name, ifname);
}

print_info(int skfd, char *ifname,…)
 ...
 get_info(skfd, ifname, …);
}

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

get_info stack frame

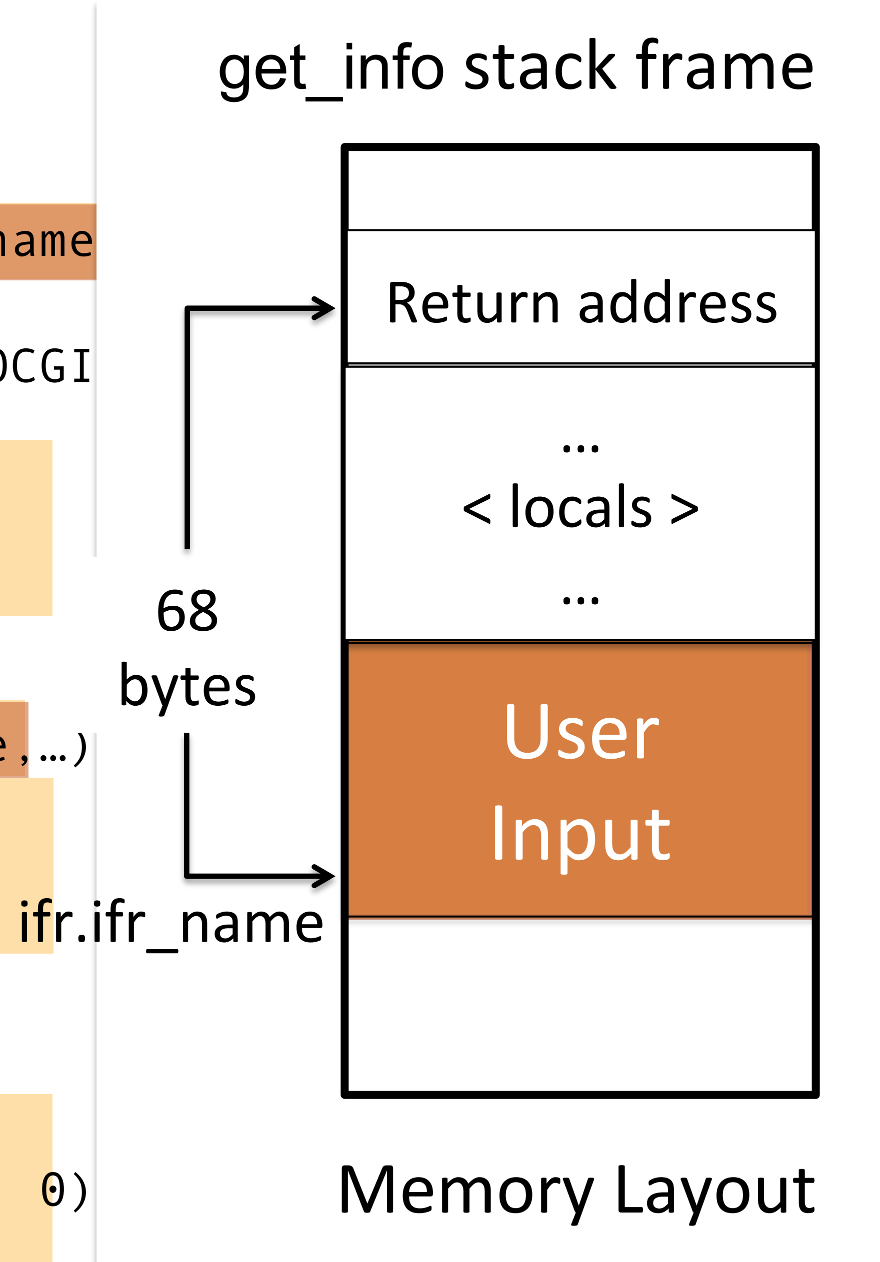


Memory Layout

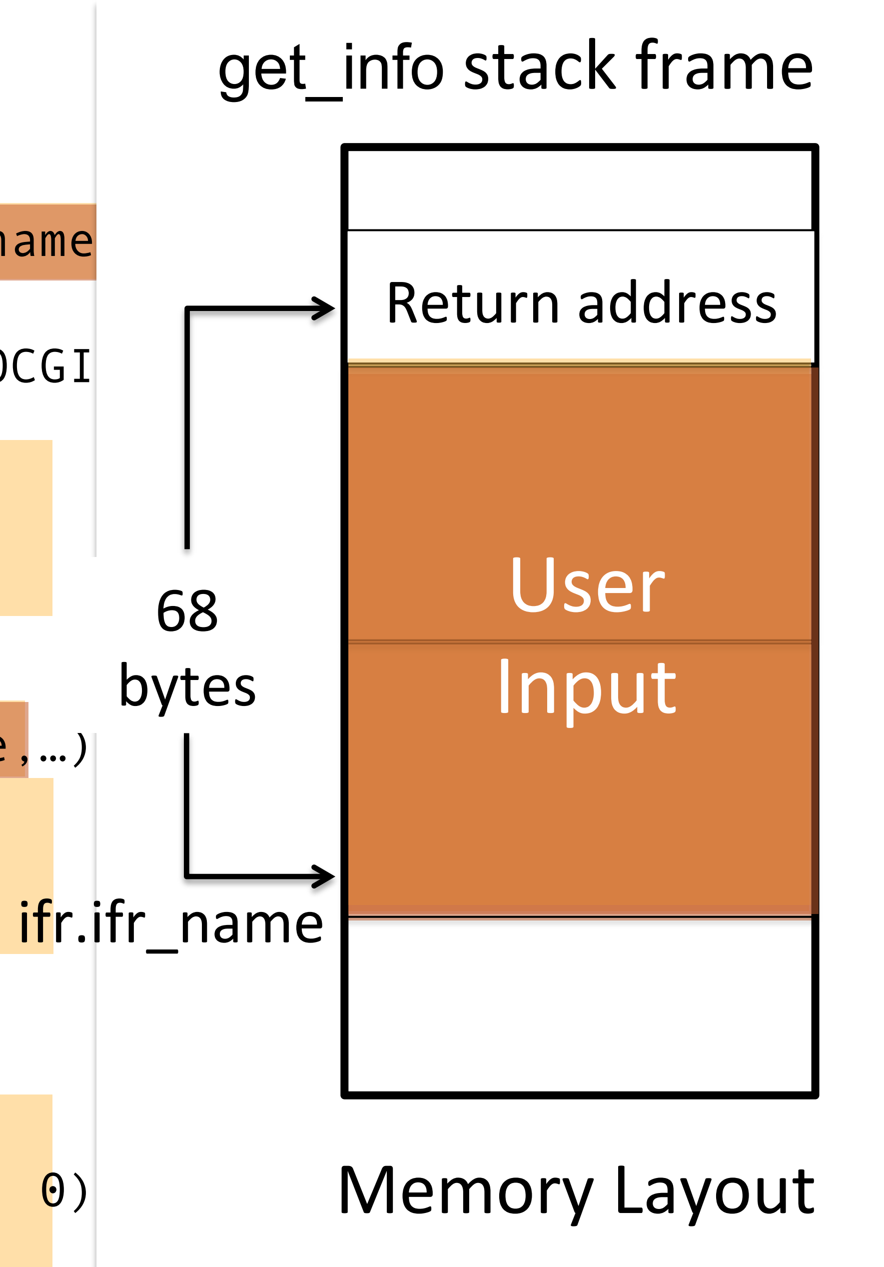
```
int get_info(int skfd, char * ifname
  ...
  if(iw_get_ext(skfd, ifname, SIOCGI
  {
