


Microsoft's IE Is the Most Targeted Application By Security Researchers



Unknown Lamer posted yesterday | from **darthcamaro** 

darthcamaro writes "Though Microsoft hasn't yet patched its Internet Explorer web browser in 2014, it did patch IE at least once every month in 2013. According to HP's 2013 [Cyber Risk Report](#), more researchers [tried to sell IE vulnerabilities](#) than any other product vulnerability. 'IE is the most prevalent browser on the systems that attackers want to compromise' said Jacob West, CTO of HP's Enterprise Security Group."

Integer overflows

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
        return -1;
    }
```

```
    i = atoi(argv[1]);
    s = i;

    if(s >= 80) {        /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
```

```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
nova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

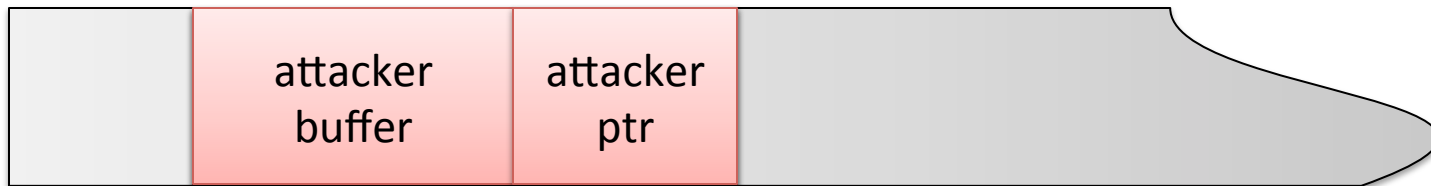
Heap overflows



Low memory
addresses



High memory
addresses



Format-string vulnerabilities

```
printf( const char* format, ... )
```

```
printf( "Hi %s %s", argv[0], argv[1] )
```

```
void main(int argc, char* argv[])  
{  
    printf( argv[1] );  
}
```

```
argv[1] = "%s%s%s%s%s%s%s%s%s%s%s"
```

Attacker controls format string gives all sorts of control

Can do control hijacking directly

	<i>Buffer Overflow</i>	<i>Format String</i>
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen
considered as	security threat	programming bug
techniques	evolved and advanced	basic techniques
visibility	sometimes very difficult to spot	easy to find

From "Exploiting format string vulnerabilities"

Summary

- Classic buffer overflow
 - corrupt program control data
 - hijack control flow easily
- Integer overflow, signedness, format string, heap overflow, ...
- These were all local privilege escalation vulns
 - Similar concepts for remote vulnerabilities
- Defenses?



Finding vulnerabilities

CS642:

Computer Security

Professor Ristenpart

<http://www.cs.wisc.edu/~rist/>

rist at cs dot wisc dot edu

Finding vulnerabilities



Manual analysis

Simple example: double free

Fuzzing tools

Static analysis, dynamic analysis

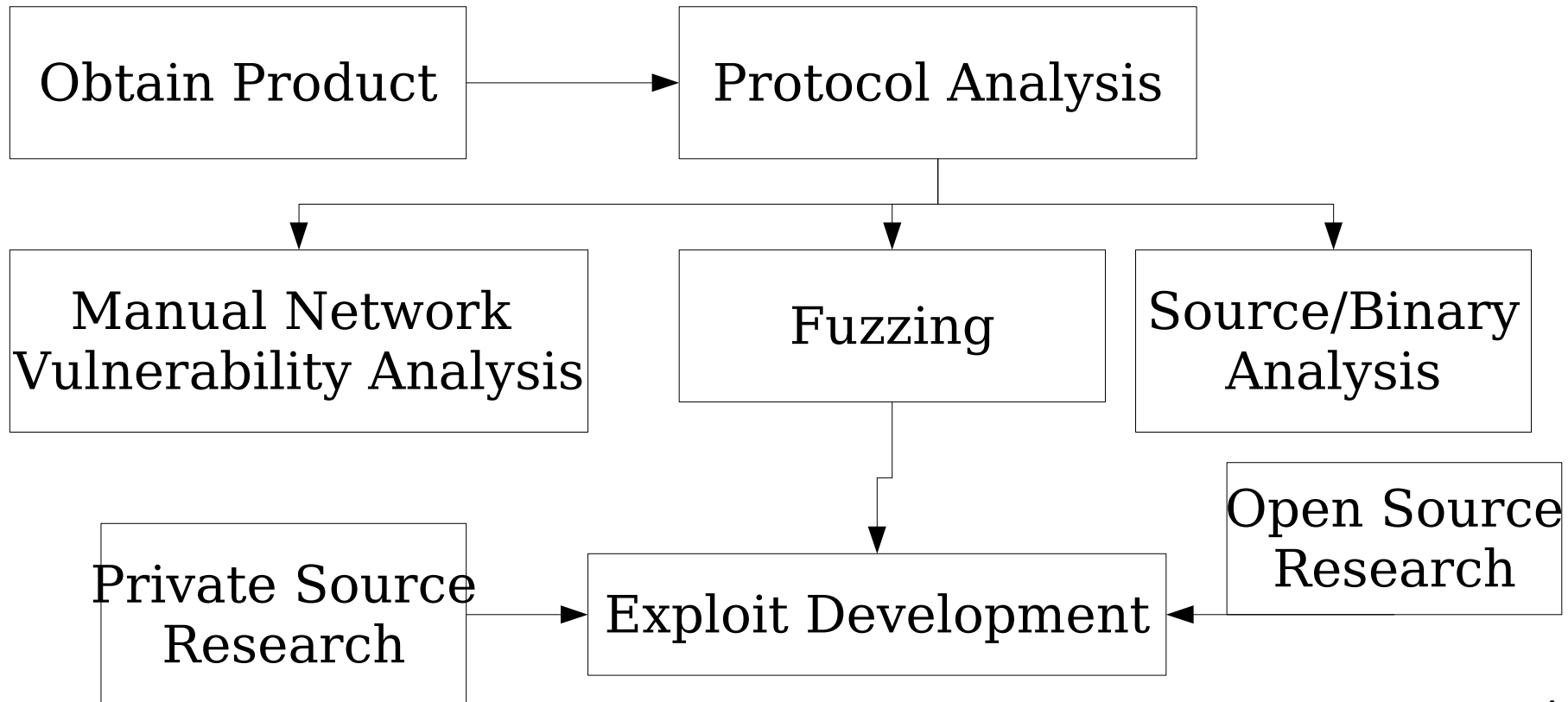
...

Hackers use People, Processes and Technology to obtain a singular goal: Information dominance



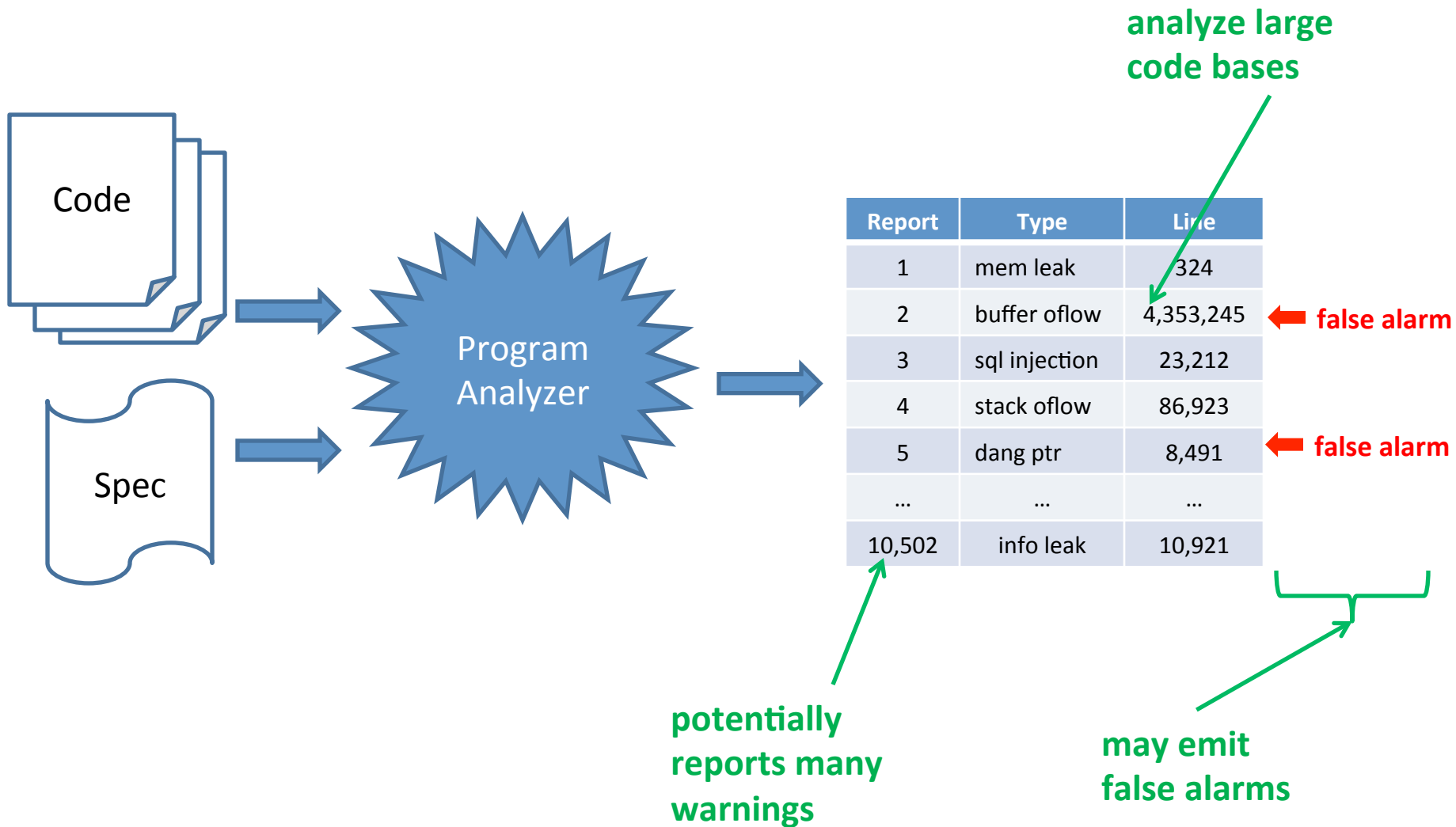
From “How Hackers Look for Bugs”, Dave Aitel

Take a sample product X and attack it remotely



From "How Hackers Look for Bugs", Dave Aitel

Program analyzers



Example program analyzers

- Manual analysis (you are the analyzer!)
- Static analysis (do not execute program)
 - Scanners
 - Abstract interpretation
 - Symbolic execution
- Dynamic analysis (execute program)
 - Debugging
 - Fuzzers
 - Ptrace

Do you have source code?

Yes: lucky you

No: can still do things, but not as easily
(missing a lot of context about program)

Program analysis: Soundness and completeness

Property	Definition
Soundness	If the program contains an error, the analysis will report a warning. “Sound for reporting correctness”
Completeness	If the analysis reports an error, the program will contain an error. “Complete for reporting correctness”

Complete

Incomplete

Sound

Reports all errors
Reports no false alarms

No false positives
No false negatives

Undecidable

Reports all errors
May report false alarms

No false negatives
False positives

Decidable

Unsound

May not report all errors
Reports no false alarms

False positives
No false negatives

Decidable

May not report all errors
May report false alarms

False negatives
False positives

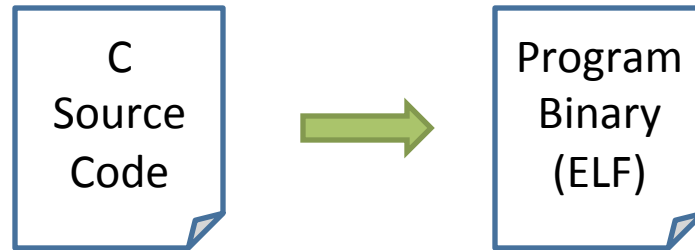
Decidable

Manual analysis

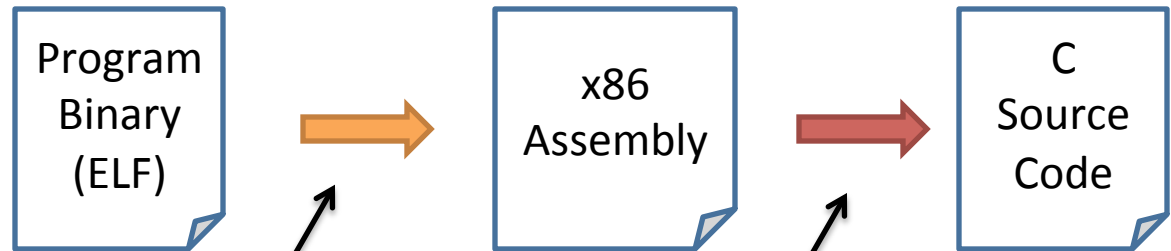
- You get a binary or the source code
- You find vulnerabilities
- Experienced analysts according to Aitel:
 - 1 hour of binary analysis:
 - Simple backdoors, coding style, bad API calls (strcpy)
 - 1 week of binary analysis:
 - Likely to find 1 good vulnerability
 - 1 month of binary analysis:
 - Likely to find 1 vulnerability *no one else will ever find*

Disassembly and decompiling

The normal compilation process



What if we start with binary?



Disassembler
(gdb, IDA Pro, OllyDebug)

Decompiler
(IDA Pro has one)

Very complex, usually poor results

Tool example: IDA Pro

Windows7 - VMware Workstation

File Edit View VM Team Tabs Help

Power Off Suspend Power On Reset Snapshot Revert Snapshot Manager Quick Switch Full Screen Unity Summary Appliance Console Record Replay

Home Boxes2 Windows7

IDA - C:\Users\vmuser\Desktop\target4

File Edit Jump Search View Options Windows Help

var_28

IDA View-A

```
.text:0048594      mov     [esp+28h+var_28], eax
.text:0048597      call   tfree
.text:004859C      mov     [esp+28h+var_28], 400h
.text:00485A3      call   tmalloc
.text:00485A8      mov     [ebp+var_10], eax
.text:00485AB      cmp     [ebp+var_10], 0
.text:00485AF      jnz    short loc_80485E5
.text:00485B1      mov     eax, ds:stderr@Glibc_2_0
.text:00485B6      mov     edx, eax
.text:00485B8      mov     eax, offset aTmallocFailure ; "tmalloc failure\n"
.text:00485BD      mov     [esp+28h+var_1C], edx
.text:00485C1      mov     [esp+28h+var_28], 10h
.text:00485C9      mov     [esp+28h+var_24], 1
.text:00485D1      mov     [esp+28h+var_28], eax
.text:00485D4      call   fwrite
.text:00485D9      mov     [esp+28h+var_28], 1
.text:00485E0      call   _exit
.text:00485E5      ;-----
.text:00485E5      loc_80485E5:          ; CODE XREF: foo+C1f
.text:00485E5      mov     [esp+28h+var_20], 400h
.text:00485ED      mov     eax, [ebp+arg_0]
.text:00485F0      mov     [esp+28h+var_24], eax
.text:00485F4      mov     eax, [ebp+var_10]
.text:00485F7      mov     [esp+28h+var_28], eax
.text:00485FA      call   obsd_strncpy
```

Names window

Name	Address	P.
fwrite	00483B0	
exit	00483C0	
_start	00483D0	P
__do_global_ctors_aux	0048400	
frame_dummy	0048460	
obsd_strncpy	0048484	
foo	00484EE	P
main	0048611	P
init	004866C	

Strings window

Address	Length	Type	String
00000011	00000011	C	tmalloc failure\n
00000014	00000014	C	target4. argc=2\n

File 'C:\Users\vmuser\Desktop\target4' is successfully loaded into the database.
Compiling file 'C:\Program Files\IDA Free\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\IDA Free\idc\onload.idc'...
Executing function 'onload'...
IDA is analyzing the input file...
You may start to explore the input file right now.
Propagating type information...
Function argument information is propagated.
The initial autoanalysis has been finished.

Alt: idle Down Disk:29GB

6:19 PM 9/19/2011

To return to your computer, move the mouse pointer outside or press Ctrl-Alt

Tool example: IDA Pro

The screenshot displays the IDA Pro interface within a VMware Workstation environment. The main window shows a disassembled assembly code window with a control flow graph (CFG) overlaid. The CFG consists of several basic blocks connected by control flow edges, representing the execution flow of the program. The assembly code is shown in a disassembled view, with instructions and their corresponding assembly mnemonics. The interface includes a menu bar, a toolbar, and a sidebar on the left with a 'Powered On' status indicator. The console window at the bottom shows the following output:

```
Compiling file 'C:\Program Files\IDA Free\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\IDA Free\idc\onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Propagating type information...
Function argument information is propagated
The initial autoanalysis has been finished.
C:\Program Files\IDA Free\idc\idc.exe (16-bit) (32-bit opcodes)
```

The console also shows system information: 'AU: idle' and 'Down Disk: 29GB'. The system tray at the bottom right indicates the time is 6:23 PM on 9/19/2011.

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

What type of vulnerability might this be?

```
movl $0xf8, (%esp)
call 0x8048364 <malloc@plt>
movl %eax, 0x14(%esp)
movl $0xf8, (%esp)
call 0x8048364 <malloc@plt>
movl %eax, 0x18(%esp)
movl 0x14(%esp), %eax
movl %eax, (%esp)
call 0x8048354 <free@plt>
movl 0x18(%esp), %eax
movl %eax, (%esp)
call 0x8048354 <free@plt>
movl $0x200, (%esp)
call 0x8048364 <malloc@plt>
movl %eax, 0x1c(%esp)
movl 0xc(%ebp), %eax
addl $0x4, %eax
movl (%eax), %eax
movl $0x1ff, 0x8(%esp)
movl %eax, 0x4(%esp)
movl 0x1c(%esp), %eax
movl %eax, (%esp)
call 0x8048334 <strncpy@plt>
movl 0x18(%esp), %eax
movl %eax, (%esp)
call 0x8048354 <free@plt>
movl 0x1c(%esp), %eax
movl %eax, (%esp)
call 0x8048354 <free@plt>
leave
ret
```



```
main( int argc, char* argv[] ) {
    char* b1;
    char* b2;
    char* b3;

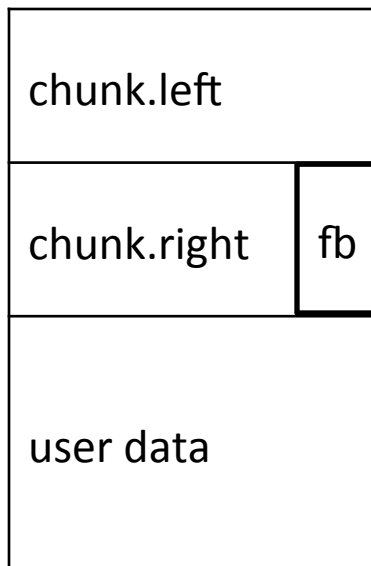
    if( argc != 3 ) then return 0;
    if( argv[2] != 31337 )
        complicatedFunction();
    else {
        b1 = (char*)malloc(248);
        b2 = (char*)malloc(248);
        free(b1);
        free(b2);
        b3 = (char*)malloc(512);
        strncpy( b3, argv[1], 511 );
        free(b2);
        free(b3);
    }
}
```

Double-free vulnerability

Double-free vulnerabilities

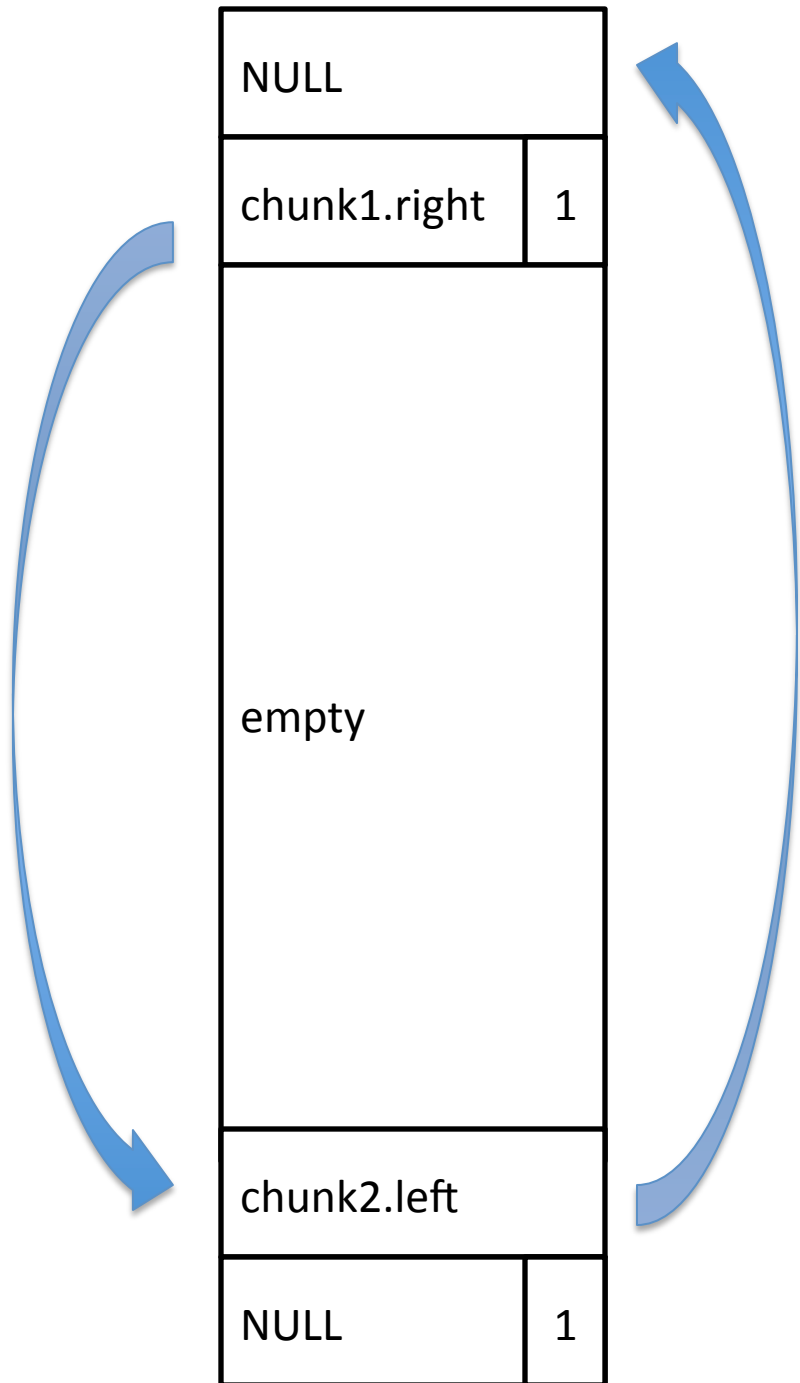
Can corrupt the state of the heap management

Say we use a simple doubly-linked list malloc implementation with control information stored alongside data



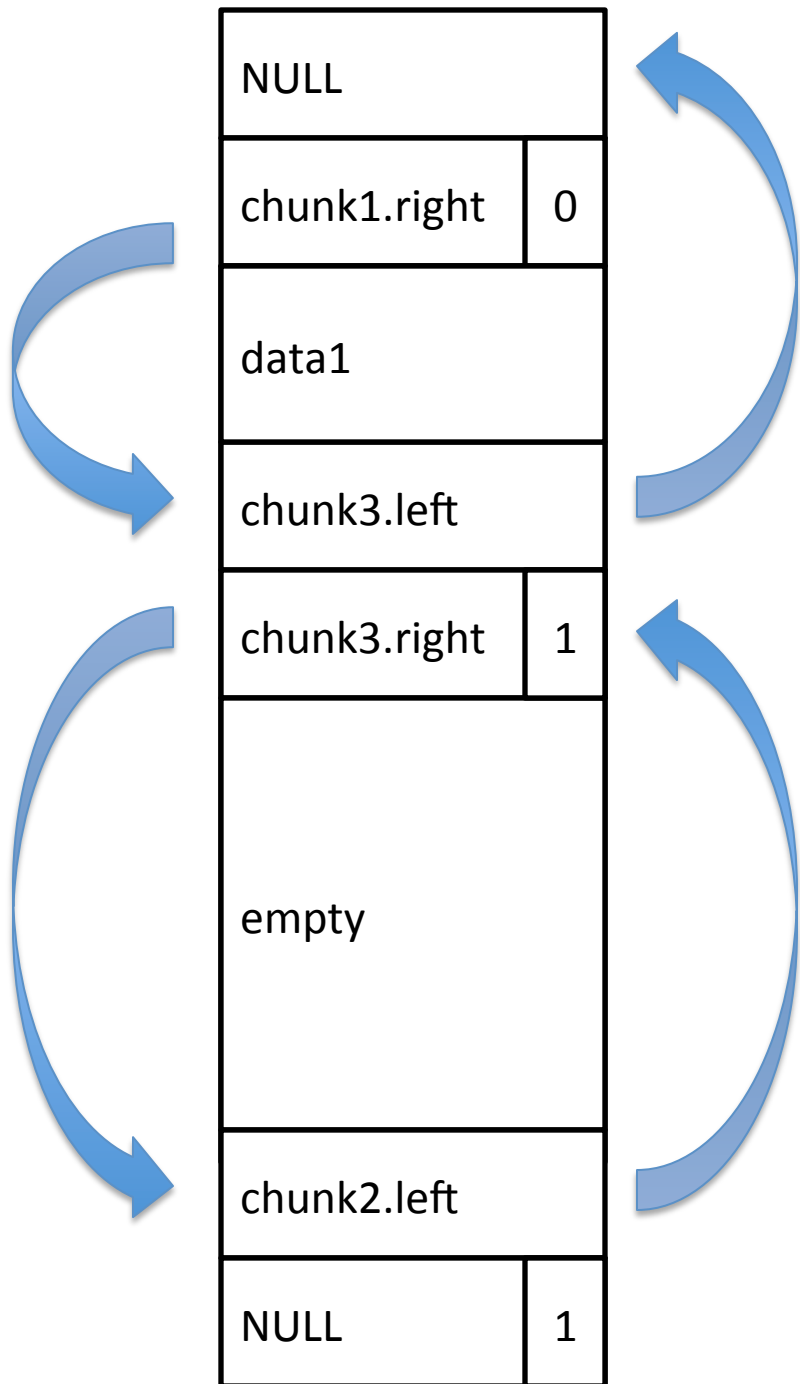
Chunk has:

- 1) left ptr (to previous chunk)
- 2) right ptr (to next chunk)
- 3) free bit which denotes if chunk is free
this reuses low bit of right ptr
because we will align chunks
- 4) user data



malloc()