    struct ifreq ifr;
    strcpy(ifr.ifr_name, ifname);
}

print_info(int skfd, char *ifname,…)
 ...
 get_info(skfd, ifname, …);
}

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



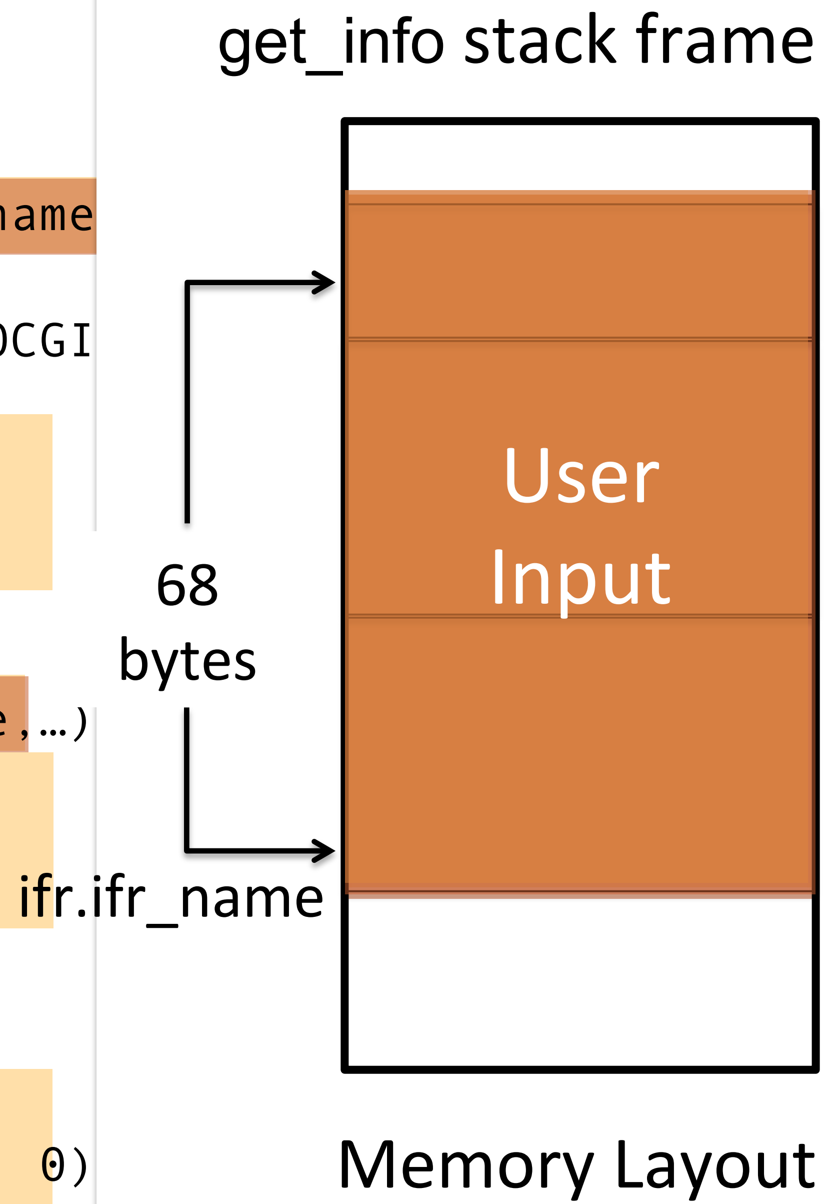
```
int get_info(int skfd, char * ifname
  ...
  if(iw_get_ext(skfd, ifname, SIOCGI
  {
    struct ifreq ifr;
    strcpy(ifr.ifr_name, ifname);
}