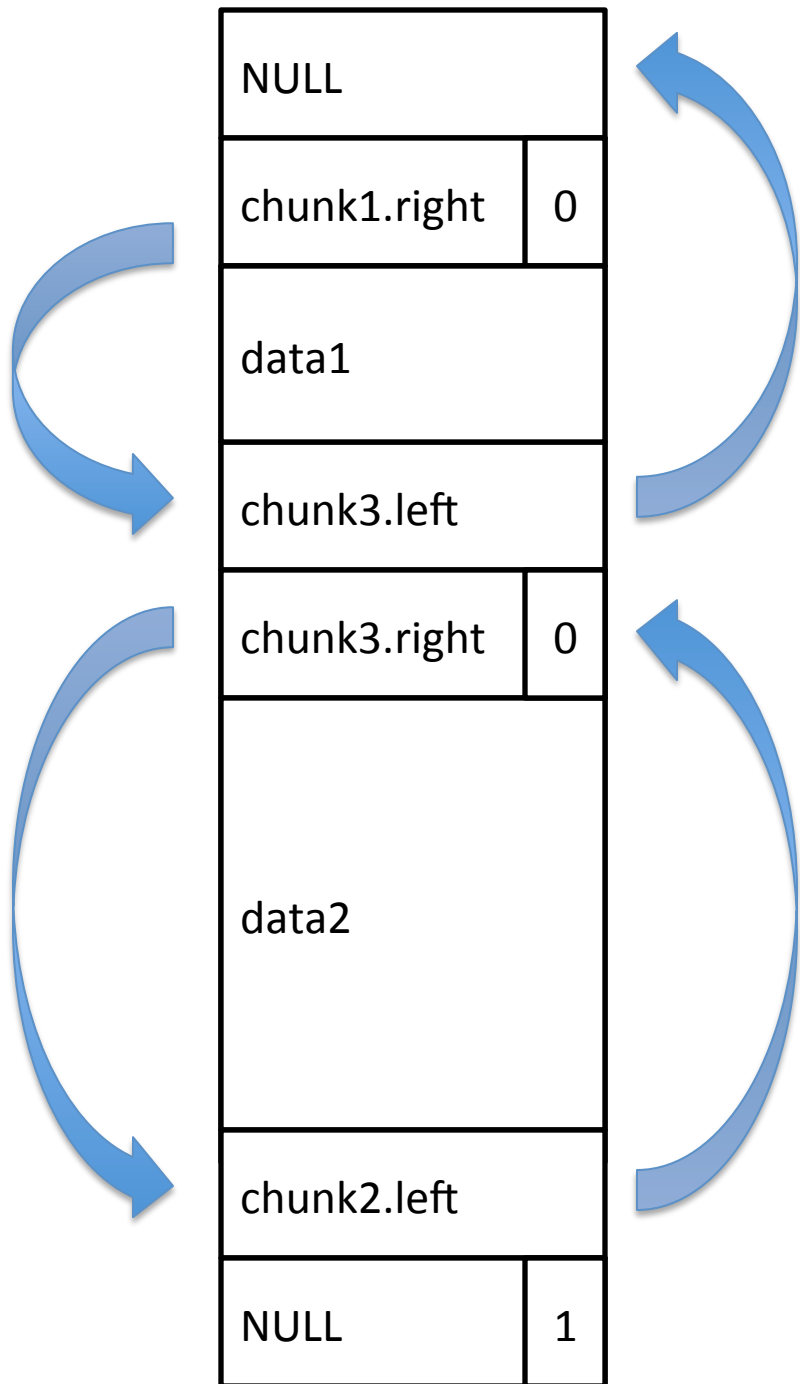
- search left-to-right for free chunk
- modify pointers



malloc()

- search left-to-right for free chunk
- modify pointers

```
b1 = malloc( BUF_SIZE1 );
```



malloc()

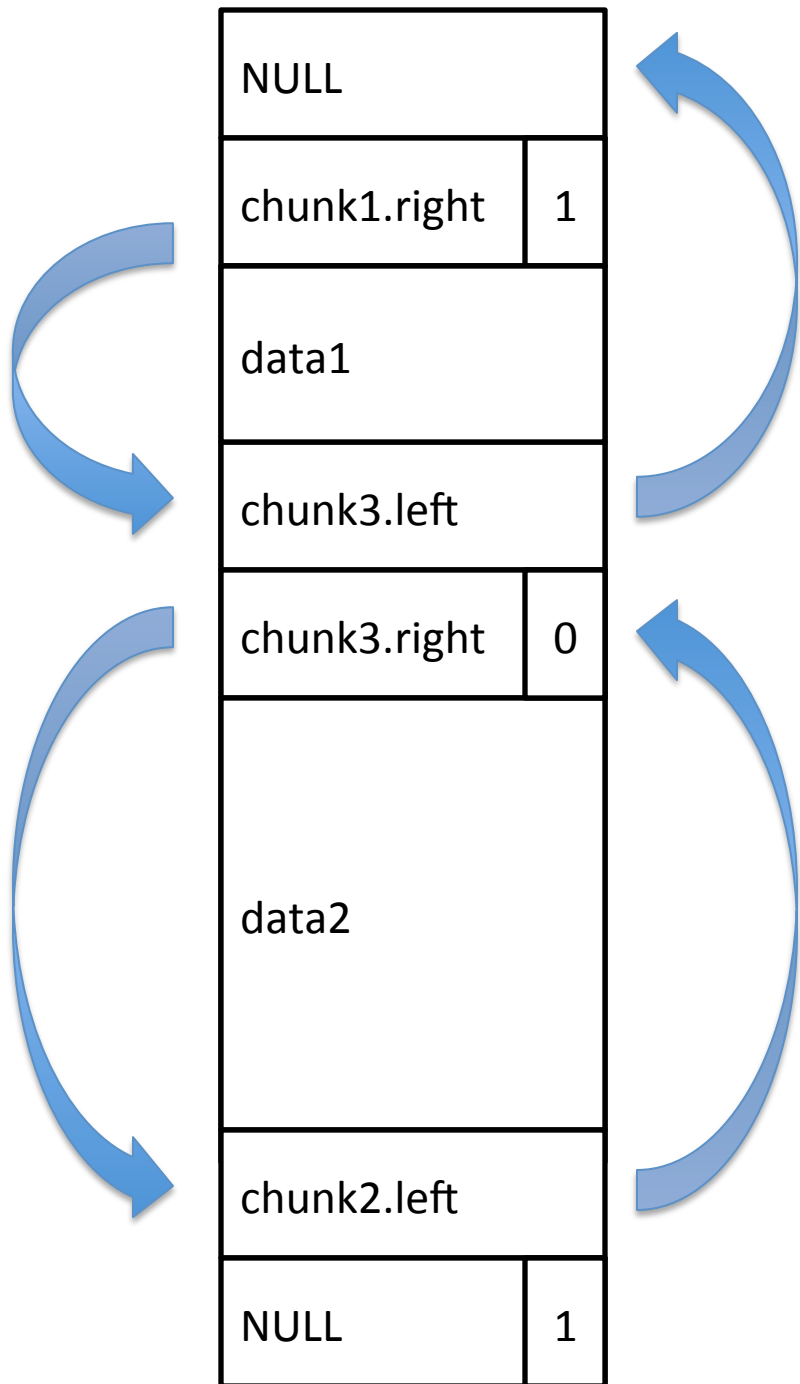
- search left-to-right for free chunk
- modify pointers

b1 = malloc(BUF_SIZE1)

b2 = malloc(BUF_SIZE2)

free()

- Consolidate with free neighbors



malloc()

- search left-to-right for free chunk
- modify pointers

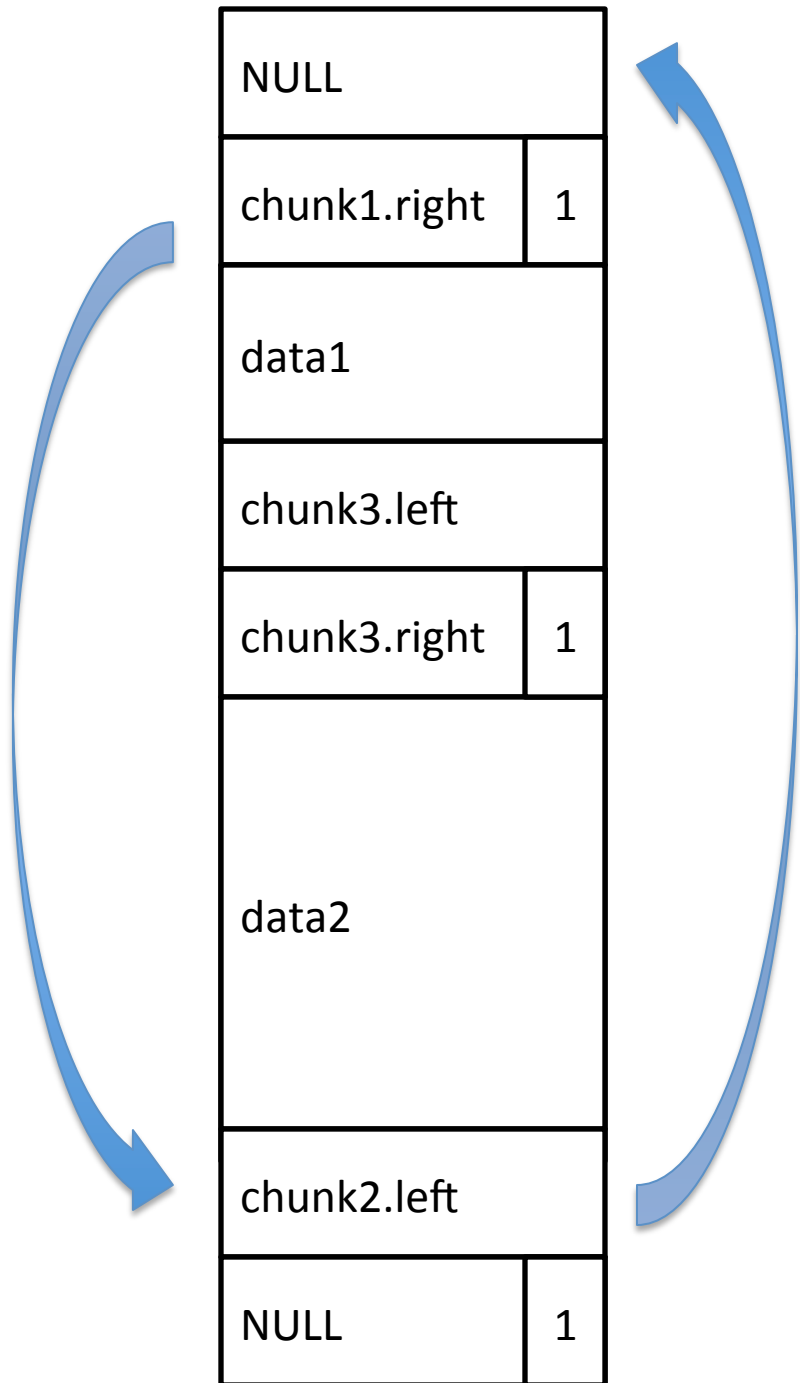
b1 = malloc(BUF_SIZE1)

b2 = malloc(BUF_SIZE2)

free()

- Consolidate with free neighbors

free(b1)



malloc()

- search left-to-right for free chunk
- modify pointers

b1 = malloc(BUF_SIZE1)

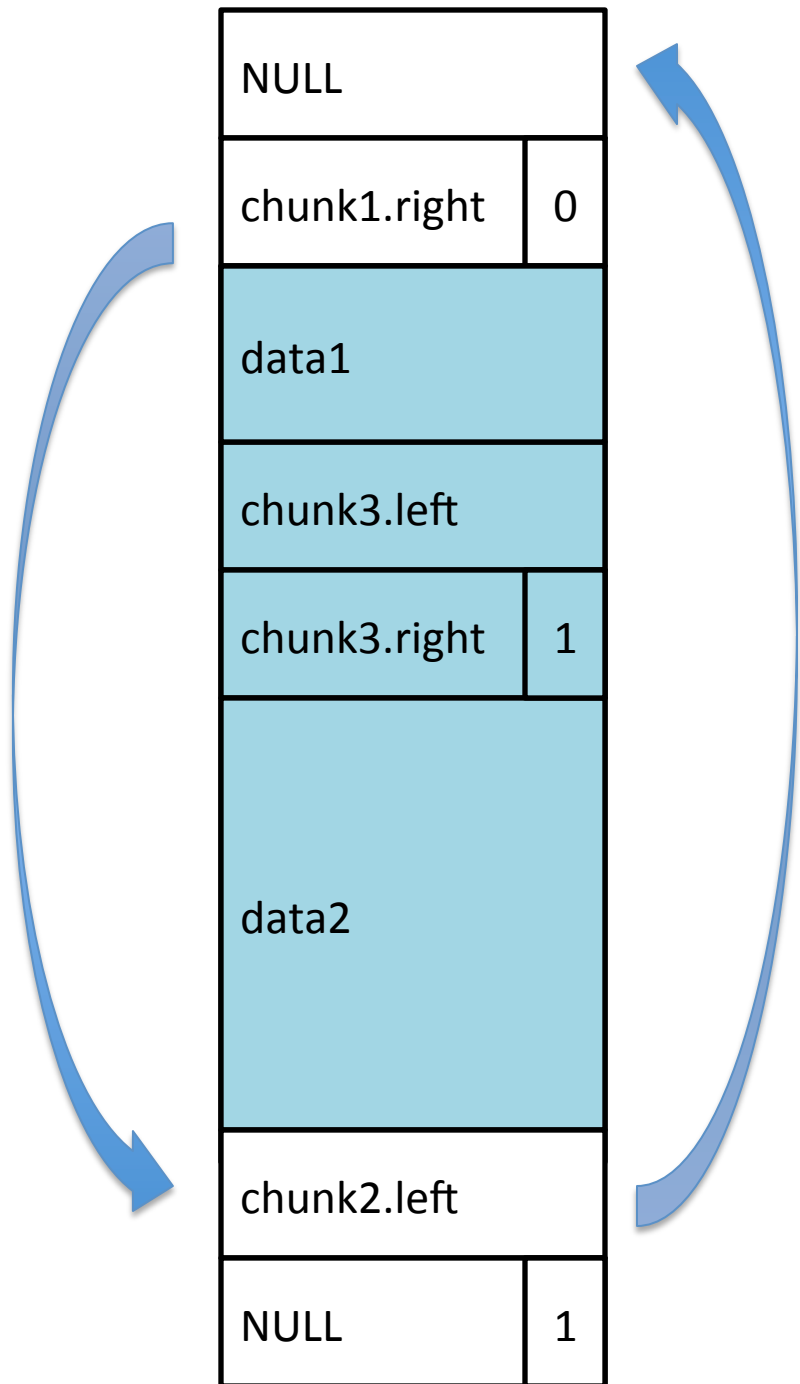
b2 = malloc(BUF_SIZE2)

free()

- Consolidate with free neighbors

free(b1)

free(b2)



malloc()

- search left-to-right for free chunk
- modify pointers

b1 = malloc(BUF_SIZE1)

b2 = malloc(BUF_SIZE2)