print_info(int skfd, char *ifname,…)
 ...
 get_info(skfd, ifname, …);
}

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



```
int get_info(int skfd, char * ifname
  ...
  if(iw_get_ext(skfd, ifname, SIOCGI
  {
     struct ifreq ifr;
     strcpy(ifr.ifr_name, ifname);
}

print_info(int skfd, char *ifname,…)
 ...
 get_info(skfd, ifname, …);
}

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

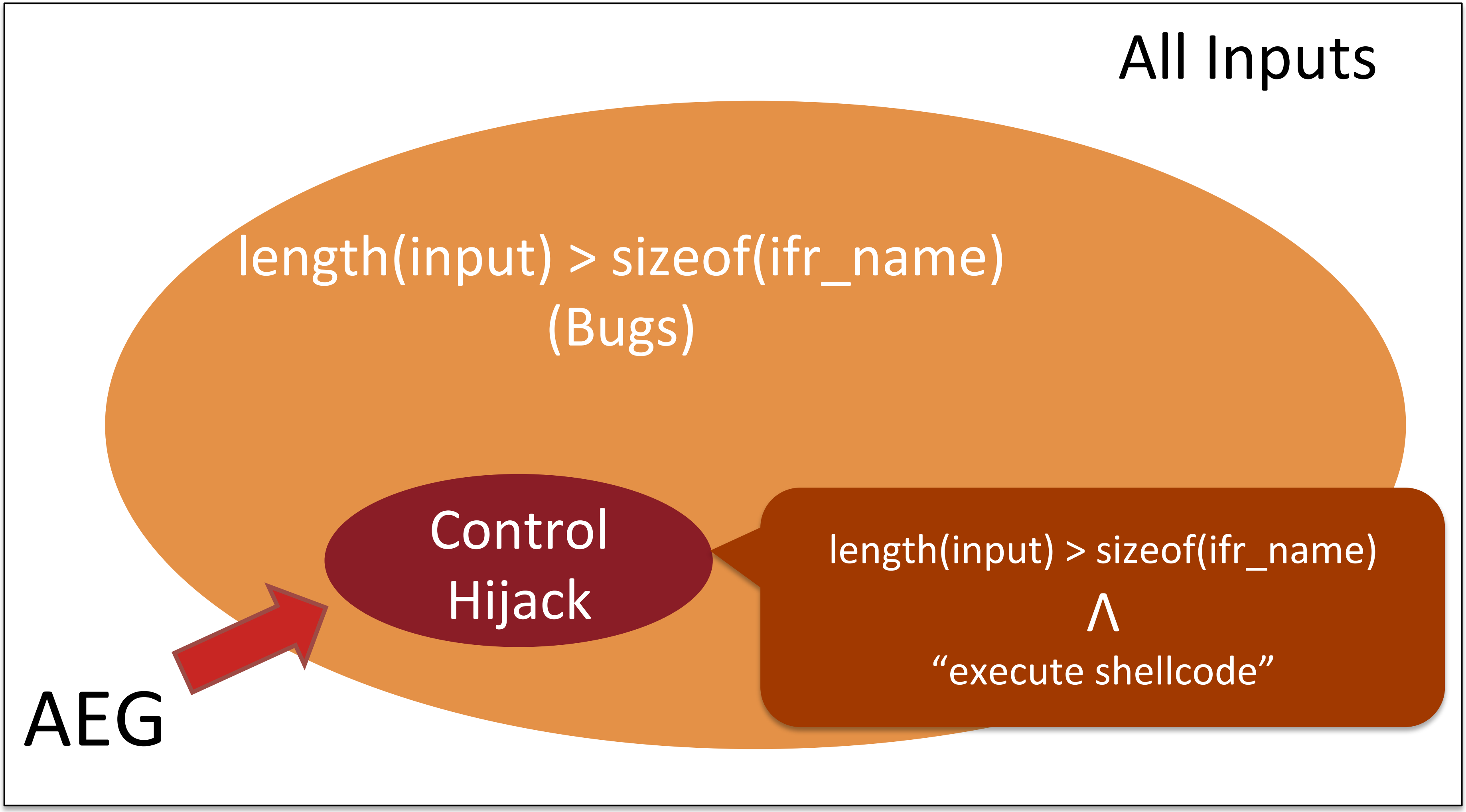


```
int get_info(int skfd, char * ifname
  ...
  if(iw_get_ext(skfd, ifname, SIOCGI
  {
    struct ifreq ifr;
    strcpy(ifr.ifr_name, ifname);
}

print_info(int skfd, char *ifname,…)
 ...
 get_info(skfd, ifname, …);
}

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



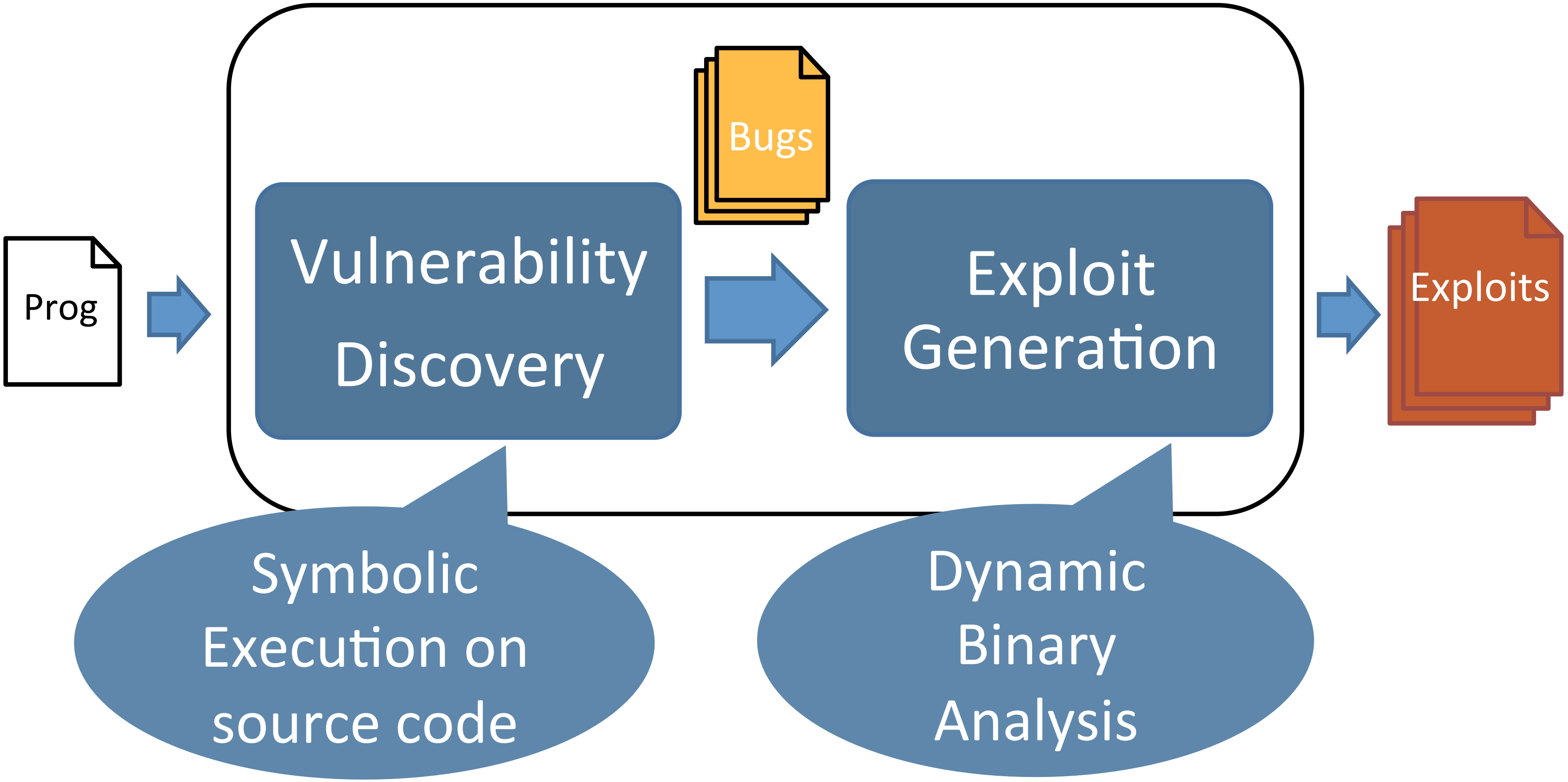
DEMO

Problem Domain



The goal

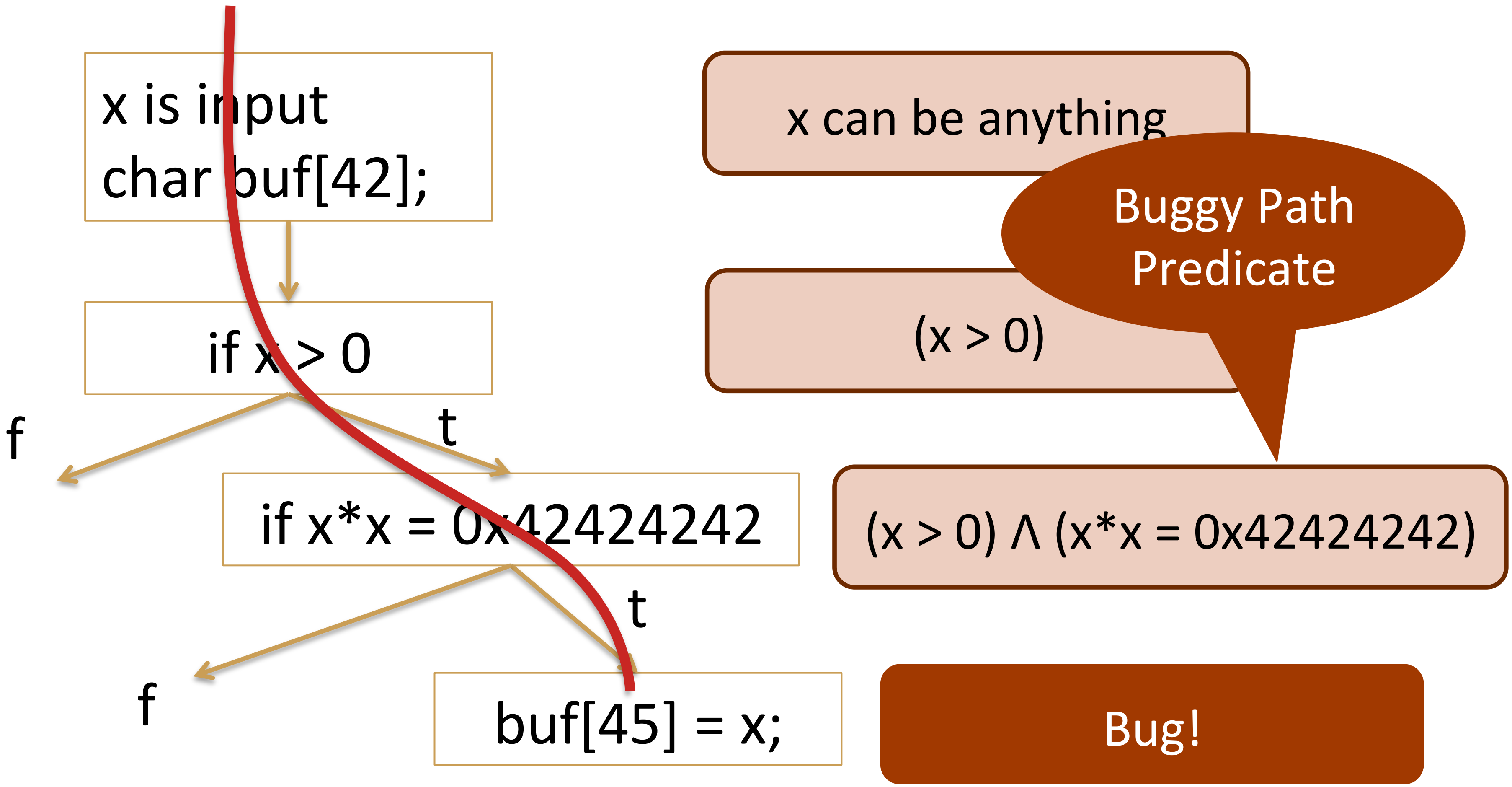
Current Approach



Vulnerability Discovery

Technique:
Symbolic Execution on source code

Symbolic Execution: How it works

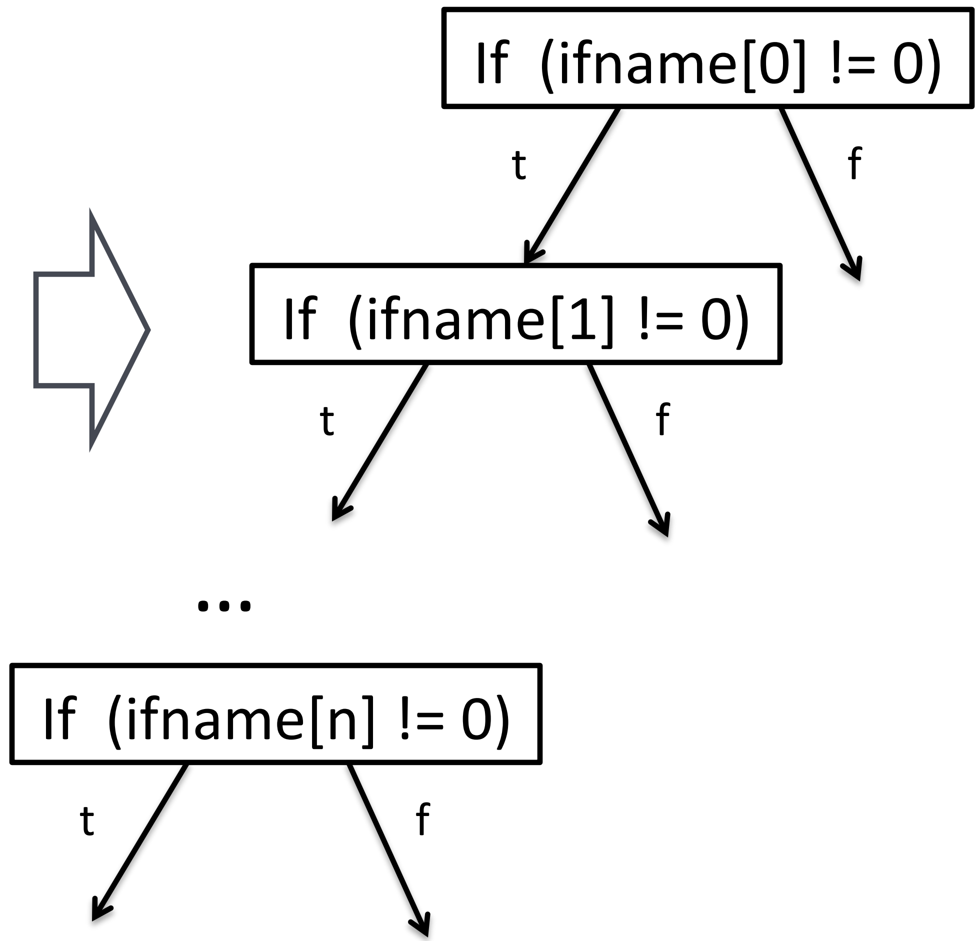


Traditional symbolic execution:
cover all paths
is ***too slow*** to find exploitable bugs