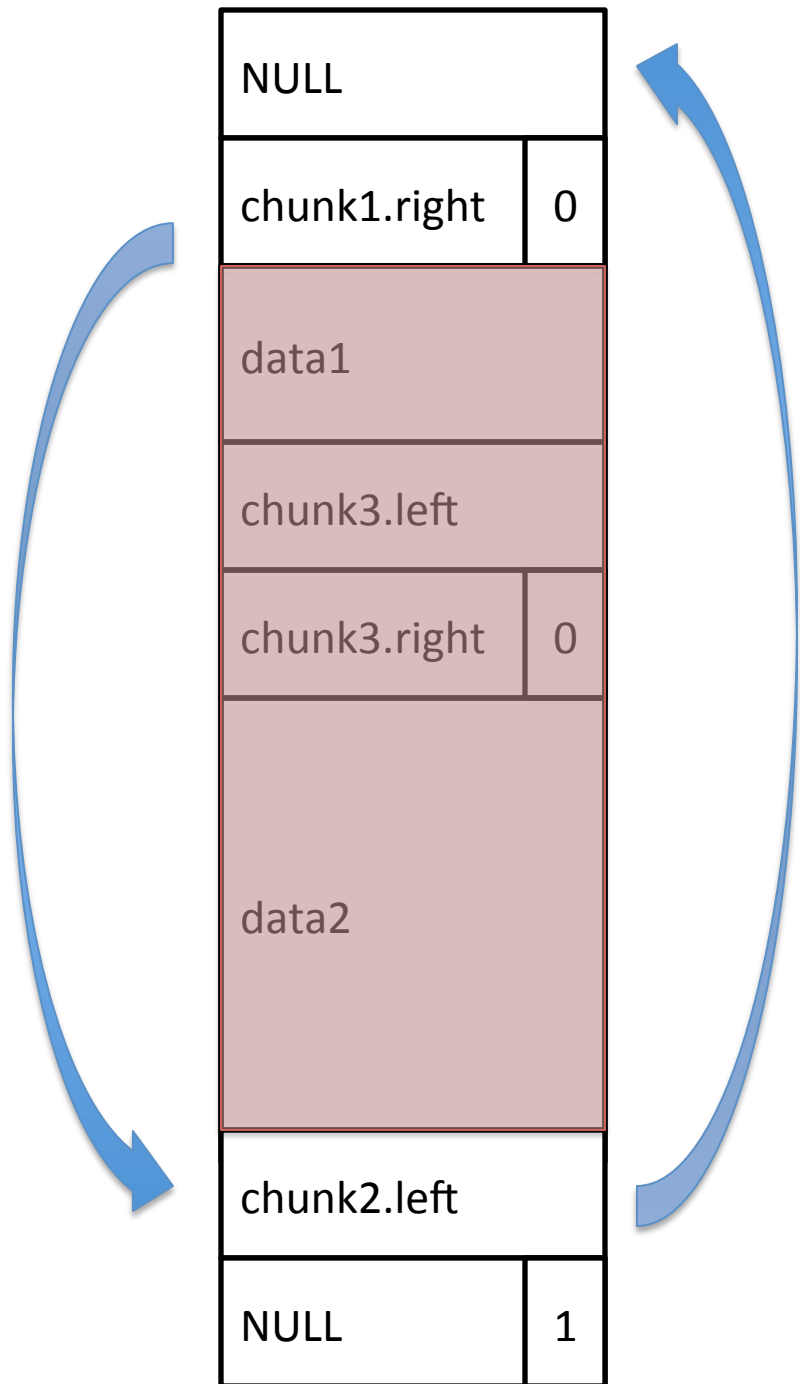
free()

- Consolidate with free neighbors

free(b1)

free(b2)

b3 = malloc(BUF_SIZE1 + BUF_SIZE2)



malloc()

- search left-to-right for free chunk
- modify pointers

b1 = malloc(BUF_SIZE1)

b2 = malloc(BUF_SIZE2)

free()

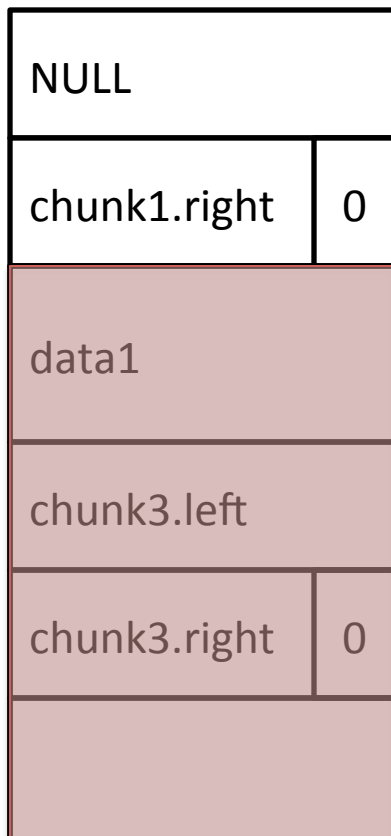
- Consolidate with free neighbors

free(b1)

free(b2)

b3 = malloc(BUF_SIZE1 + BUF_SIZE2)

strncpy(b3, argv[1], BUF_SIZE1+BUF_SIZE2-1)



malloc()

- search left-to-right for free chunk
- modify pointers

b1 = malloc(BUF_SIZE1)

b2 = malloc(BUF_SIZE2)

free()

- Consolidate with free neighbors

free(b1)

free(b2)

b3 = malloc(BUF_SIZE1 + BUF_SIZE2)

strncpy(b3, argv[1], BUF_SIZE1+BUF_SIZE2-1)

free(b2)

Interprets b2-8 as a chunk3.left

Interprets b2-4 as a chunk3.right

(b2 - 8)->left->right = (b2-8)->right

(b2 - 8)->right->left = (b2-8)->left

**With a clever argv[1]:
write a 4-byte word to an
arbitrary location in memory**



```
movl    $0xf8, (%esp)
call    0x8048364 <malloc@plt>
mov     %eax, 0x14(%esp)
movl    $0xf8, (%esp)
call    0x8048364 <malloc@plt>
mov     %eax, 0x18(%esp)
mov     0x14(%esp), %eax
mov     %eax, (%esp)
call    0x8048354 <free@plt>
mov     0x18(%esp), %eax
mov     %eax, (%esp)
call    0x8048354 <free@plt>
movl    $0x200, (%esp)
call    0x8048364 <malloc@plt>
mov     %eax, 0x1c(%esp)
mov     0xc(%ebp), %eax
add     $0x4, %eax
mov     (%eax), %eax
movl    $0x1ff, 0x8(%esp)
mov     %eax, 0x4(%esp)
mov     0x1c(%esp), %eax
mov     %eax, (%esp)
call    0x8048334 <strncpy@plt>
mov     0x18(%esp), %eax
mov     %eax, (%esp)
call    0x8048354 <free@plt>
mov     0x1c(%esp), %eax
mov     %eax, (%esp)
call    0x8048354 <free@plt>
leave
ret
```

What type of vulnerability might this be?

This is very simple example.
Manual analysis is very time
consuming.

Security analysts use a variety of
tools to augment manual analysis

Aiding analysts with tools

How can we automatically find the bug?

```
main( int argc, char* argv[] ) {
    char* b1;
    char* b2;
    char* b3;

    if( argc != 3 ) then return 0;
    if( argv[2] != 31337 )
        complicatedFunction();
    else {
        b1 = (char*)malloc(248);
        b2 = (char*)malloc(248);
        free(b1);
        free(b2);
        b3 = (char*)malloc(512);
        strncpy( b3, argv[1], 511 );
        free(b2);
        free(b3);
    }
}
```

Start with dynamic analysis: Fuzzing



“The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing, syntax testing or negative testing.”

Wikipedia

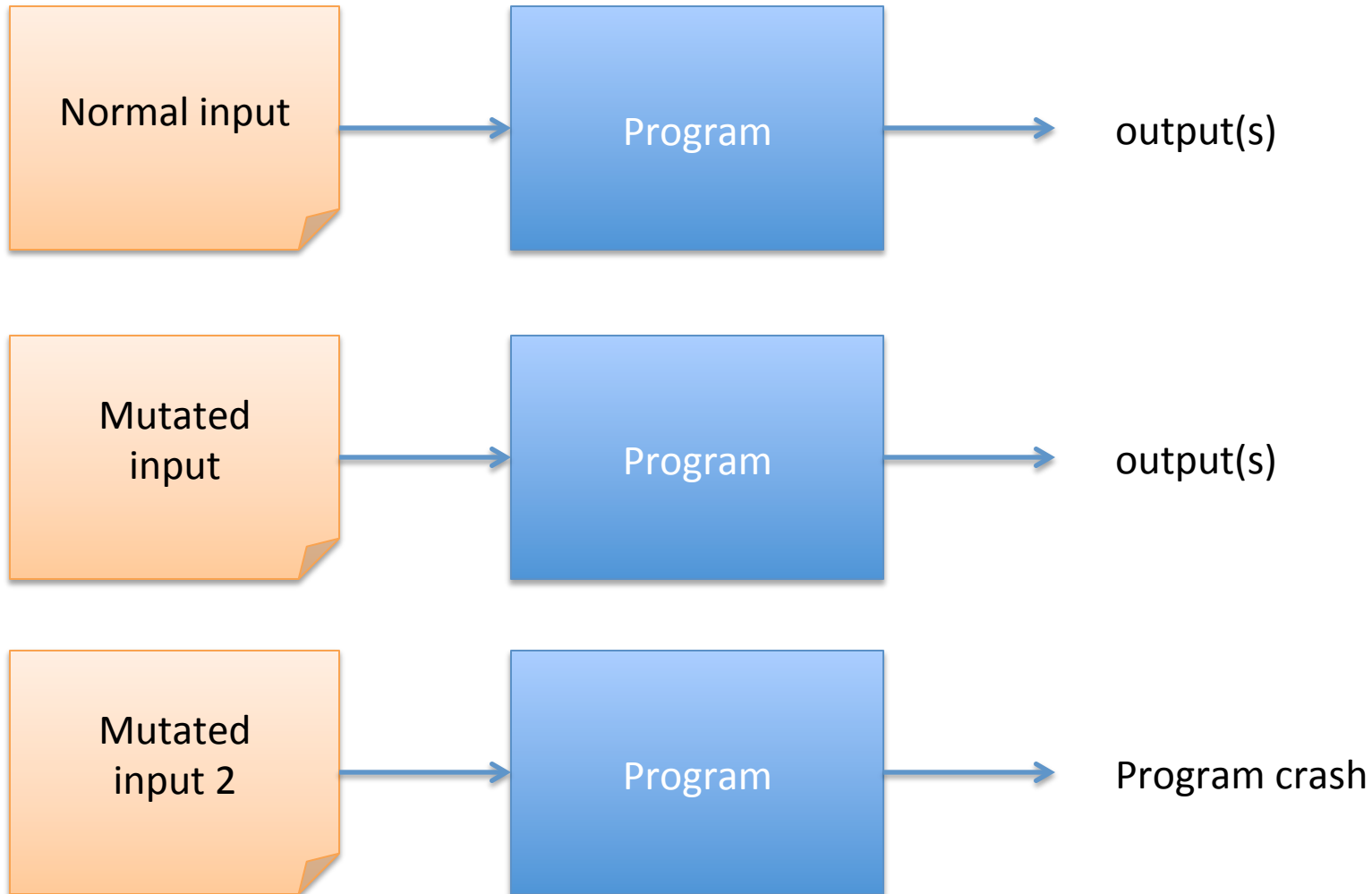
http://en.wikipedia.org/wiki/Fuzz_testing

Choose a bunch of inputs

See if they cause program to misbehave

Example of dynamic analysis

Black-box fuzz testing: the goal



Black-box fuzz testing

argv[1]="AAAA"
argv[2]=1

Program

argv[1] = random str
argv[2] =
random 32-bit int

Program

```
main( int argc, char* argv[] ) {  
    char* b1;  
    char* b2;  
    char* b3;  
  
    if( argc != 3 ) then return 0;  
    if( argv[2] != 31337 )  
        complicatedFunction();  
    else {  
        b1 = (char*)malloc(248);  
        b2 = (char*)malloc(248);  
        free(b1);  
        free(b2);  
        b3 = (char*)malloc(512);  
        strncpy( b3, argv[1], 511 );  
        free(b2);  
        free(b3);  
    }  
}
```

If x is 32 bits, then probability
of crashing is **at most what?**

$$1/2^{32}$$

Achieving code coverage can
be very difficult

Fuzzing is a lot about code coverage

- Code coverage defined in many ways
 - # of basic blocks reached
 - # of paths followed
 - # of conditionals followed
 - gcov is useful standard tool
- Mutation based
 - Start with known-good examples
 - Mutate them to new test cases
 - heuristics: increase string lengths (AAAAAAAAAA...)
 - randomly change items
- Generative
 - Start with specification of protocol, file format
 - Build test case files from it
 - Rarely used parts of spec

Manually refine fuzzing (example from Miller slides)

Multiplayer game

Fuzz for remote exploits

- Capture packets during normal use
- Replace some packet contents with random values
- Send to game, determine code coverage

Initial: 614 out of 36183 basic blocks

One big switch statement controlled by third byte of packet

Update fuzz rules to exhaust the values of this third byte

Improves coverage by 4x.

Repeat several times to improve coverage.

Heap overflow found.

From Wikipedia: *Freeciv*



Freeciv 2.1.0-beta3, with the SDL client

Example program analyzers

- Manual analysis (you are the analyzer!)
- Static analysis (do not execute program)
 - Scanners
 - Symbolic execution
 - Abstract representations
- Dynamic analysis (execute program)
 - Debugging
 - Fuzzers
 - Ptrace

Do you have source code?