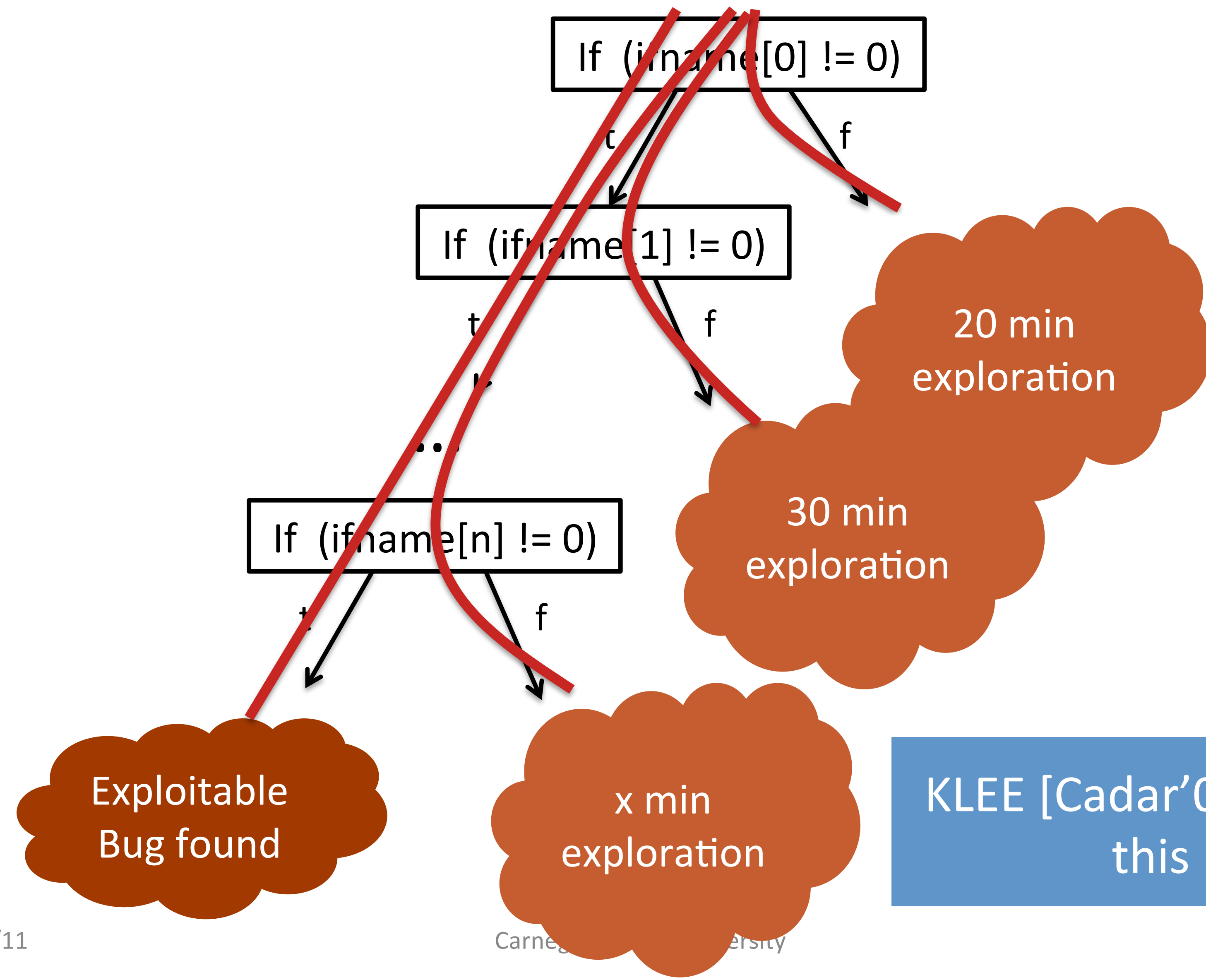
Traditional Symbolic Execution

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

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



Traditional Symbolic Execution

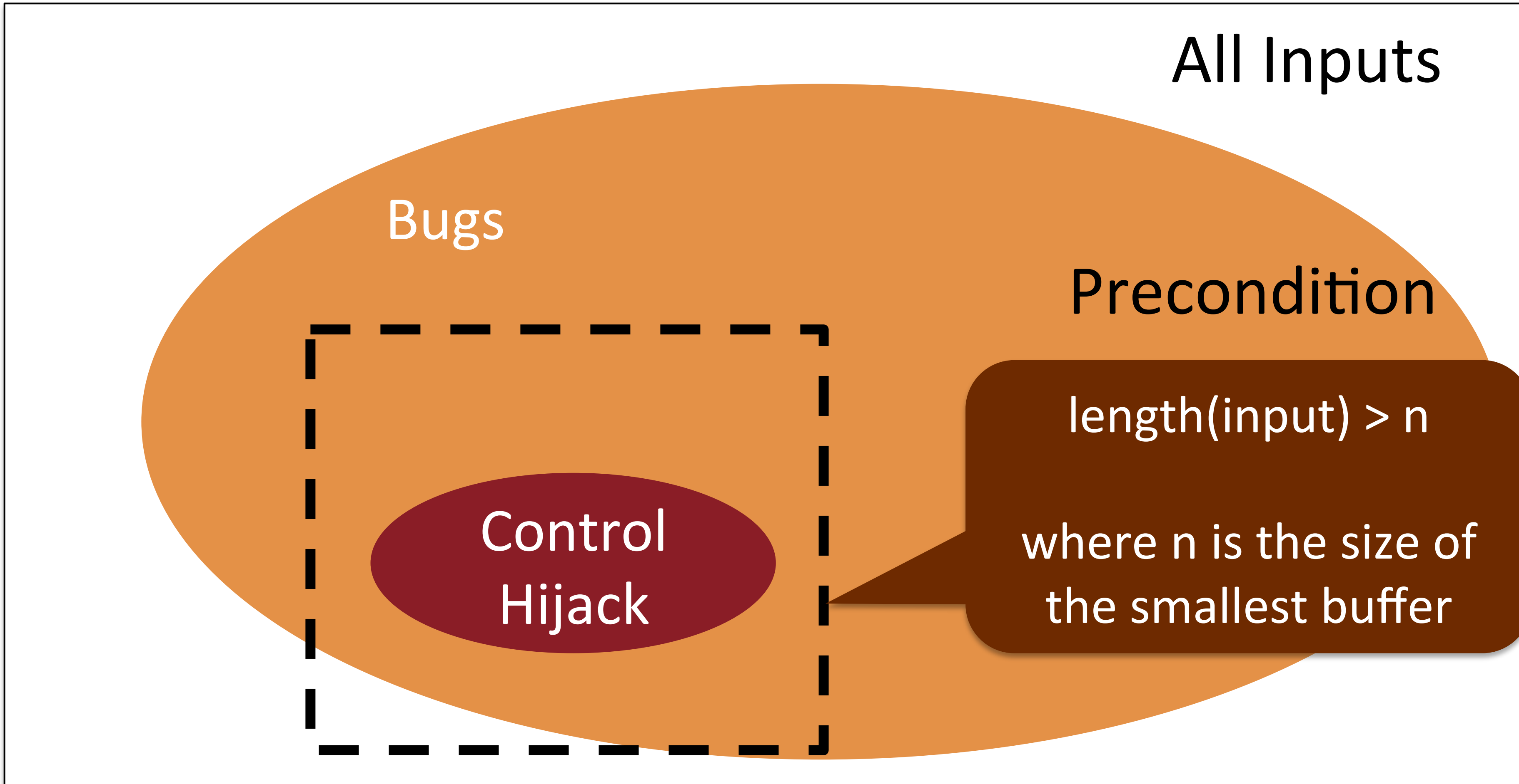


KLEE [Cadar'08] does this

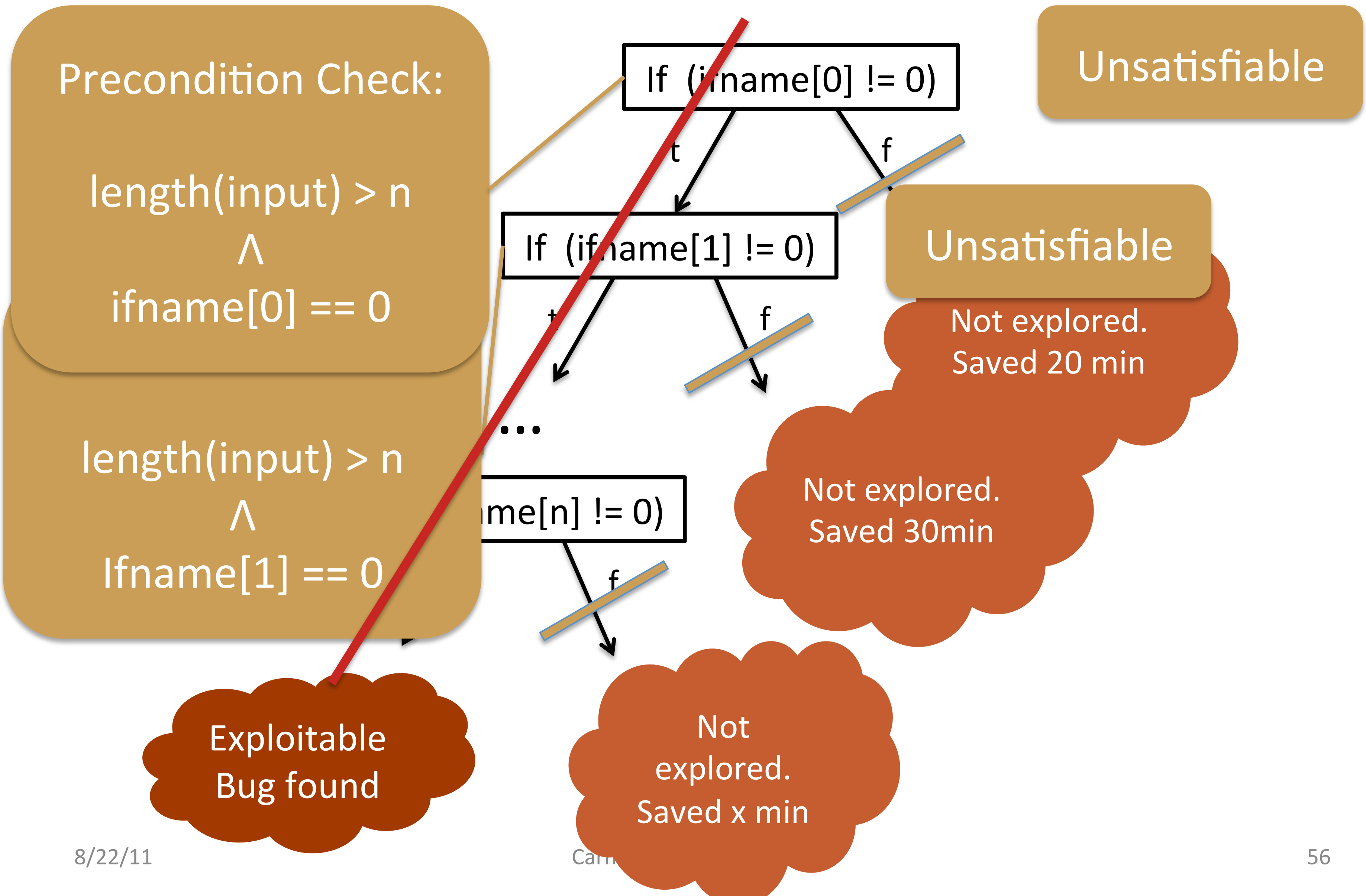
Problem:
Forward symbolic execution blindly checks program paths
(Slow to find exploitable bugs)

Our Intuition for Exploit Generation:
only explore buggy paths (Fast)