Yes: lucky you

No: can still do things, but not as easily
(missing a lot of context about program)

Source code scanners

Look at source code, flag suspicious constructs

```
...  
strcpy( ptr1, ptr2 );  
...
```

Warning: Don't use strcpy

Simplest example: grep

Lint is early example

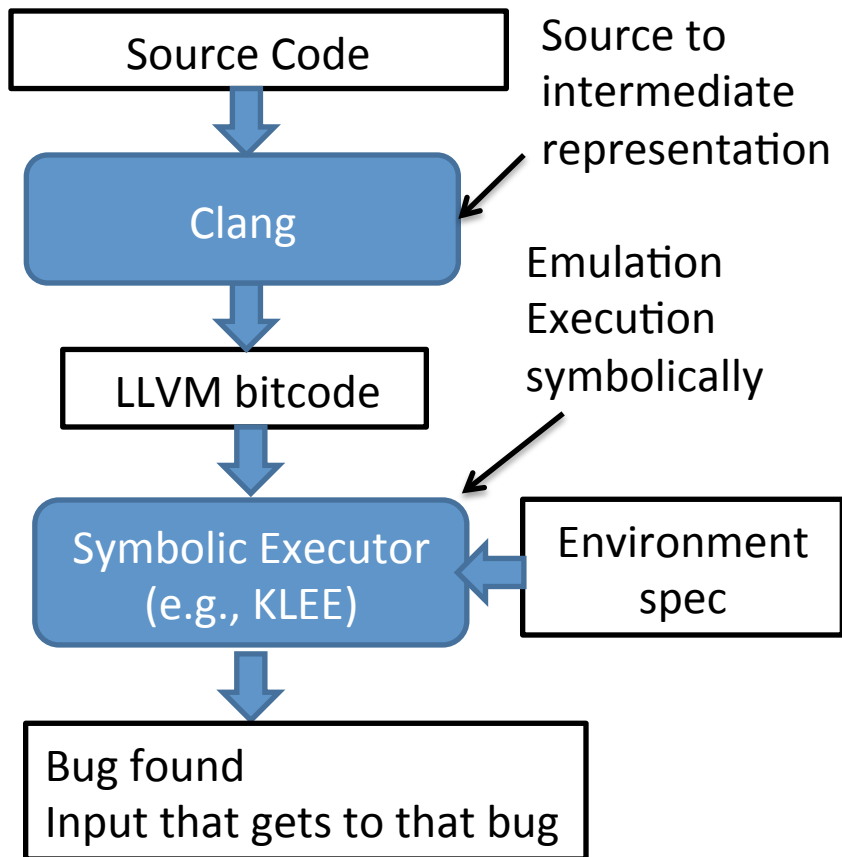
RATS (Rough auditing tool for security)

ITS4 (It's the Software Stupid Security Scanner)

Circa 1990's technology:

shouldn't work for reasonable modern codebases

Symbolic execution



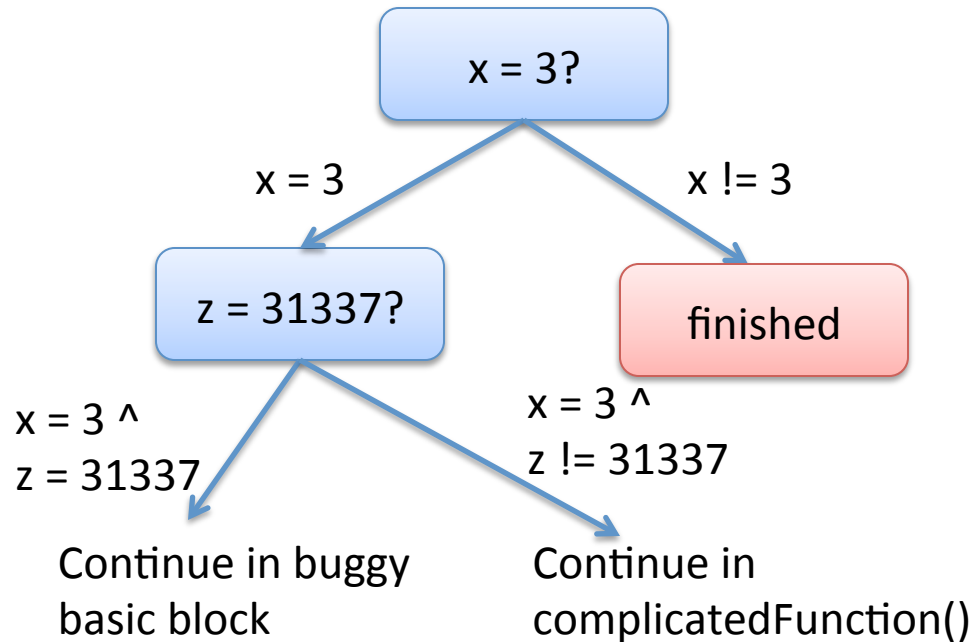
- Technique for statically analyzing code paths and finding inputs
- Associate to each input variable a special symbol
 - called symbolic variable
- Simulate execution symbolically
 - Update symbolic variable's value appropriately
 - Conditionals add constraints on possible values
- Cast constraints as satisfiability, and use SAT solver to find inputs

Symbolic execution

```
main( int argc, char* argv[] ) {  
    char* b1;  
    char* b2;  
    char* b3;  
  
    if( argc != 3 ) then return 0;  
    if( argv[2] != 31337 )  
        complicatedFunction();  
    else {  
        b1 = (char*)malloc(248);  
        b2 = (char*)malloc(248);  
        free(b1);  
        free(b2);  
        b3 = (char*)malloc(512);  
        strncpy( b3, argv[1], 511 );  
        free(b2);  
        free(b3);  
    }  
}
```

Initially:

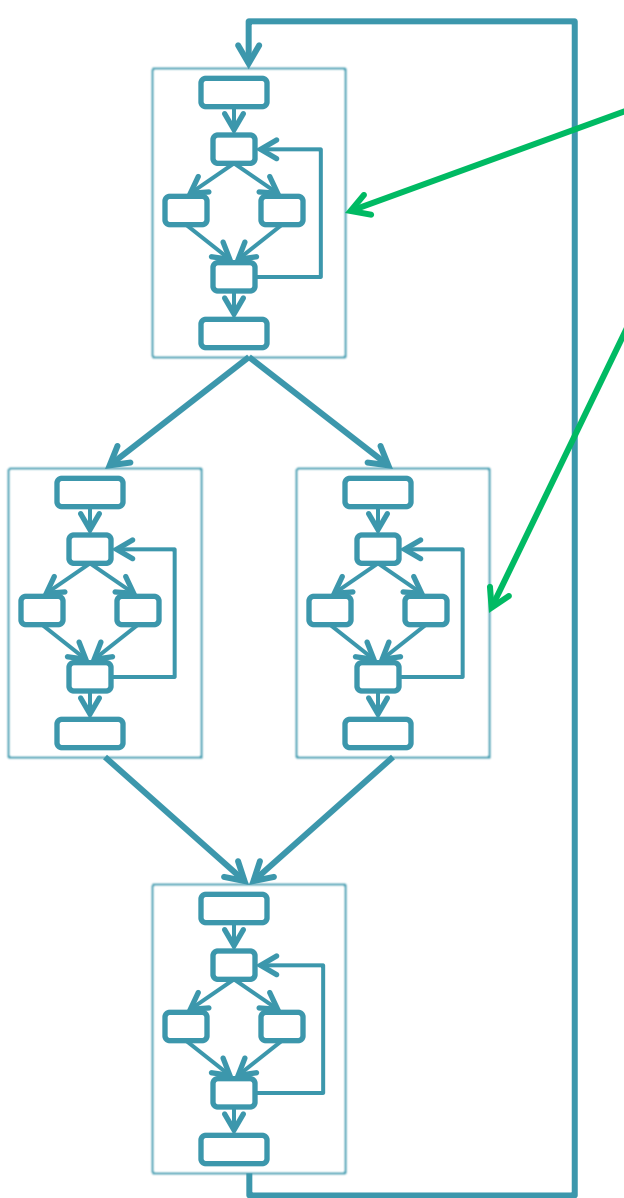
$argc = x$ (unconstrained int)
 $argv[2] = z$ (memory array)



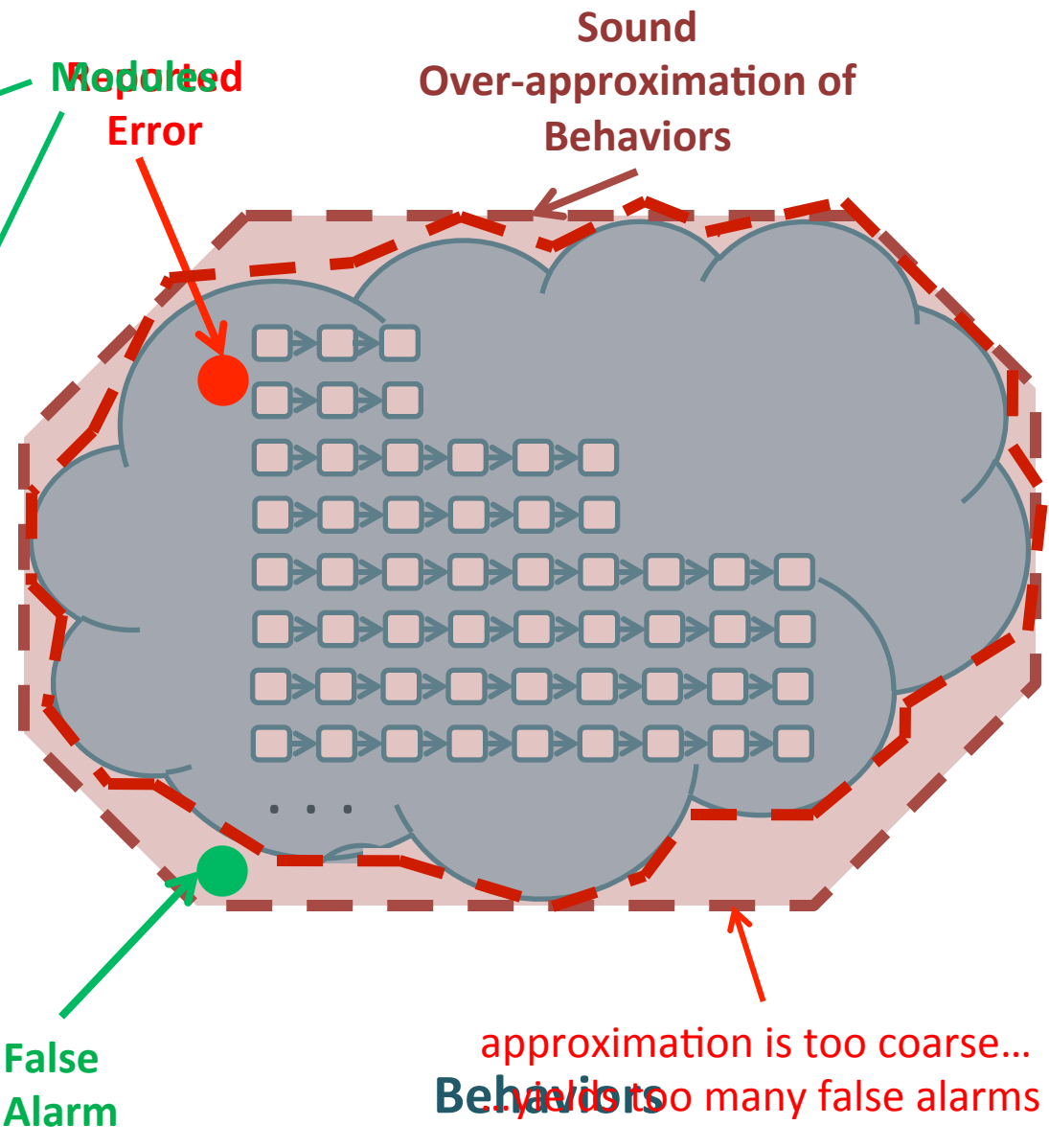
- Eventually emulation hits a double free
- Can trace back up path to determine what x, z must have been to hit this basic block

Symbolic execution challenges

- Can we complete analyses?
 - Yes, but only for very simple programs
 - Exponential # of paths to explore
- Path selection
 - Might get stuck in complicatedFunction()
- Encoding checks on symbolic states
 - Must include logic for double free check
 - Symbolic execution on binary more challenging (lose most memory semantics)



Software



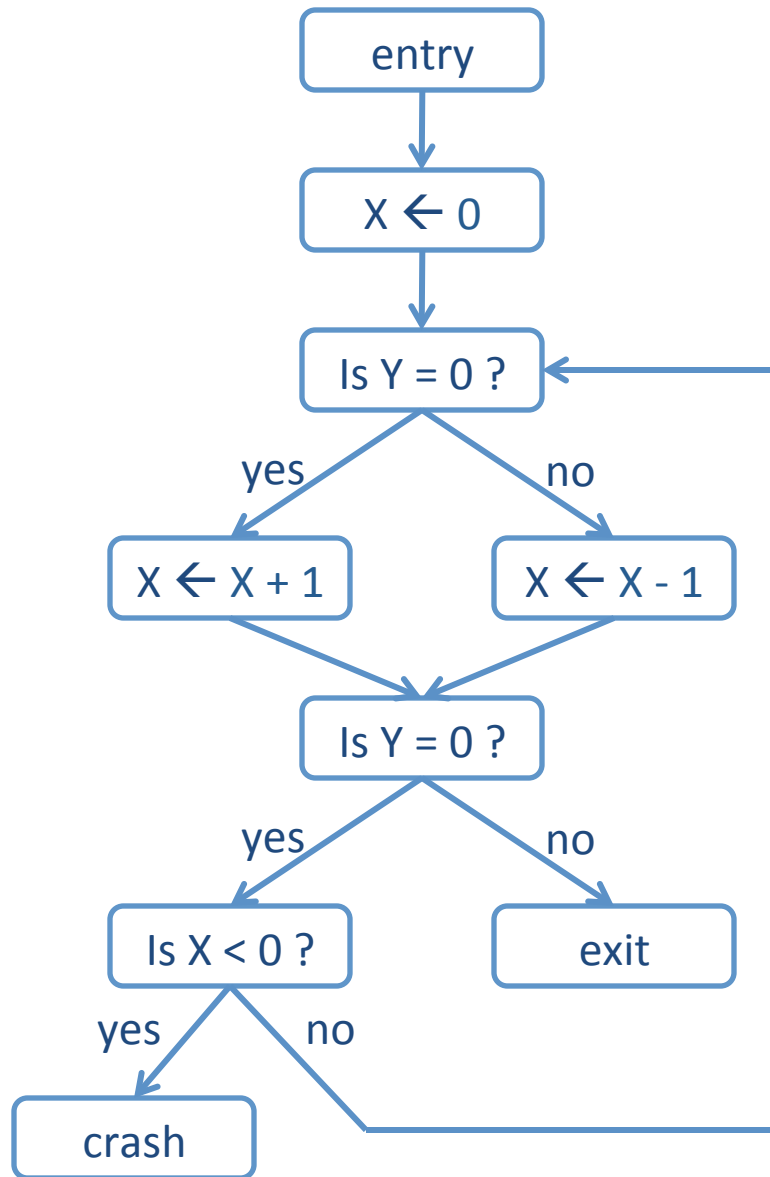
False Alarm

Modulated Error

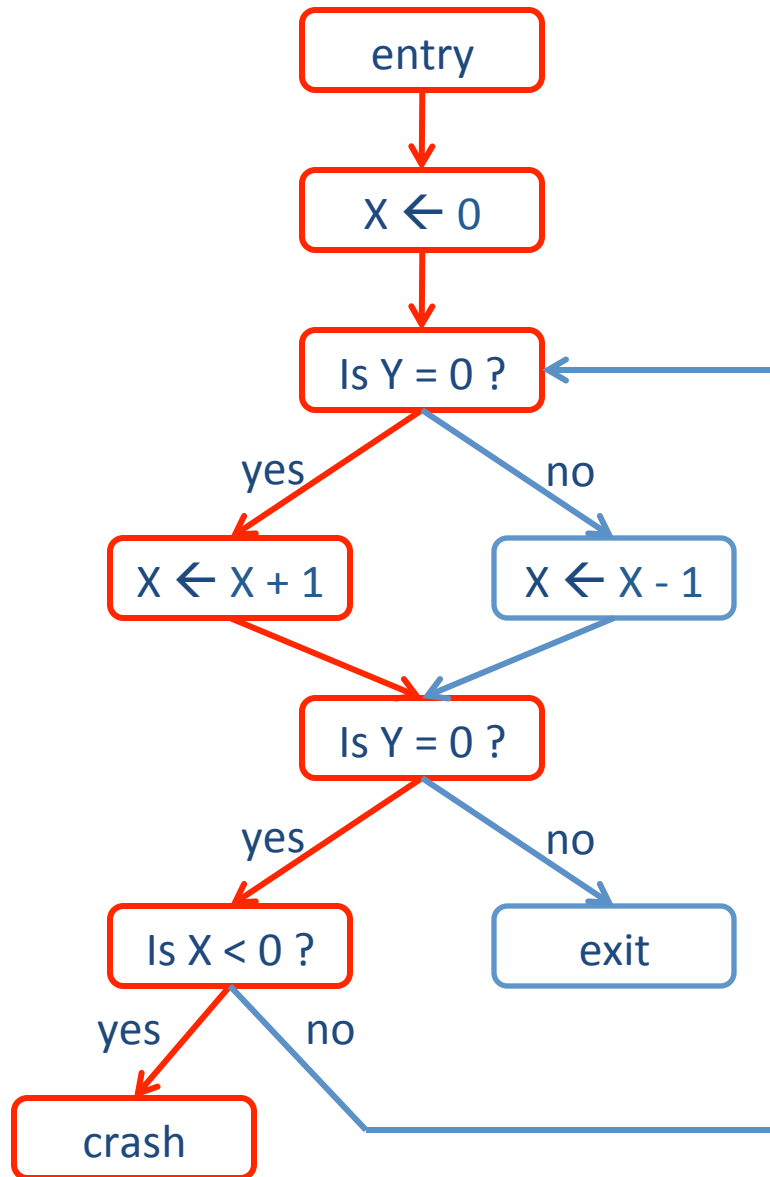
Sound Over-approximation of Behaviors

approximation is too coarse... Behaviors too many false alarms

Does this program ever crash?

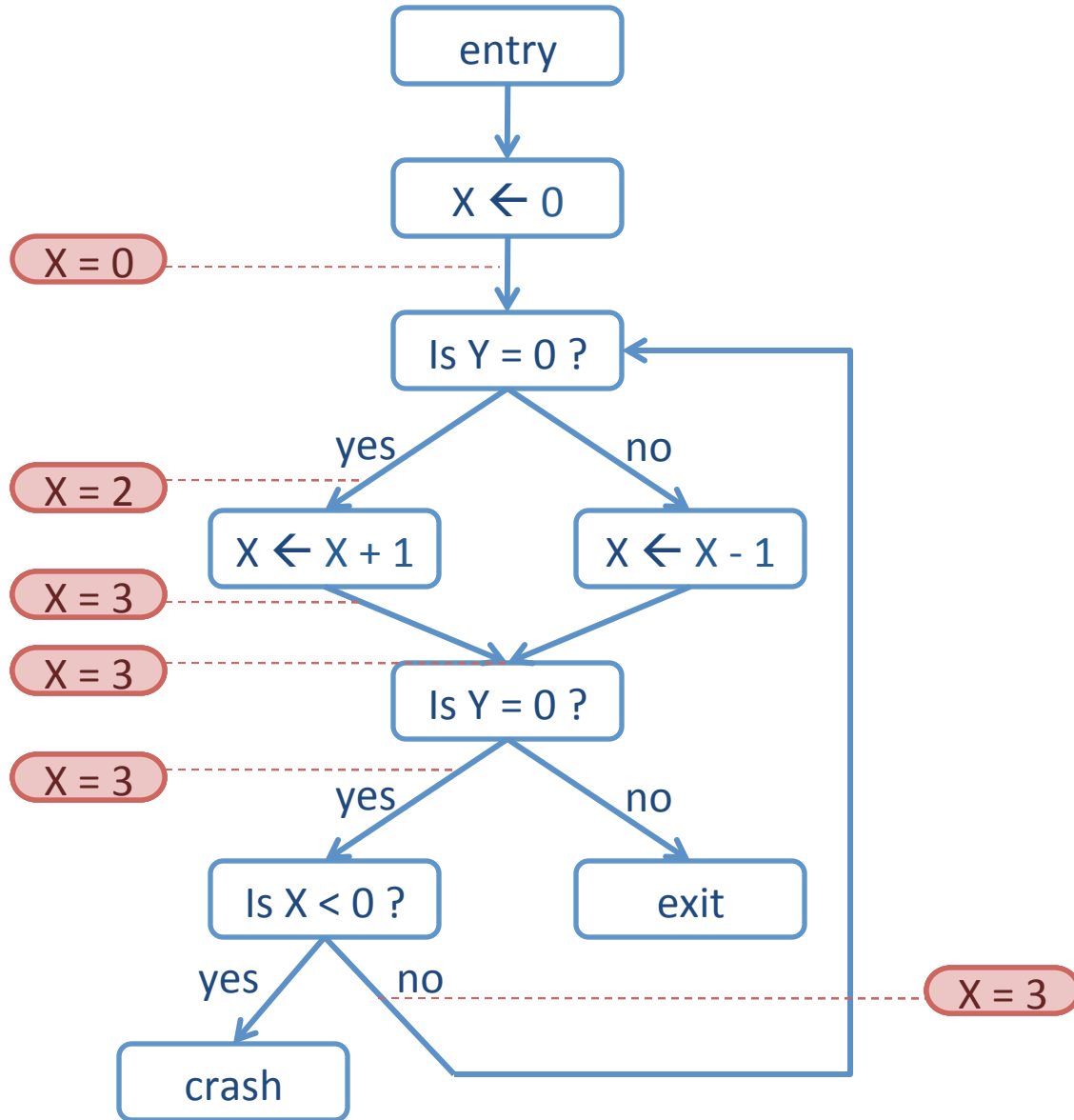


Does this program ever crash?



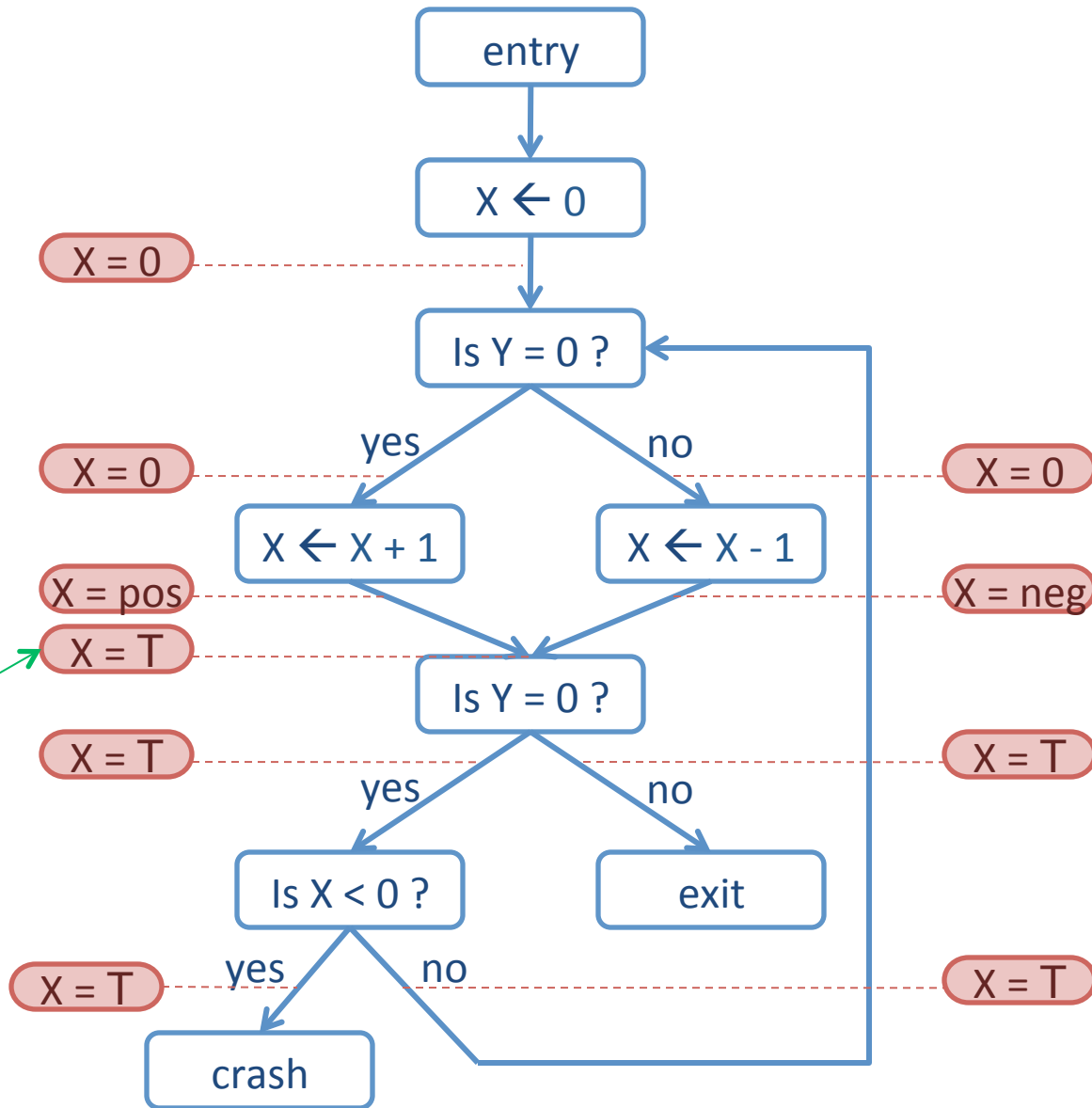
infeasible path!
... program will never crash

Try analyzing without approximating...



non-termination!
... therefore, need to approximate

Try analyzing with “signs” approximation...



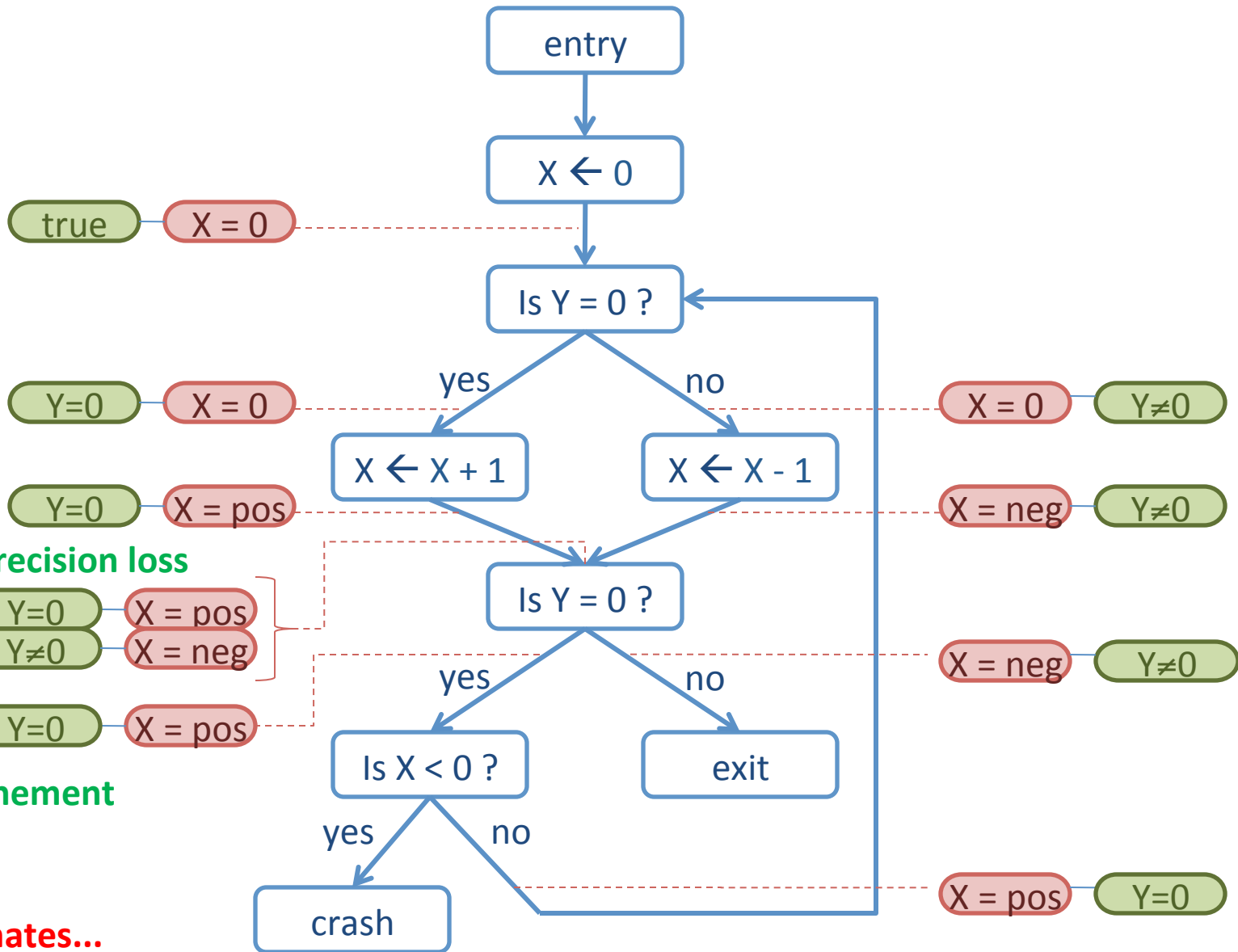
lost precision

terminates...

... but reports false alarm

... therefore, need more precision

Try analyzing with “path-sensitive signs” approximation...



no precision loss

Y=0 X = pos
Y≠0 X = neg

refinement

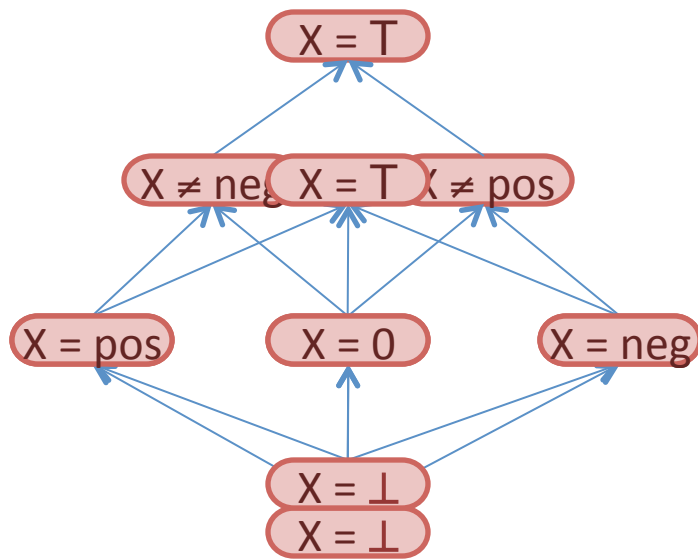
Y=0 X = pos

terminates...

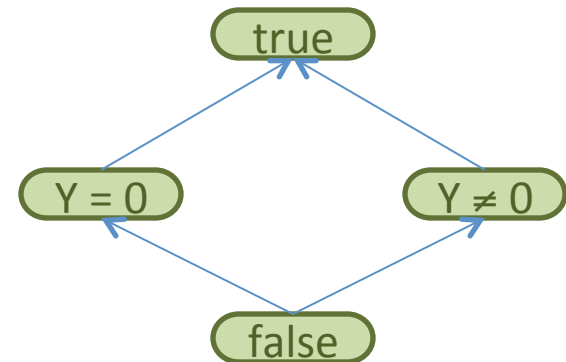
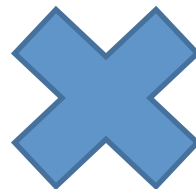
... no false alarm

... soundly proved never crashes

We represent program states using lattices; this specifies the possible values we assign to variables



sign lattice



Boolean formula lattice

Bug finding is a big business

- Grammatech (Prof Reps here at Wisconsin)
- Coverity (Stanford startup)
- Fortify
- many, many others...

Example program analyzers

- Manual analysis (you are the analyzer!)
- Static analysis (do not execute program)
 - Scanners
 - Abstract interpretation
 - Symbolic execution
- Dynamic analysis (execute program)
 - Debugging
 - Fuzzers
 - Ptrace

Do you have source code?

Yes: lucky you

No: can still do things, but not as easily
(missing a lot of context about program)

Tales in insecurity...

"The most critical servers contain malicious software that can normally be detected by anti-virus software," it says. "The separation of critical components was not functioning or was not in place. We have strong indications that the CA-servers, although physically very securely placed in a tempest proof environment, were accessible over the network from the management LAN."

All CA servers were members of one Windows domain and all accessible with one user/password combination. Moreover, the used password was simple and susceptible to brute-force attacks.

<http://www.net-security.org/secworld.php?id=11570>

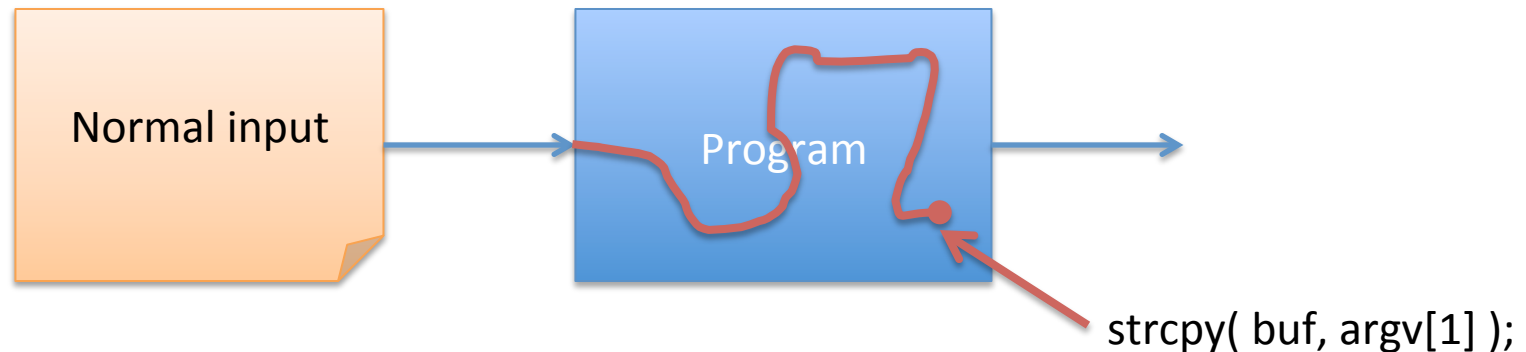
DigiNotar

Taint tracking

Track information flow from user input to it's use

Can be either static or dynamic

Useful to augment manual testing



White-box fuzz testing

- Start with real input and do static analysis
 - Symbolic execution of program
 - Gather constraints (control flow) along way
 - Systematically negate constraints backwards
 - Eventually this yields a new input
- Repeat

Godefroid, Levin, Molnar. “Automated Whitebox Fuzz Testing”

Symbolic execution + fuzzing

```
void top(char input[4]) {  
    int cnt=0;  
    if (input[0] == 'b') cnt++;  
    if (input[1] == 'a') cnt++;  
    if (input[2] == 'd') cnt++;  
    if (input[3] == '!') cnt++;  
    if (cnt >= 3) abort(); // error  
}
```

Example from Godefroid et al.

Start with some input.

Run program for real & symbolically

Say input = "good"

$i_0 \neq 'b'$

$i_1 \neq 'a'$

$i_2 \neq 'd'$

$i_3 \neq '!$

i_0, i_1, i_2, i_3
are all
symbolic
variables

This gives set of constraints on input

Negate them one at a time to generate a
new input that explores new path

Example

$i_0 \neq 'b'$ and $i_1 \neq 'a'$ and $i_2 \neq 'd'$ and $i_3 = '!$
input would be "goo!"

Repeat with new input

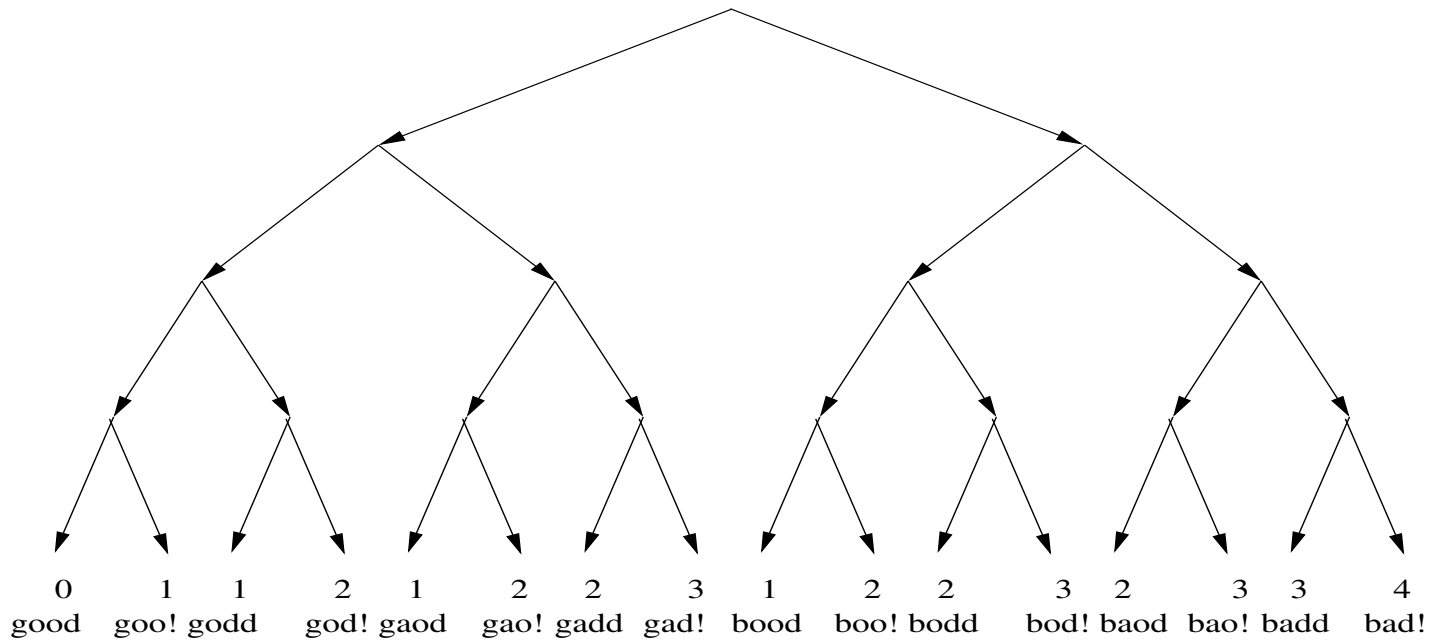
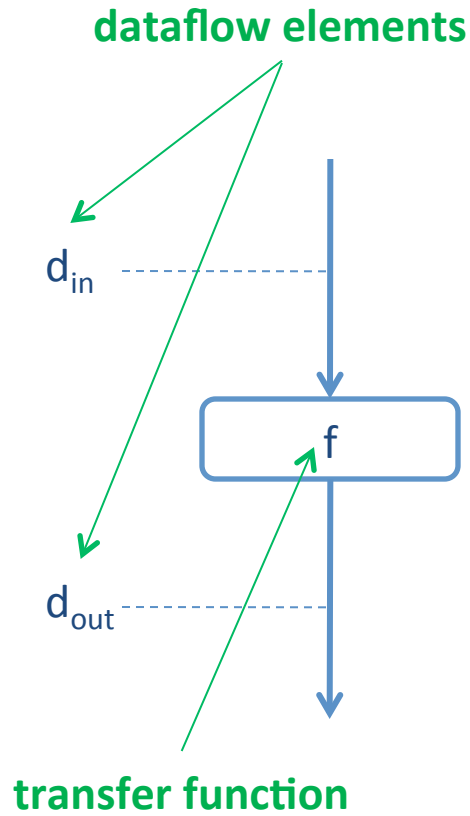
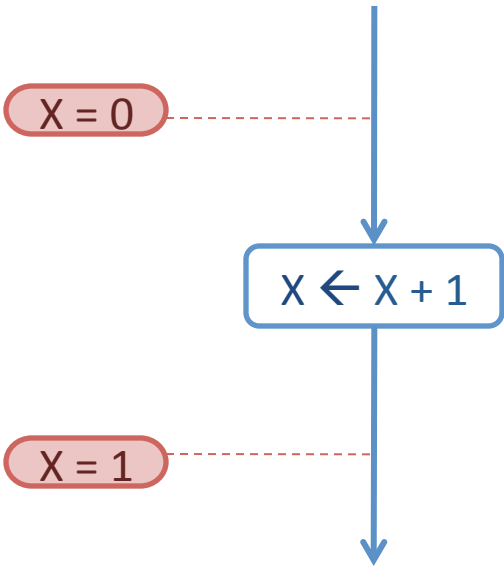


Figure 2. Search space for the example of Figure 1 with the value of the variable `cnt` at the end of each run and the corresponding input string.

Example from Godefroid et al.

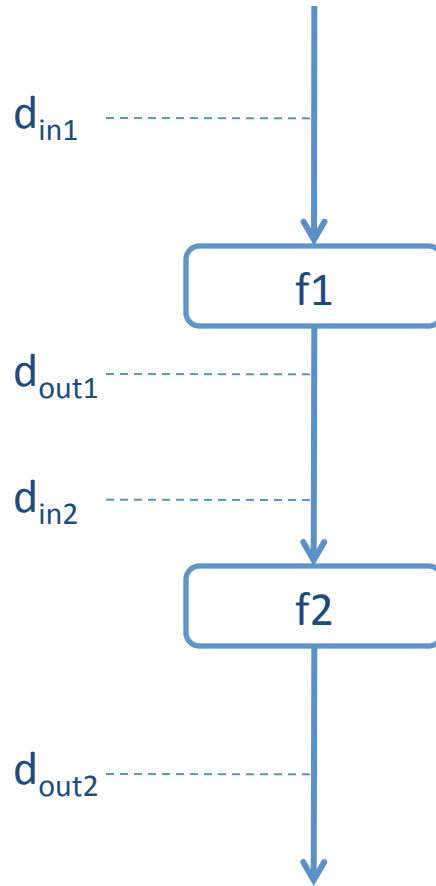
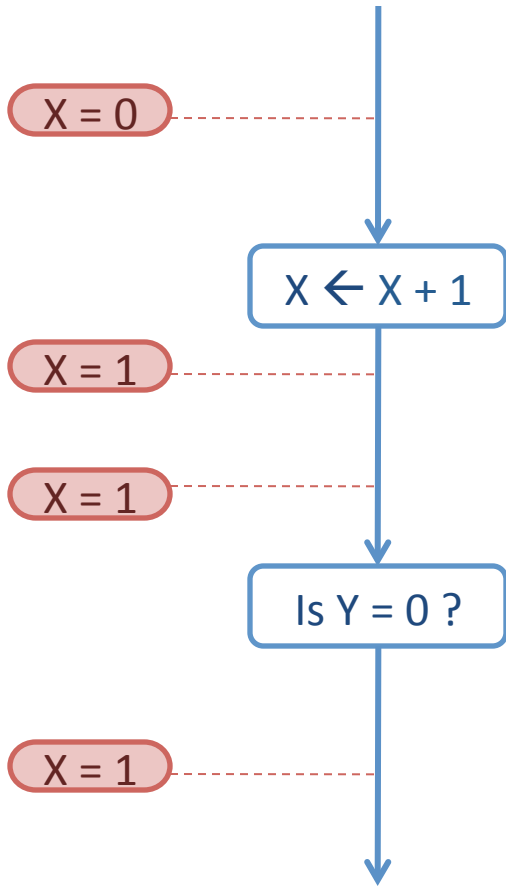
Larger programs have too many paths to explore so they specify various heuristics

In-use at Microsoft



$$d_{out} = f(d_{in})$$

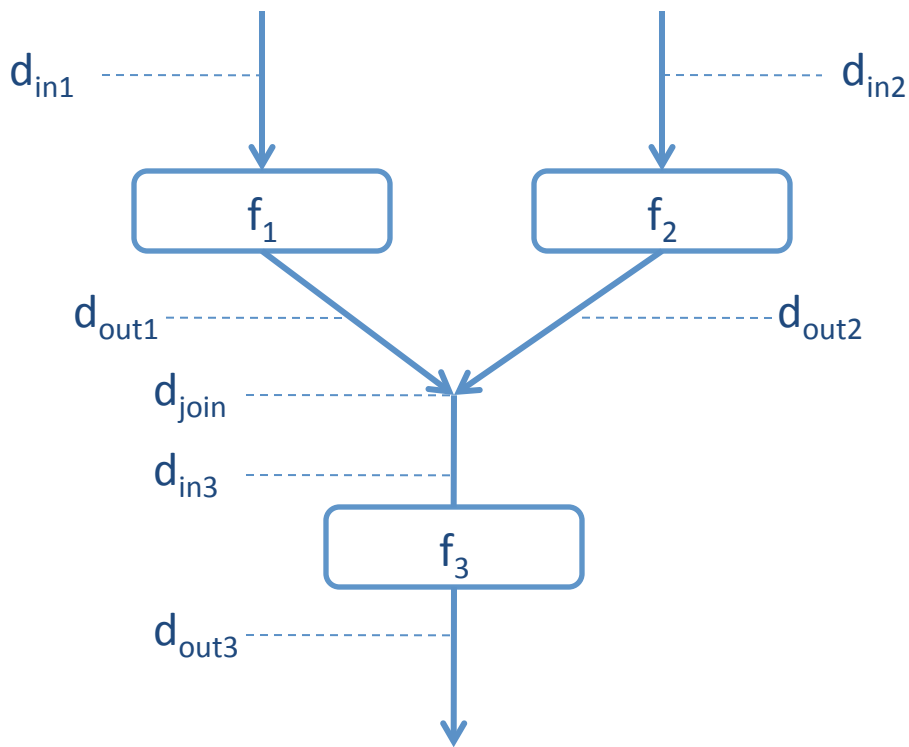
A green arrow points from the text "dataflow equation" below to the f in the equation above.



$$d_{out1} = f_1(d_{in1})$$

$$d_{out1} = d_{in2}$$

$$d_{out2} = f_2(d_{in2})$$



$$d_{out1} = f_1(d_{in1})$$

$$d_{out2} = f_2(d_{in2})$$

$$d_{join} = d_{out1} \sqcup d_{out2}$$

$$d_{join} = d_{in3}$$

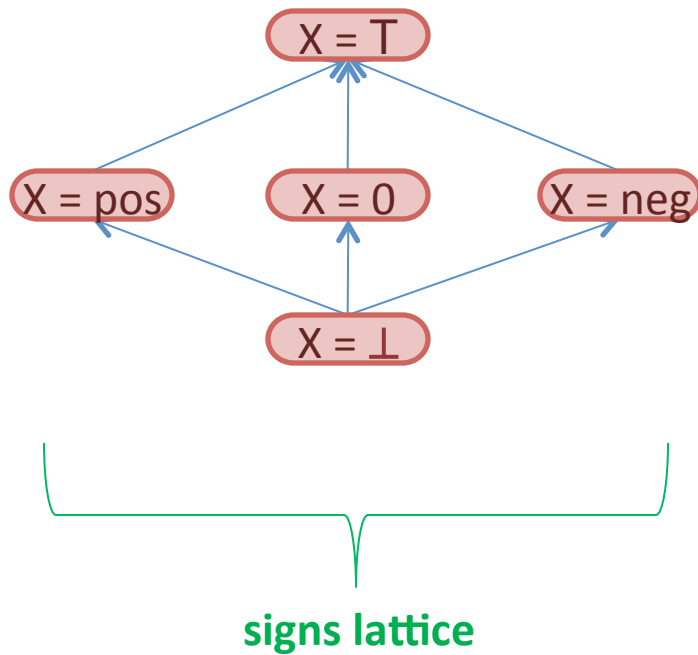
$$d_{out3} = f_3(d_{in3})$$

least upper bound operator

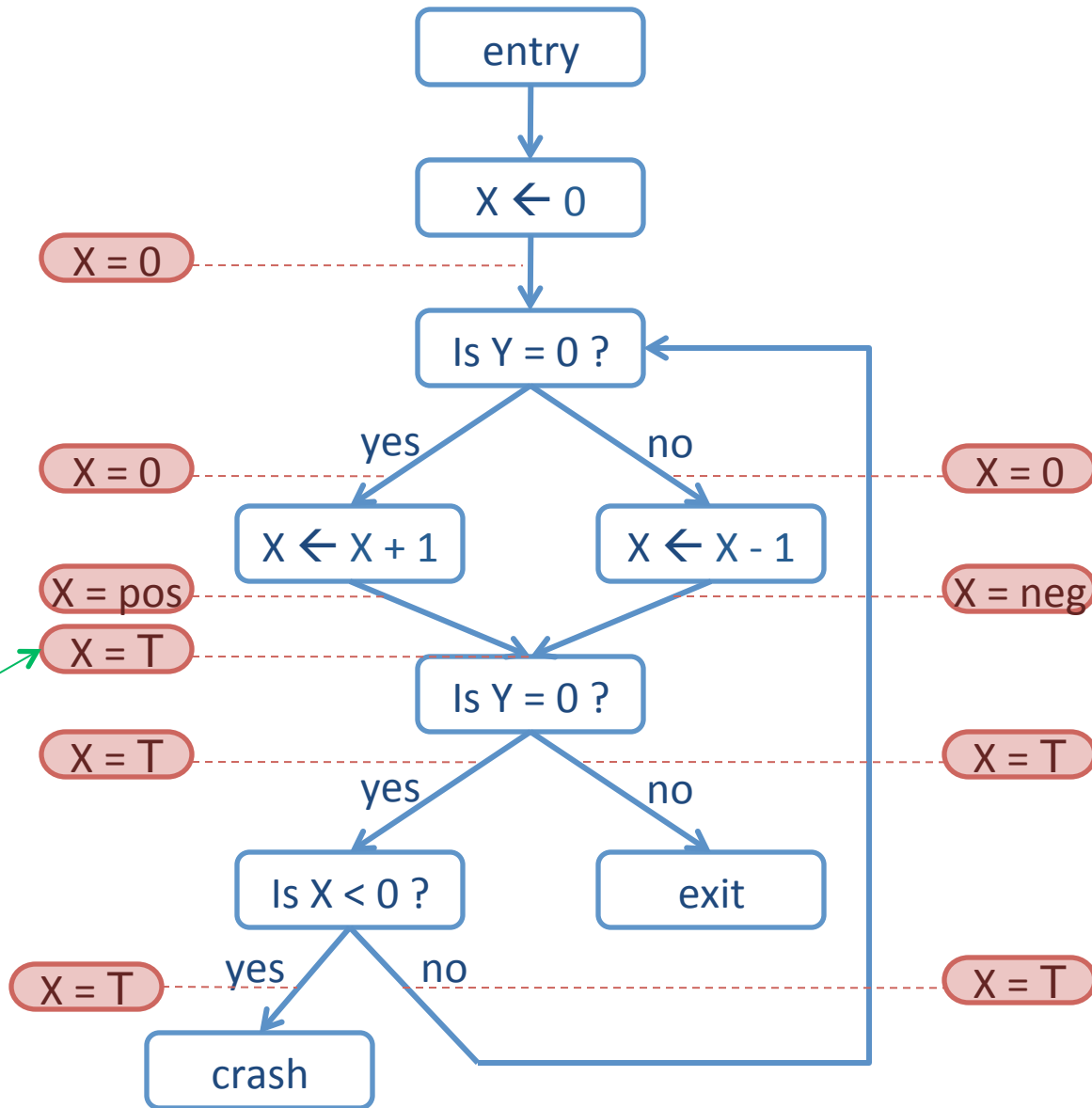
Example: union of possible values

What is the space of dataflow elements, Δ ?
 What is the least upper bound operator, \sqcup ?

We give a lattice to specify the possible values we assign to symbolic variables



Try analyzing with “signs” approximation...

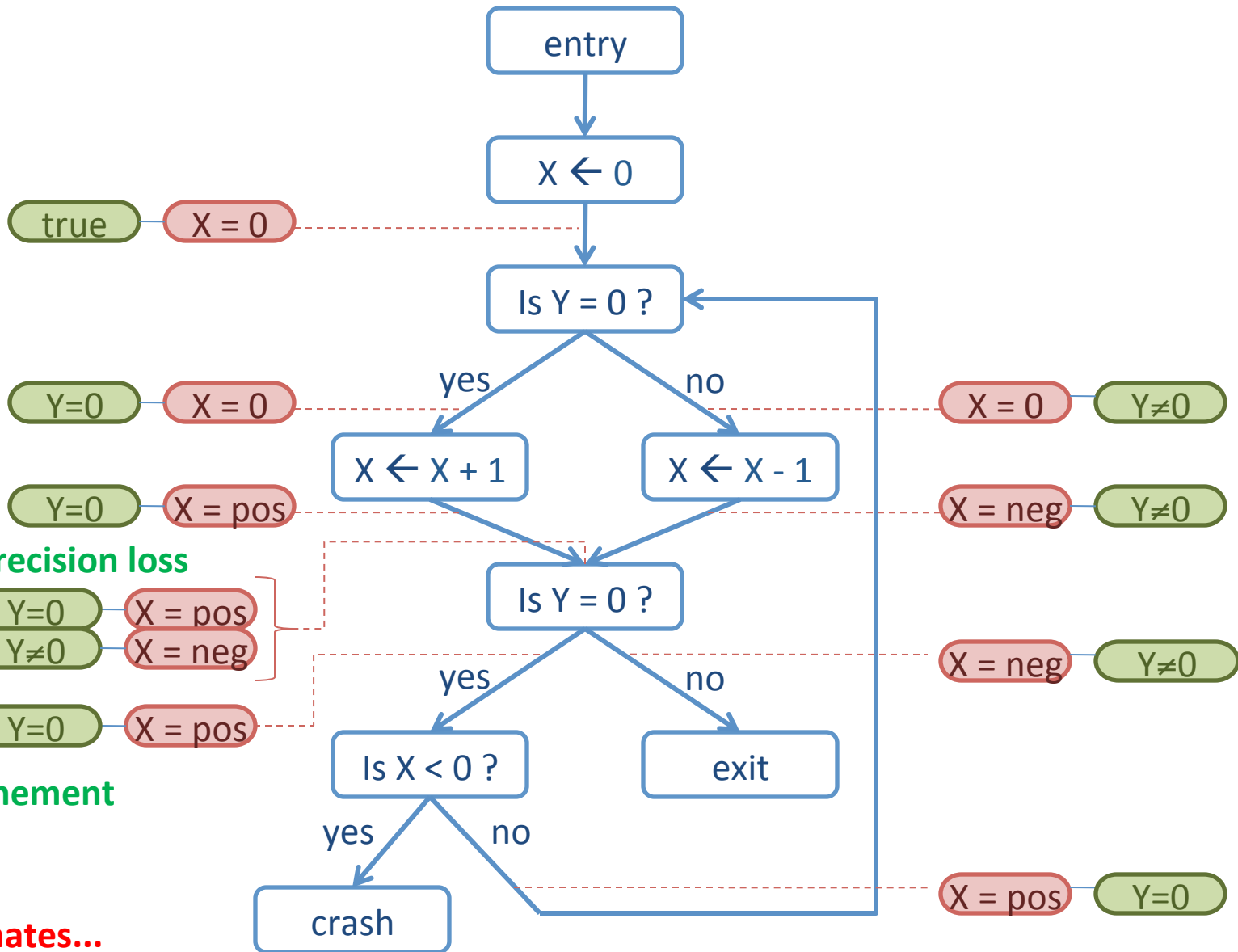


terminates...

... but reports false alarm

... therefore, need more precision

Try analyzing with “path-sensitive signs” approximation...



terminates...

... no false alarm

... soundly proved never crashes