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



AEG: Preconditioned Symbolic Execution



Static Analysis Infers Preconditions

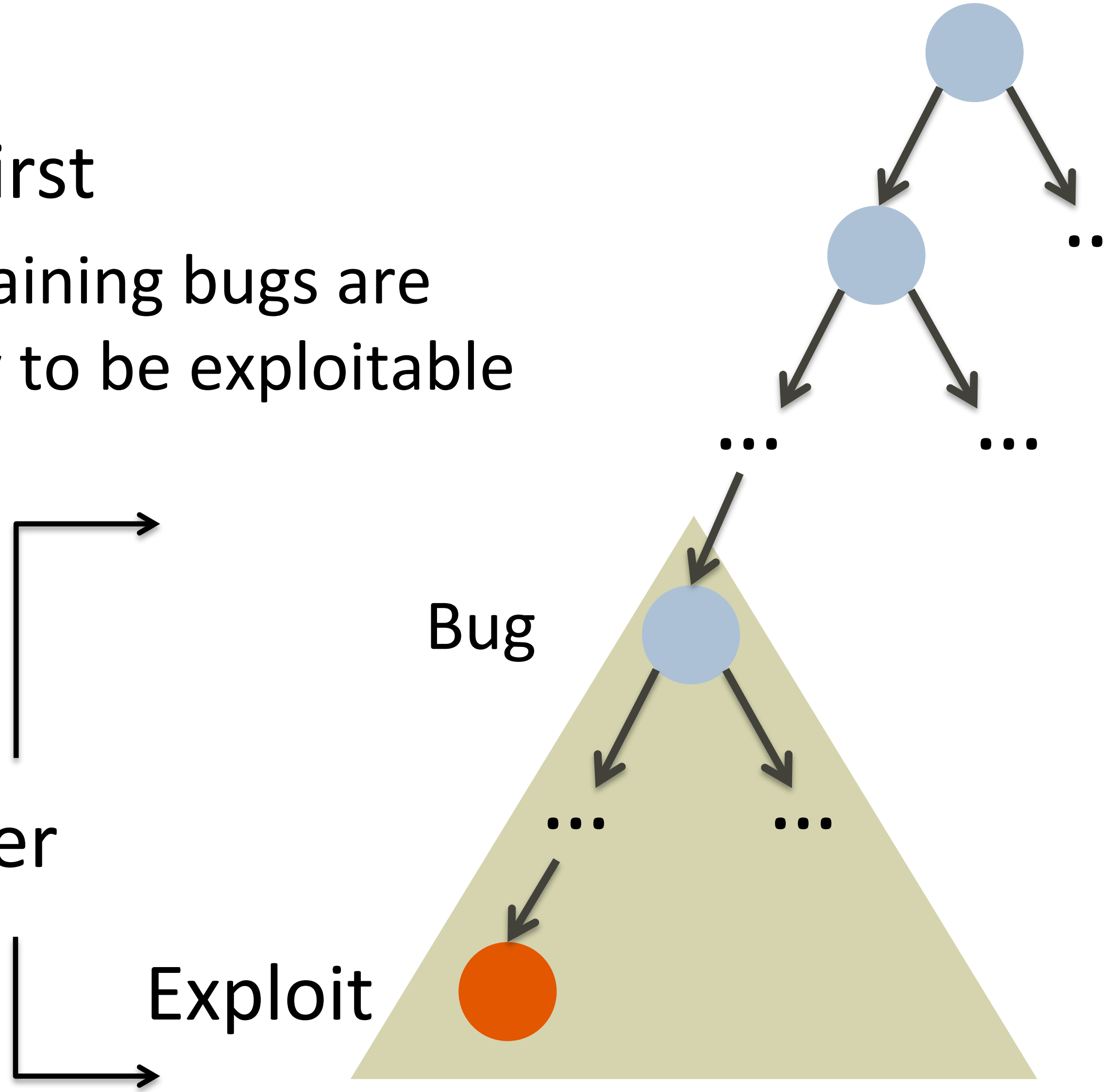
- Size of the largest statically allocated buffer
- Type of arguments
- Known prefix on input

Second Insight

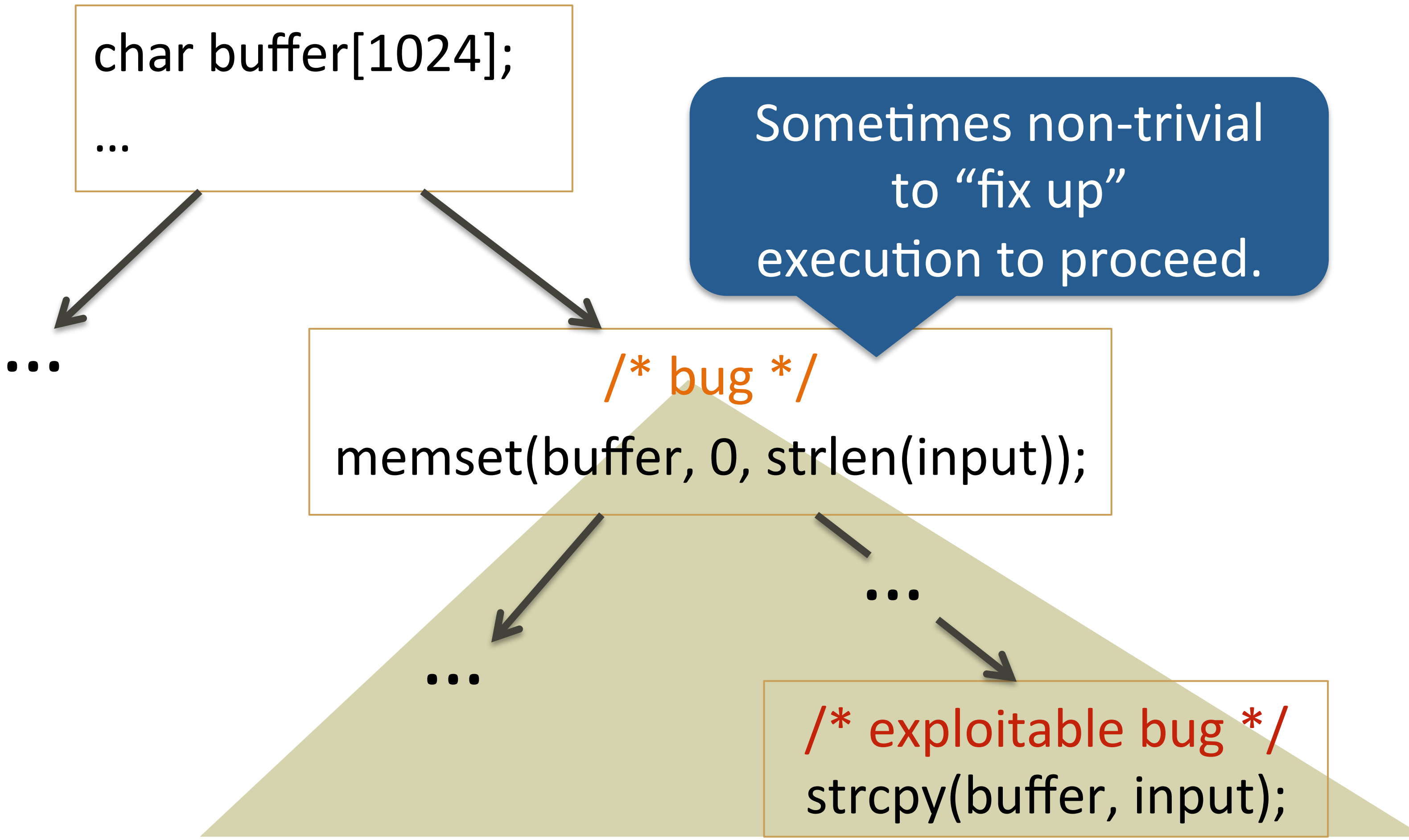
Not all paths are equally likely to be exploitable

Path Prioritization

- Buggy-path first
  - Paths containing bugs are more likely to be exploitable



Buggy Path First: Example



Given the path to a bug, how do you create an exploit?

Exploit Generation

Technique:
Dynamic Binary Analysis

Goal: Test exploitability of buggy path

Control Hijack for bug found:

length(input) > sizeof(ifr_name)

$\wedge$

length(input) > 68 bytes

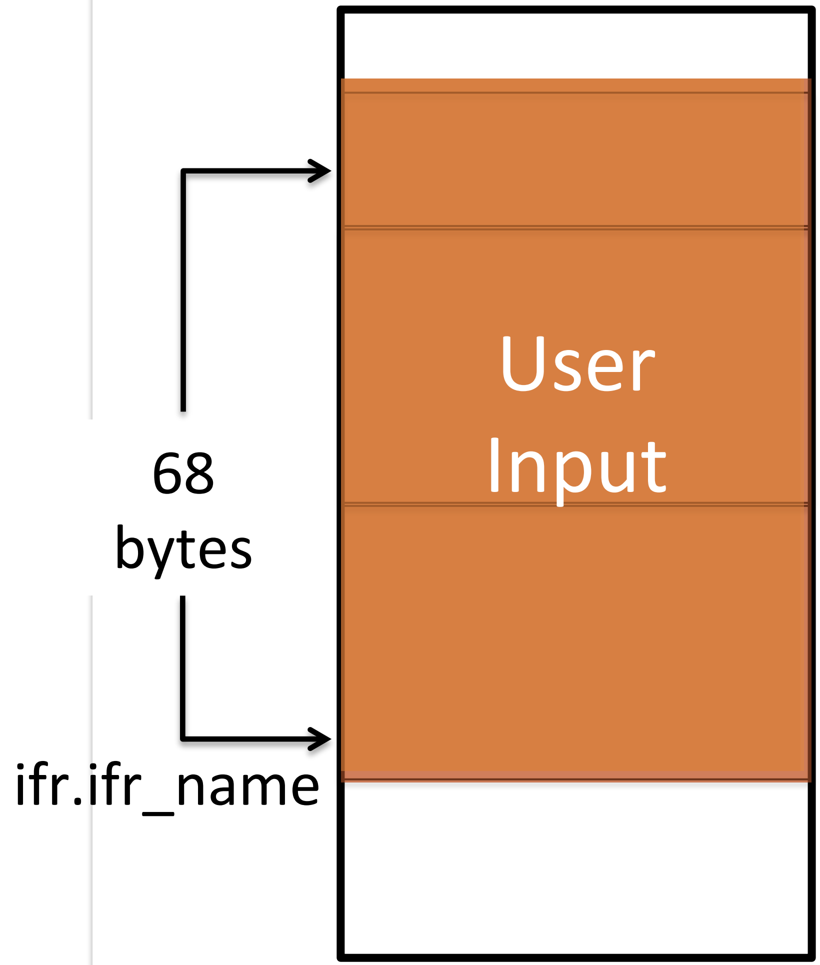
$\wedge$

input[0-63] == <shellcode>

$\wedge$

input[64-67] == <shellcode addr>

get_info stack frame



Memory Layout

Generating Exploits

Control Hijack for bug found:

length(input) > sizeof(ifr_name)

∧

length(input) > 68 bytes

∧

input[0-63] == <shellcode>

∧

input[64-67] == <shellcode addr>

SMT Solver

Example:

```
02 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01
01 01 01 01 70 f3 ff bf  31 c0 50 68 2f 2f 73 68
68 2f 62 69 6e 89 e3 50  53 89 e1 31 d2 b0 0b cd
80 01 01 01 00
```

Results Overview

AEG vs Real-world applications

Analyzed **14** applications for 3 hours and generated **16** working control-hijack exploits

Name	Advisory ID	Time	Exploit Type	Exploit Class
Iwconfig	CVE-2003-0947	1.5s	Local	Buffer Overflow
Htget	CVE-2004-0852	< 1min	Local	Buffer Overflow
Htget	-	1.2s	Local	Buffer Overflow
Ncompress	CVE-2001-1413	12. 3s	Local	Buffer Overflow
Aeon	CVE-2005-1019	3.8s	Local	Buffer Overflow
Tipxd	OSVDB-ID#12346	1.5s	Local	Format String
Glftpd	OSVDB-ID#16373	2.3s	Local	Buffer Overflow
Xserver	CVE-2007-3957	31.9s	Remote	Buffer Overflow
Aspell	CVE-2004-0548	15.2s	Local	Buffer Overflow
Corehttp	CVE-2007-4060	< 1min	Remote	Buffer Overflow
Exim	EDB-ID#796	< 1min	Local	Buffer Overflow
Socat	CVE-2004-1484	3.2s	Local	Format String
Xmail	CVE-2005-2943	< 20min	Local	Buffer Overflow
Expect	OSVDB-ID#60979	< 4min	Local	Buffer Overflow
Expect	-	19.7s	Local	Buffer Overflow
Rsync	CVE-2004-2093	< 5min	Local	Buffer Overflow

What AEG is *NOT*

Not Complete

- We do not claim to find all exploitable bugs
- Given an exploitable bug, we do not guarantee we will always find an exploit

But AEG is sound: if AEG outputs an exploit, the bug is guaranteed to be exploitable

Not A Weapon

AEG does not consider defenses, which may defend against otherwise exploitable bugs.

But a typical conservative security posture should still consider the bug “exploited”.

However…

Other Great Work

- KLEE [OSDI 08], SAGE [NDSS 08], DART[Godefroid 05], etc.
  - Goal: Generate inputs achieving high code coverage
  - Main Difference: AEG focuses on exploitable paths
- Heelan [MS Thesis'09]
  - Goal: Automatic Generation of Control-Flow Hijacking Exploits
  - Main Difference: Focuses on generating exploit once path to bug known.
- Hand-made tools [Medeiros et al, Toorcon'07]
  - Goal: Automated Exploit Development

Thank you!

dbrumley@cmu.edu
http://security.ece.cmu.edu/

Made possible with support in part from: