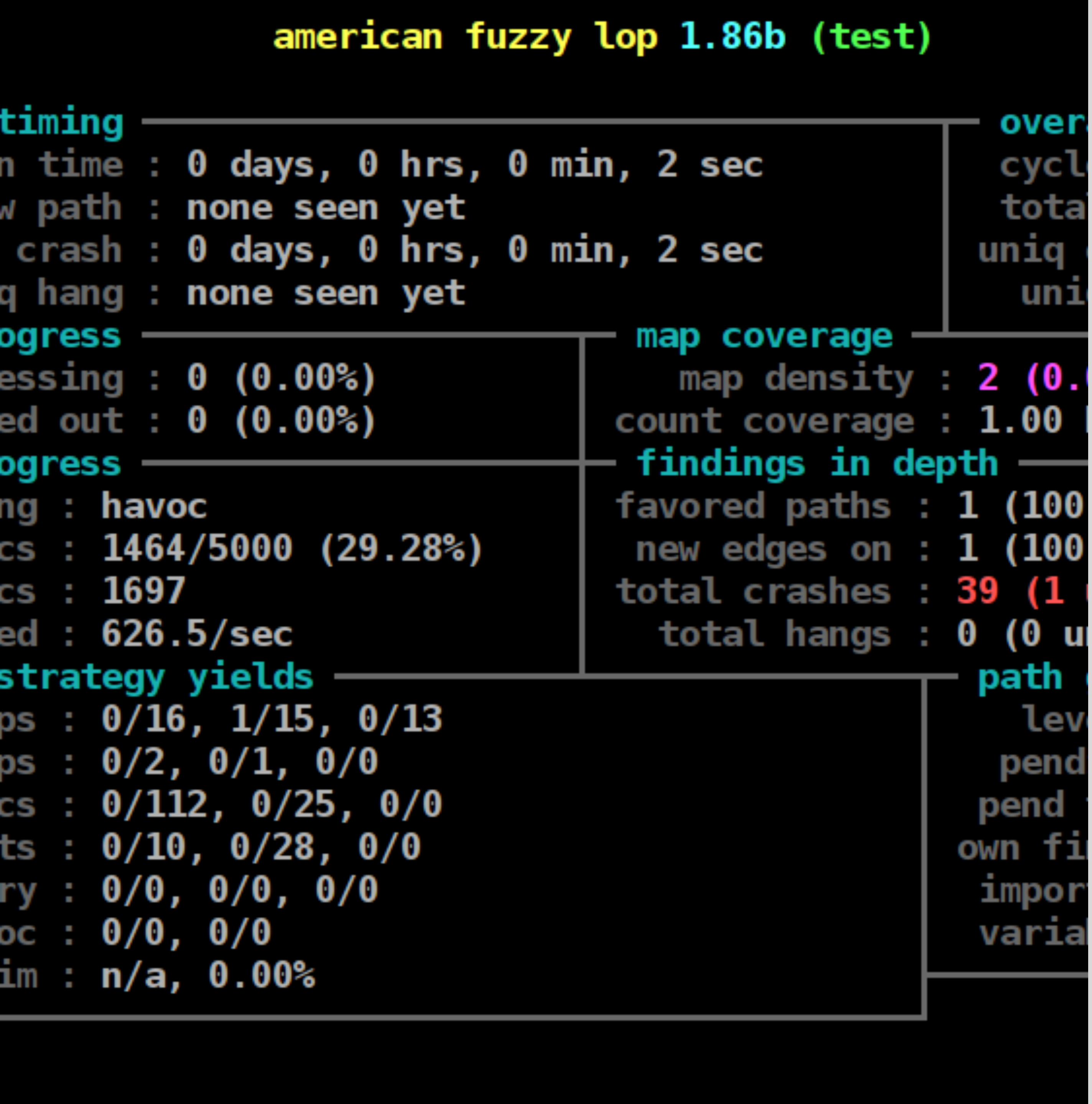


PROGRAM ANALYSIS AND FINDING VULNERABILITIES

VITALY SHMATIKOV

Better Languages

- (More) type-safe languages prevent some vulnerabilities by design
  - "A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data."
  - Traditionally less performance
- New generation of safer high-performance languages:
  - Rust (Mozilla), Swift (Apple), Go (Google)
- Efforts to improve security of unsafe languages
- Safe pointer libraries in C / C++
- Coding standards, defensive programming, unit testing

Better Software Engineering

- Organize software lifecycle around security
- Require use of organizational and software tools to improve security outcomes
- Microsoft security development lifecycle (SDL):

Training	Manage risk of third-party components
Design security requirements	Use approved tools
Metrics & compliance reporting	Static analysis security testing
Threat modeling	Dynamic analysis security testing
Establish design requirements	Penetration testing
Define & use crypto standards	Incident response

https://www.microsoft.com/en-us/securityengineering/sdl/practices

Most Software Very Complex

Linux kernel v4.1: ~19.5 million lines of code, 14,000 contributors

Apache HTTP server: ~1.5 million lines of code, 125 contributors

OpenSSL: 608K lines of code, 572 contributors

Remember Heartbleed?

OpenSSL implements TLS, used in Apache and Nginx

March 2014: researchers discover vulnerability in the OpenSSL implementation of TLS heartbeat

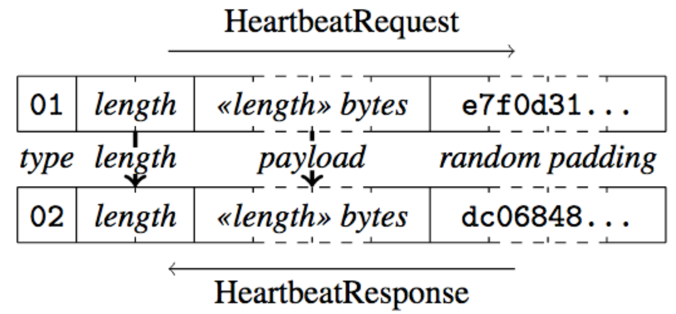
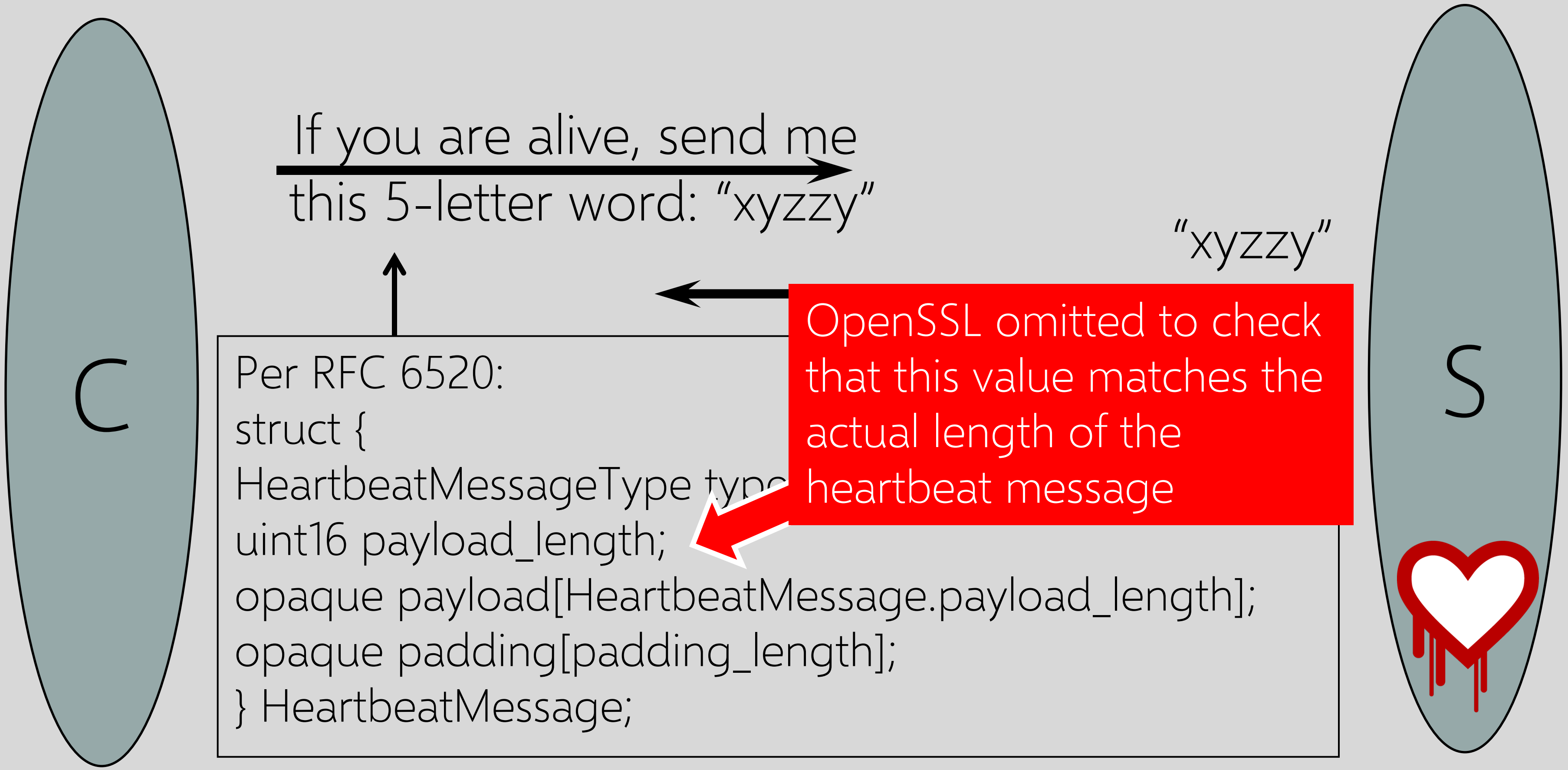


Figure 1: **Heartbeat Protocol.** Heartbeat requests include user data and random padding. The receiving peer responds by echoing back the data in the initial request along with its own padding.

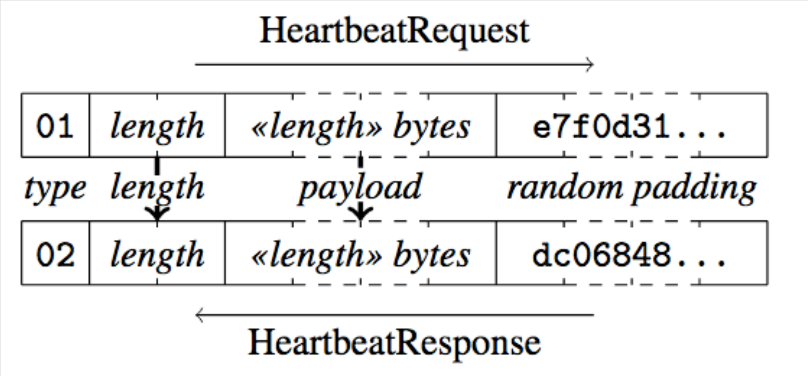
[Durumeric et al. 2014]

TLS Heartbeat

A way to keep TLS connection alive
without constantly transferring data



Heartbleed



<u>Buffer overread</u> vulnerability
Copy up to almost 2^{16} bytes of data from memory

```
dtls1_process_heartbeat(SSL *s)
  {
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  pl = p;

  if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
      &s->s3->rrec.data[0], s->s3->rrec.length,
      s, s->msg_callback_arg);

  if (hbtype == TLS1_HB_REQUEST)
    {
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;

    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    bp += payload;
```

Heartbleed Chronology

"I was doing laborious auditing of OpenSSL, going through the [Secure Sockets Layer] stack line by line"

Date	Event
03/21	Neel Mehta of Google discovers Heartbleed
03/21	Google patches OpenSSL on their servers
03/31	CloudFlare is privately notified and patches
04/01	Google notifies the OpenSSL core team
04/02	Codenomicon independently discovers Heartbleed
04/03	Codenomicon informs NCSC-FI
04/04	Akamai is privately notified and patches
04/05	Codenomicon purchases the `heartbleed.com` domain
04/06	OpenSSL notifies several Linux distributions
04/07	NCSC-FI notifies OpenSSL core team
04/07	OpenSSL releases version 1.0.1g and a security advisory
04/07	CloudFlare and Codenomicon disclose on Twitter
04/08	Al-Bassam scans the Alexa Top 10,000
04/09	University of Michigan begins scanning

[Durumeric et al. 2014]

Scanning for Heartbleed

Internet scanning to determine vulnerability:
Send heartbeat request with zero length (indicates vulnerable system)

Web Server	Alexa Sites	Heartbeat Ext.	Vulnerable
Apache	451,270 (47.3%)	95,217 (58.4%)	28,548 (64.4%)
Nginx	182,379 (19.1%)	46,450 (28.5%)	11,185 (25.2%)
Microsoft IIS	96,259 (10.1%)	637 (0.4%)	195 (0.4%)
Litespeed	17,597 (1.8%)	6,838 (4.2%)	1,601 (3.6%)
Other	76,817 (8.1%)	5,383 (3.3%)	962 (2.2%)
Unknown	129,006 (13.5%)	8,545 (5.2%)	1,833 (4.1%)

[Durumeric et al. 2014]

Scanning for Heartbleed

Internet scanning to determine vulnerability:
Send heartbeat request with zero length (indicates vulnerable system)

Site	Vuln.	Site	Vuln.	Site	Vuln.
Google	Yes	Bing	No	Wordpress	Yes
Facebook	No	Pinterest	Yes	Huff. Post	?
Youtube	Yes	Blogspot	Yes	ESPN	?
Yahoo	Yes	Go.com	?	Reddit	Yes
Amazon	No	Live	No	Netflix	Yes
Wikipedia	Yes	CNN	?	MSN.com	No
LinkedIn	No	Instagram	Yes	Weather.com	?
eBay	No	Paypal	No	IMDB	No
Twitter	No	Tumblr	Yes	Apple	No
Craigslist	?	Imgur	Yes	Yelp	?

[Durumeric et al. 2014]

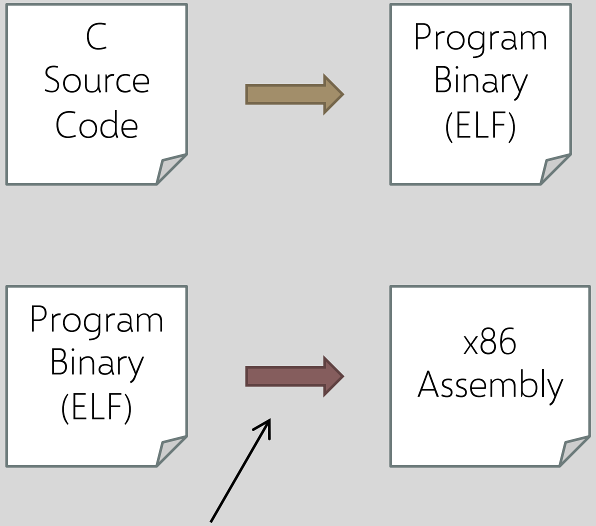
Disassembly and Decompiling

Heartbleed discovered by direct inspection of open-source C code

What if you only have the binary?

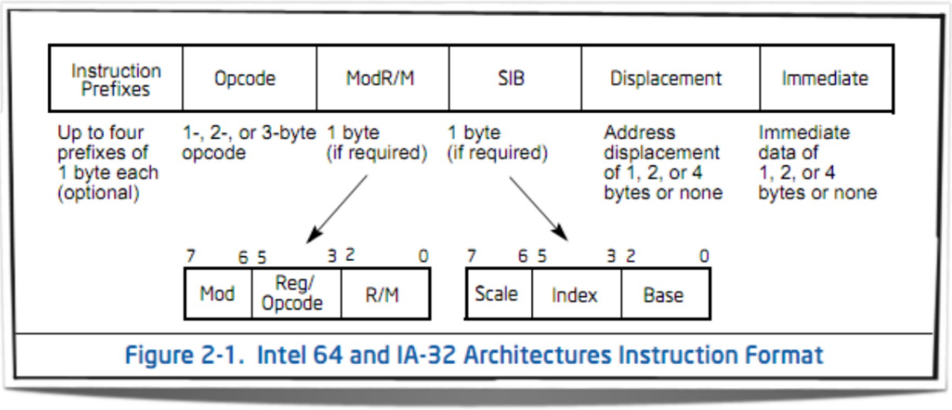
Normal compilation process

What if we start with binary?



Disassembler
(gdb, IDA Pro, OllyDebug)

Disassembly



```
Dump of assembler code for function main:
0x08048434 <main+0>:    push   %ebp
0x08048435 <main+1>:    mov    %esp,%ebp
0x08048437 <main+3>:    and    $0xfffffff0,%esp
0x0804843a <main+6>:    sub    $0x20,%esp
0x0804843d <main+9>:    movl   $0xf8,(%esp)
0x08048444 <main+16>:   call   0x8048364 <malloc@plt>
0x08048449 <main+21>:   mov    %eax,0x14(%esp)
0x0804844d <main+25>:   movl   $0xf8,(%esp)
0x08048454 <main+32>:   call   0x8048364 <malloc@plt>
0x08048459 <main+37>:   mov    %eax,0x18(%esp)
0x0804845d <main+41>:   mov    0x14(%esp),%eax
0x08048461 <main+45>:   mov    %eax,(%esp)
0x08048464 <main+48>:   call   0x8048354 <free@plt>
0x08048469 <main+53>:   mov    0x18(%esp),%eax
0x0804846d <main+57>:   mov    %eax,(%esp)
0x08048470 <main+60>:   call   0x8048354 <free@plt>
0x08048475 <main+65>:   movl   $0x200,(%esp)
0x0804847c <main+72>:   call   0x8048364 <malloc@plt>
0x08048481 <main+77>:   mov    %eax,0x1c(%esp)
0x08048485 <main+81
0x08048488 <main+84
0x0804848b <main+87
0x0804848d <main+89>:
0x08048495 <main+97>:
0x08048499 <main+101>:
0x0804849d <main+105>:
0x080484a0 <main+108>:
0x080484a5 <main+113>:
0x080484a9 <main+117>:
0x080484ac <main+120>:  call   0x8048354 <free@plt>
0x080484b1 <main+125>:  mov    0x1c(%esp),%eax
0x080484b5 <main+129>:  mov    %eax,(%esp)
0x080484b8 <main+132>:  call   0x8048354 <free@plt>
0x080484bd <main+137>:  leave
0x080484be <main+138>:  ret
End of assembler dump.
(gdb) _
```

Double-free vulnerability

Exploit can trick heap management software into writing adversary-controlled value to adversary-controlled address

What type of vulnerability might this be?

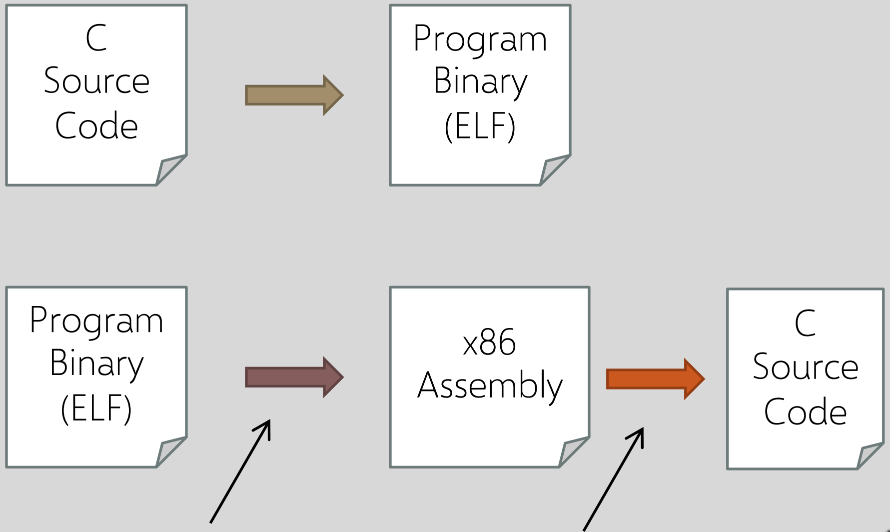
```
main(  int argc, char* argv[] ) {
  char* b1;
  char* b2;
  char* b3;

  if( argc != 3 ) then return 0;
  if( atoi(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);
   }
}
```

Disassembly and Decompiling

Normal compilation process

What if we start with binary?

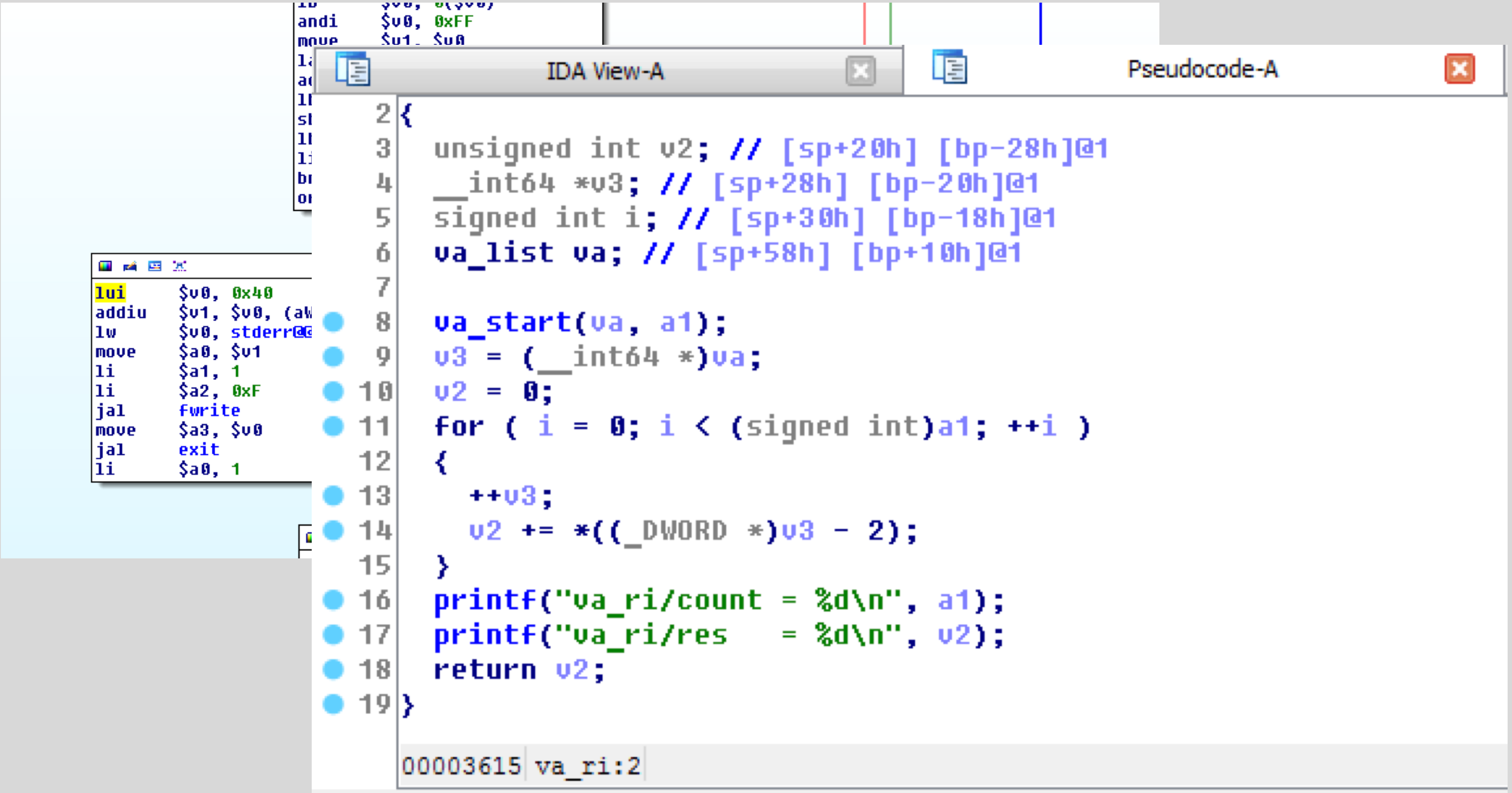


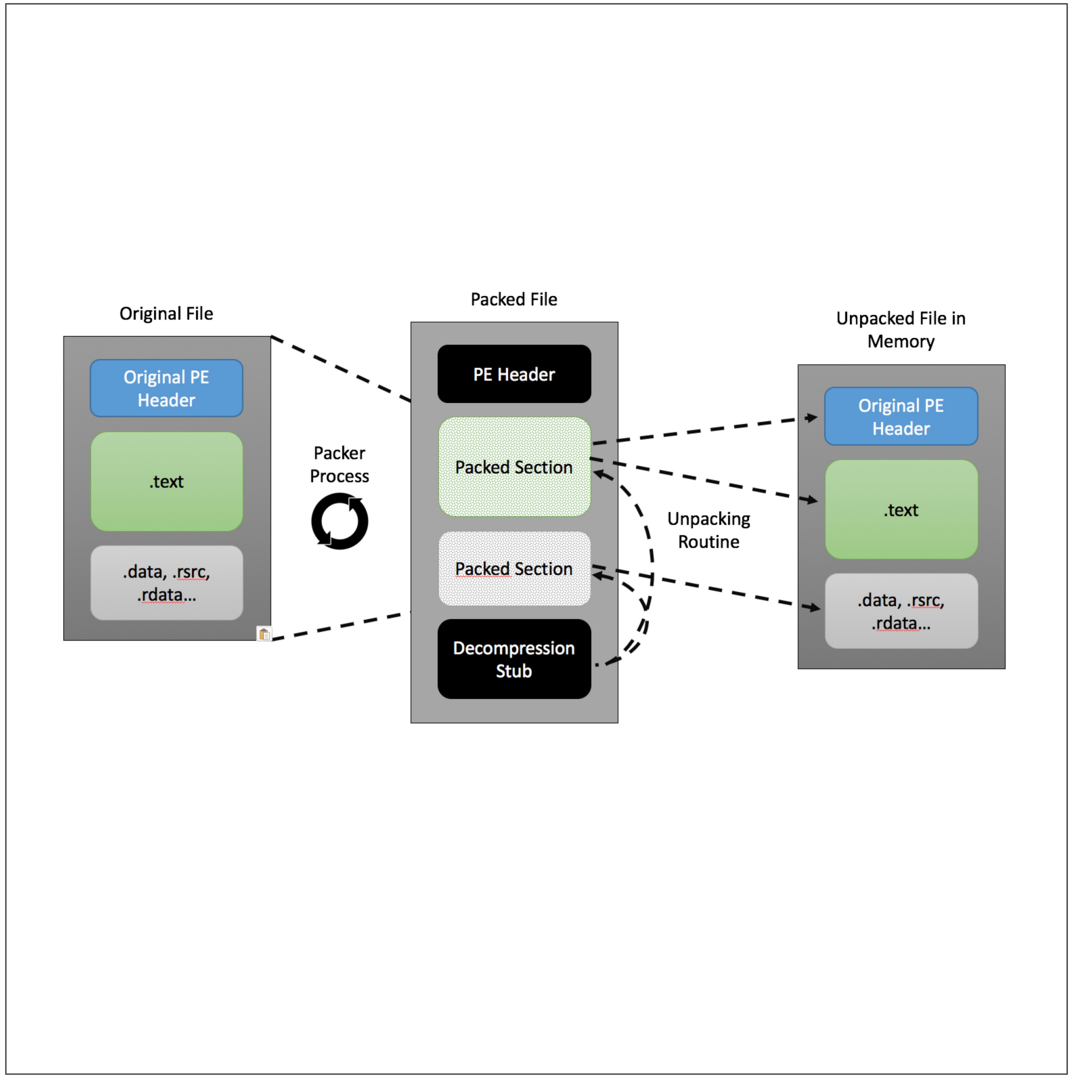
Disassembler
(gdb, IDA Pro, OllyDebug)

Decompiler
(IDA Pro has one)

Very complex, usually poor results

Decompilation





Packing

Packing hides the real code of a program through one or more layers of compression/encryption

At run-time the unpacking routine restores the original code in memory and then executes it

Vulnerability Discovery

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

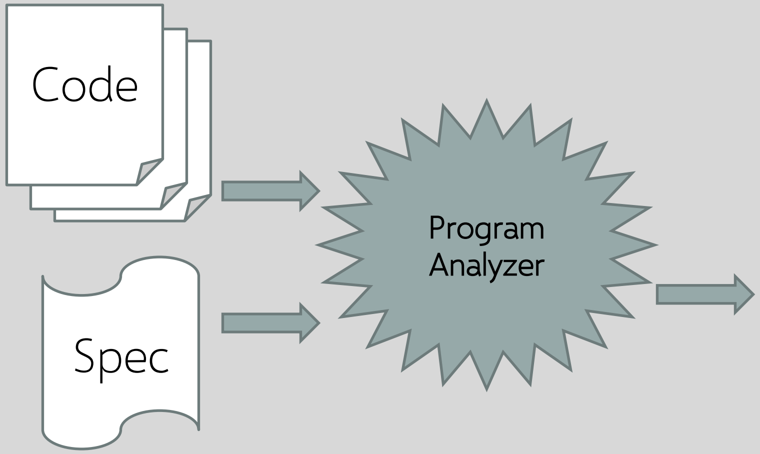
http://www.immunityinc.com/downloads/DaveAitel_TheHackerStrategy.pdf

How to Find Vulnerabilities?

Manual analysis
- Source code review
- Reverse engineering

Program analysis tools:
- Static analysis
- Fuzzing
- Symbolic analysis

Program Analyzers



analyze large code bases

Report	Type	Line
1	mem leak	324
2	buffer oflow	4,353,245
3	sql injection	23,212
4	stack oflow	86,923
5	dang ptr	8,491
…	…	…
10,502	info leak	10,921

false alarm (report 2)

false alarm (report 4)

potentially reports many warnings

may emit false alarms

False Positives and False Negatives

Term	Definition
False positive	A spurious warning that does not indicate an actual vulnerability
False negative	Does not emit a warning for an actual vulnerability

Complete analysis: no false negatives
Sound analysis: no false positives

Soundness and Completeness

	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**

Example Tools

Approach	Type	Comment
Lexical analyzers	Static analysis	Perform syntactic checks Ex: LINT, RATS, ITS4
Fuzz testing	Dynamic analysis	Run on specially crafted inputs to test
Symbolic execution	Emulated execution	Run program on many inputs at once Ex: KLEE, S2E, FiE
Model checking	Static analysis	Abstract program to a model, check that model satisfies security properties Ex: MOPS, SLAM, etc.

Source Code Scanners

Program that looks at source code, flags suspicious constructs

```
…
strcpy( ptr1, ptr2 );
…
```

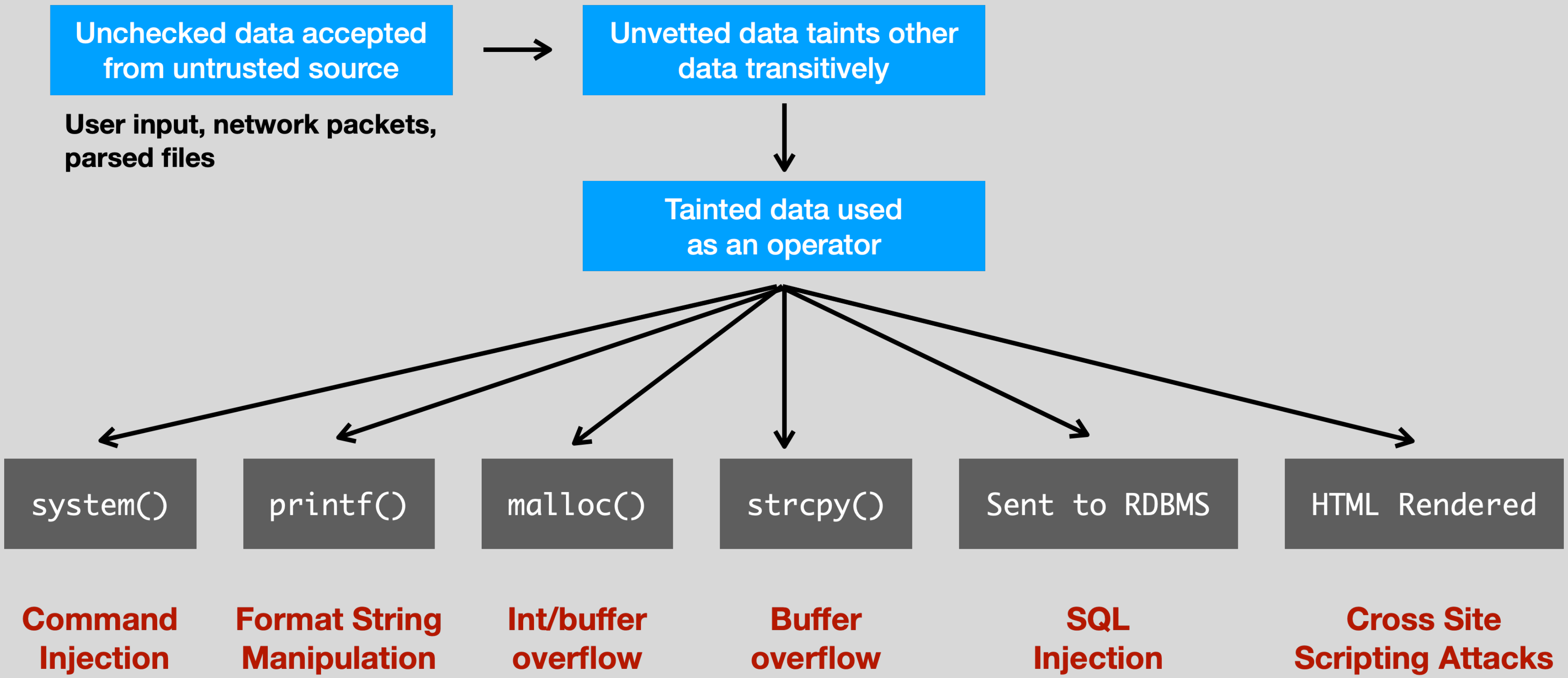
Warning: Don't use strcpy

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

(… but probably will)

Taint Checkers



Dynamic Analysis: Fuzzing

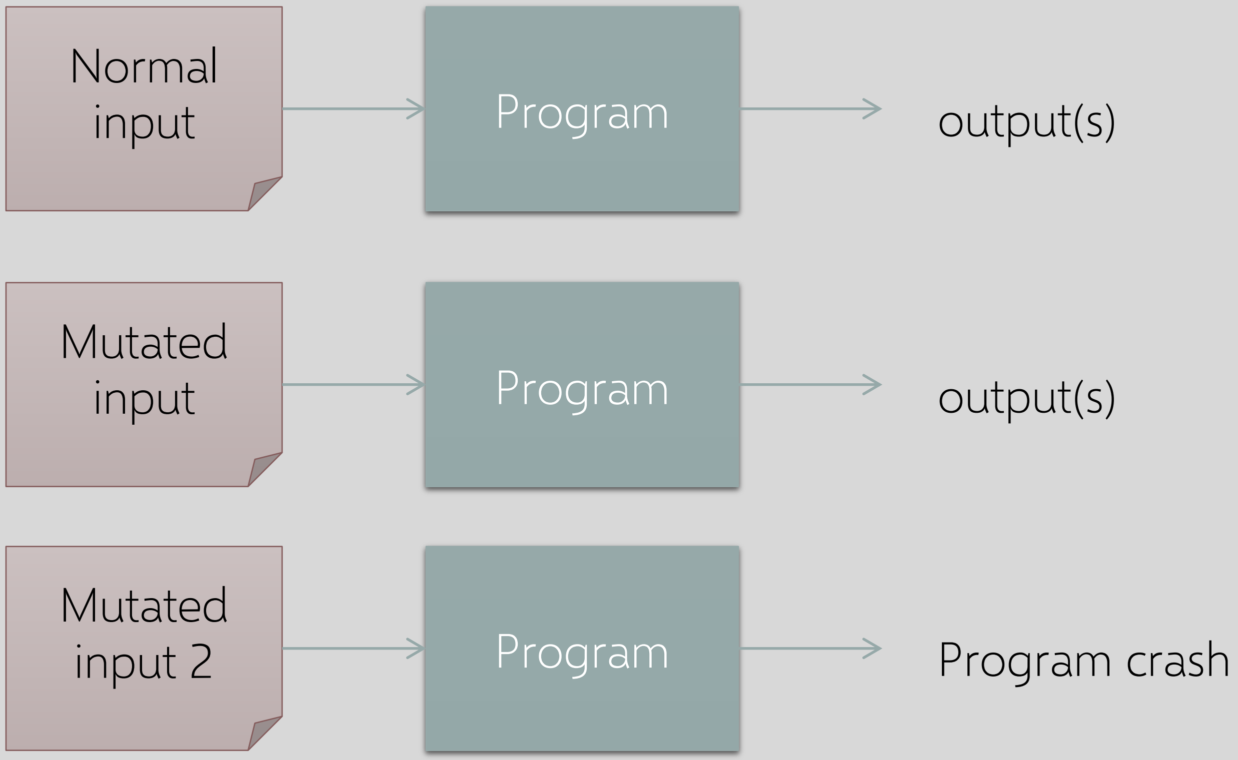
Choose a bunch of inputs
See if they cause program to crash

Key challenge: finding good inputs

"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."

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

Fuzzing



Normal input
Program
output(s)
Mutated input
Program
output(s)
Mutated input 2
Program
Program crash

HTTP Fuzzing Example

Standard HTTP GET request

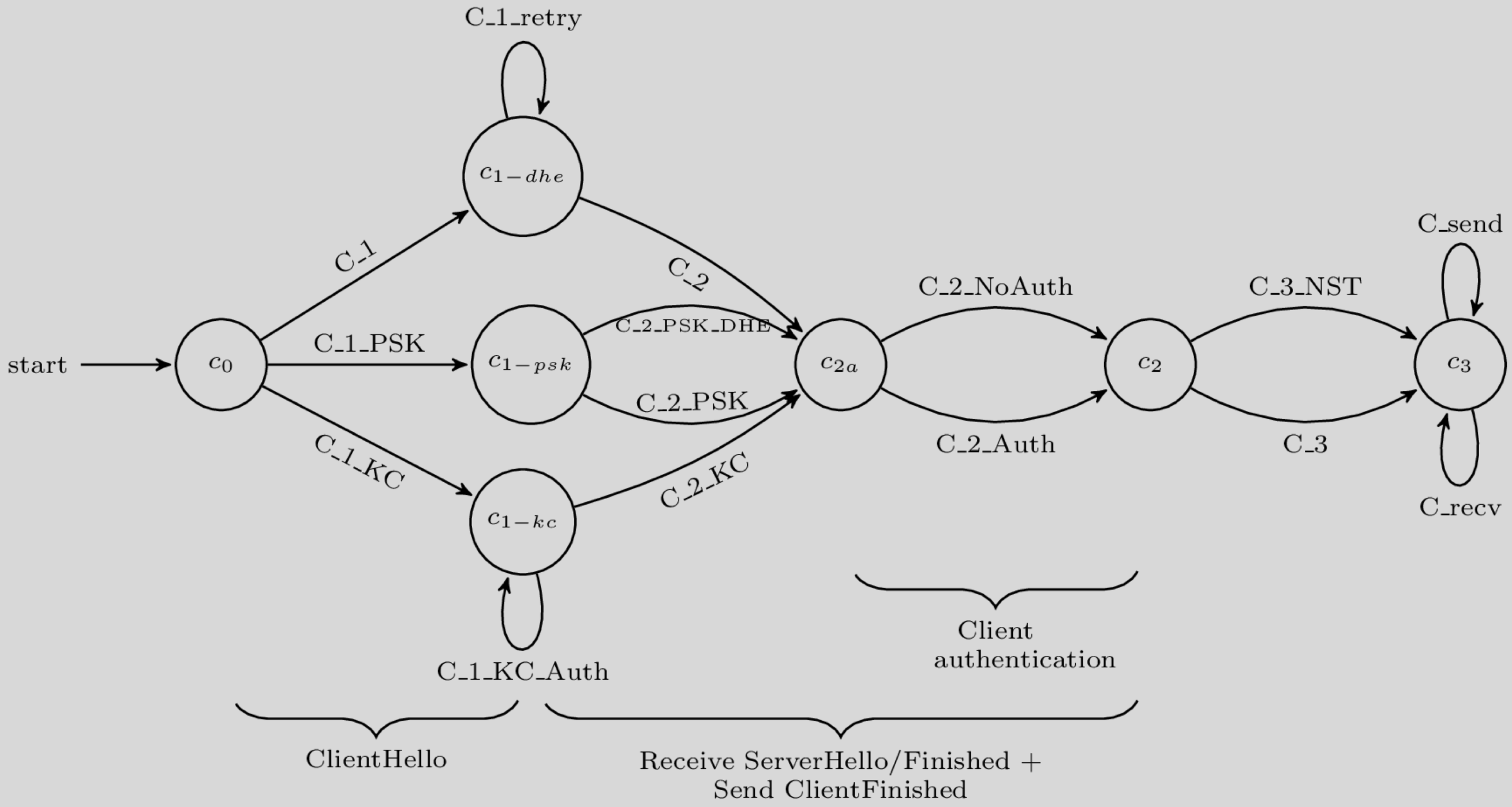
- GET /index.html HTTP/1.1

Anomalous requests

- GEEEE…EET /index.html HTTP/1.1
- GET ///////index.html HTTP/1.1
- GET %n%n%n%n%n%n.html HTTP/1.1
- GET /AAAAAAAAAAAAAAAAAAAAAAAAA.html HTTP/1.1
- GET /index.html HTTTTTTTTTTTTTP/1.1
- GET /index.html HTTP/1.1.1.1.1.1.1.1

but not df%w3rasd8#r78jskdflasdjf (why?)

Problem with Random Fuzzing



TLS 1.3 state diagram

Fuzzing

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( atoi(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 integers are 32 bits, then probability of crashing is **at most what?** $1/2^{32}$

Achieving code coverage can be very difficult

Code Coverage and Fuzzing Techniques

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

- Heuristics: increase string lengths (AAAAAAAAA…), randomly change items

Generative: start with specification of protocol, file format

- Build test case files from the spec, especially rarely used parts

Generation Example

```
<!--  A.  Local file header -->
  <Block name="LocalFileHeader">
    <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mut
    <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
    ...
    [truncated for space]
    ...
    <Number name="lfh_CompSize" size="32" endian="little" signed="false">
      <Relation type="size" of="lfh_CompData"/>
    </Number>
    <Number name="lfh_DecompSize" size="32" endian="little" signed="false"/>
    <Number name="lfh_FileNameLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FileName"/>
    </Number>
    <Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FldName"/>
    </Number>
    <String name="lfh_FileName"/>
    <String name="lfh_FldName"/>
    <!--  B.  File data -->
    <Blob name="lfh_CompData"/>
  </Block>
```

Mutation vs. Generation

	Ease of Use	Knowledge	Completeness	Complex Programs
Mutation	Easy to setup and automate	Little to no protocol knowledge required	Limited by initial corpus	May fail for protocols with checksums or other complexity
Generative	Writing generator is labor intensive	Requires having protocol specification	More complete than mutations	Handles arbitrarily compl ex protocols

Evolutionary Fuzzing

Generate inputs based on the structure and **response** of the program

Autodafe: Prioritizes based on inputs that reach dangerous API functions

EFS: Generates test cases based on code coverage metrics

Typically instrument program with additional instructions to track what code has been reached — or, if no source is available, track with Valgrind.

American Fuzzy Lop (AFL)

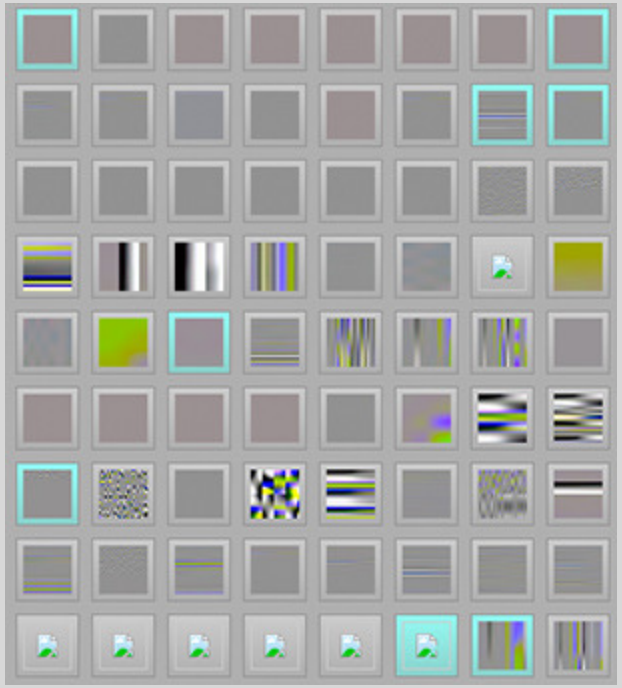
Widely used, highly effective fuzzing tool

- Specify example inputs
- Compile program with special afl compiler
- Run it

Performs mutation-based fuzzing

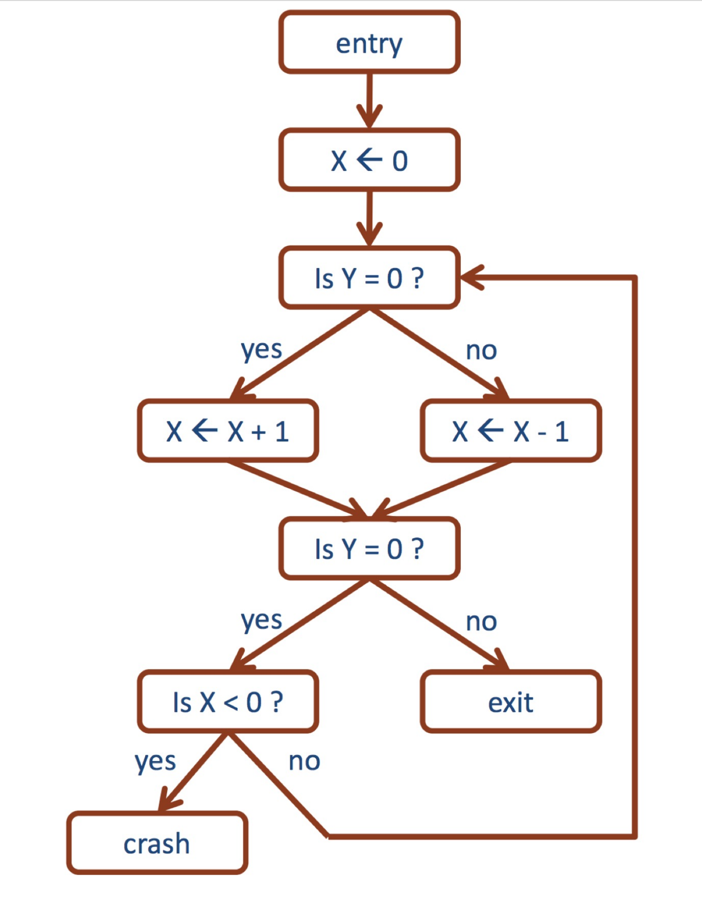
- Deterministic transforms to input (flip each bit, "walking byte flips", etc.)
- Randomized stacked transforms
- Measure (approximation of) path coverage, keep and mutate set of files that increase coverage

Really fast and simple. Used to find bugs in Firefox, OpenSSH, BIND, ImageMagick, iOS, …

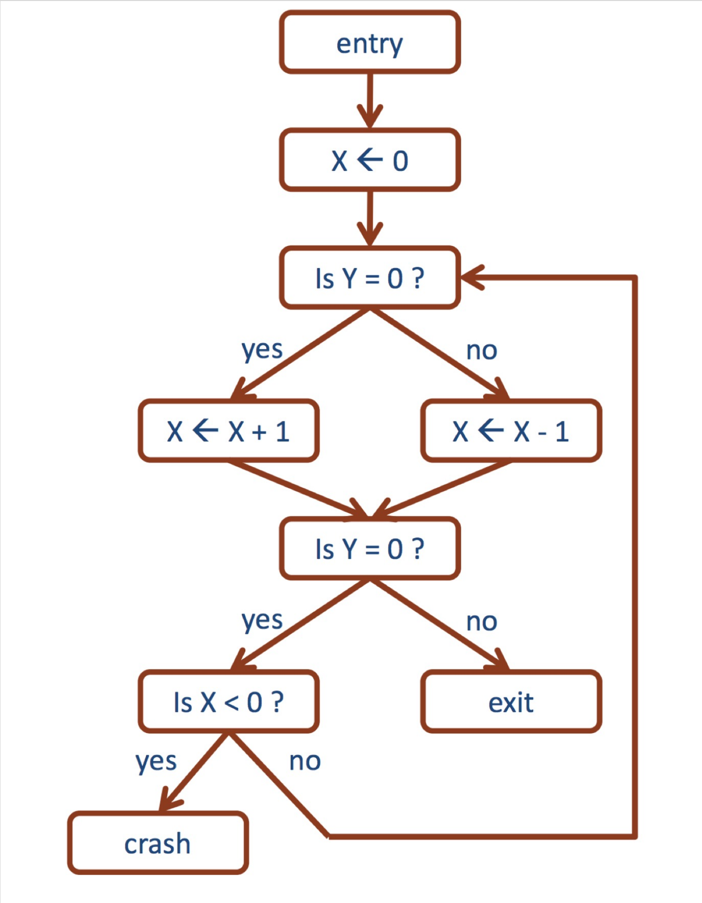


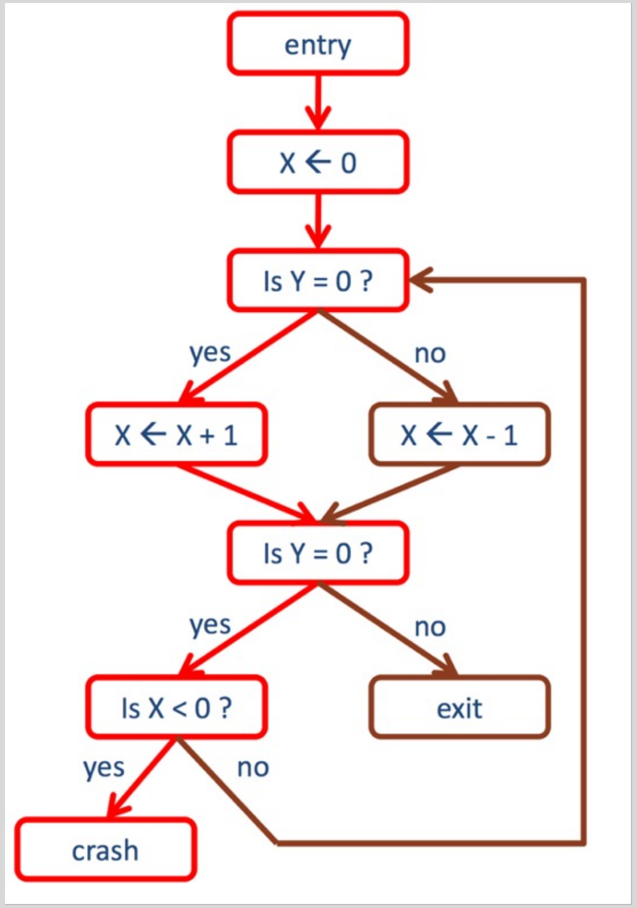
https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

Does This Program Crash?



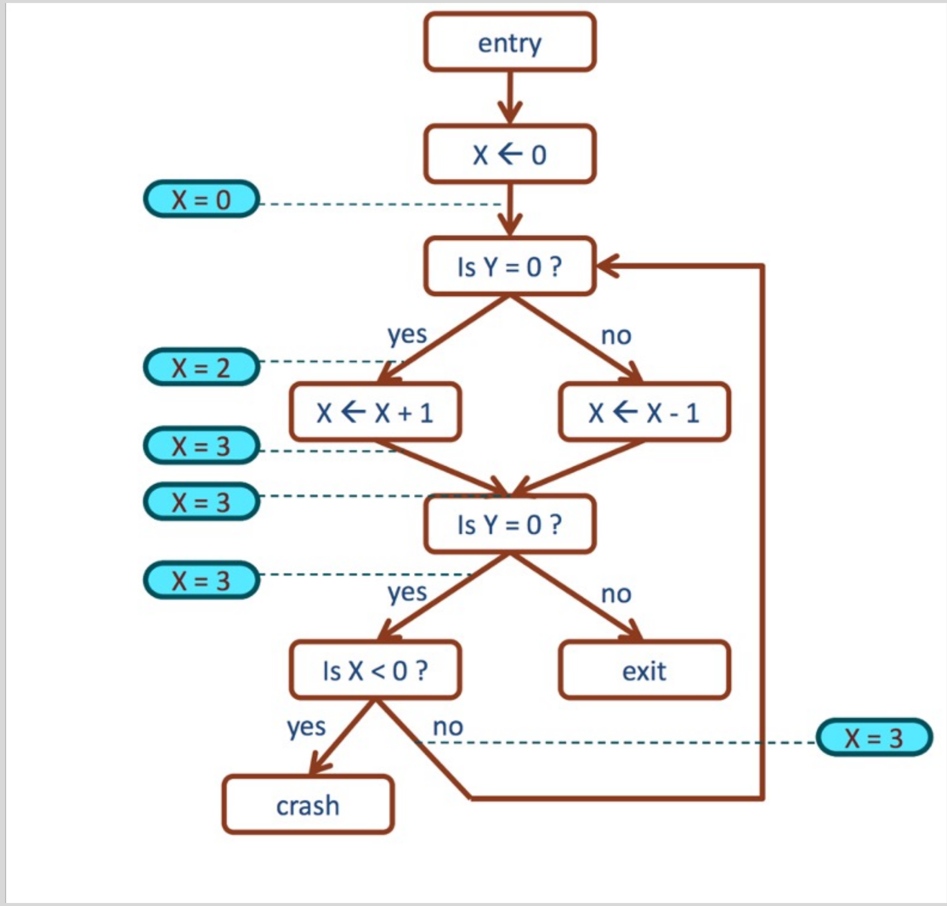
Does This Program Crash?





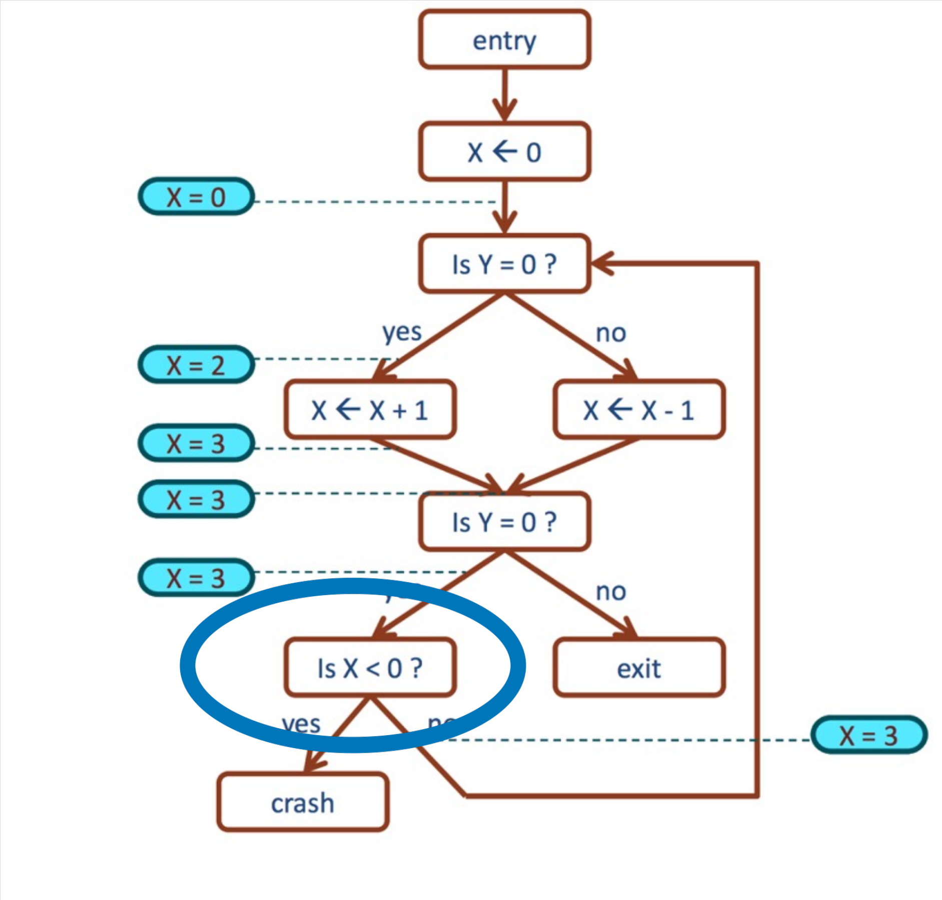
No!

Analysis With Concrete Values



Does not terminate...

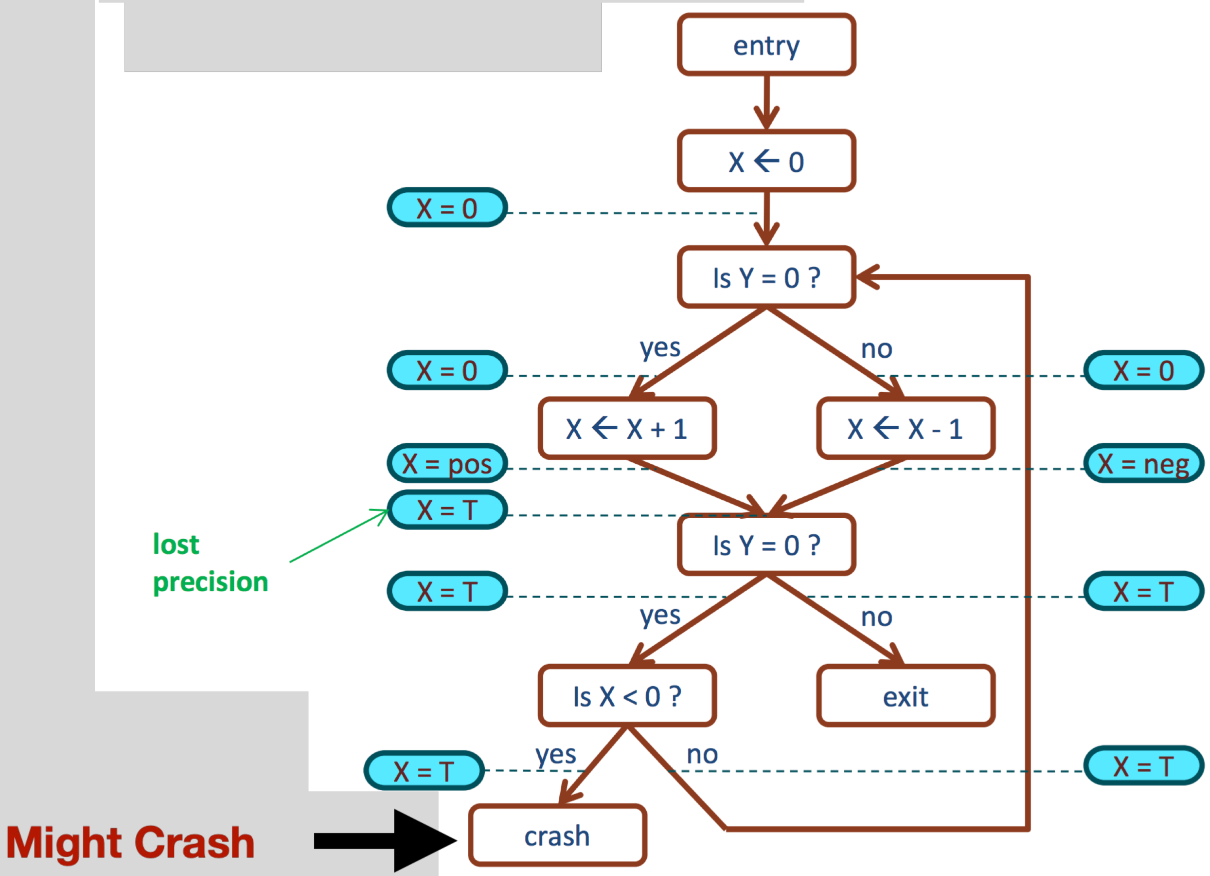
Abstract From Concrete Values



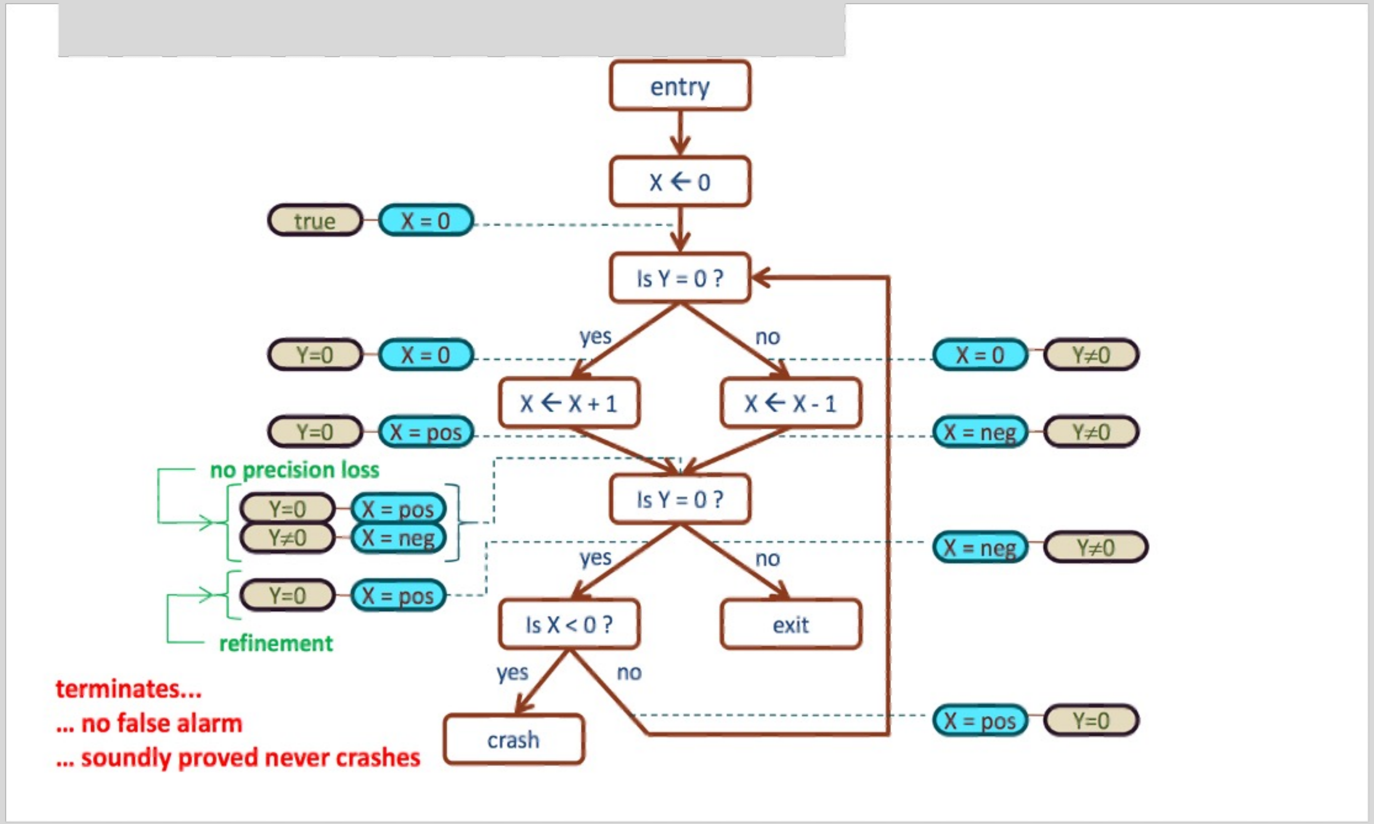
Abstraction

Concrete domain of integers		Abstract domain of signs
x=5	→	Positive integers
x=-5	→	Negative integers
x=0	→	Zero
x=b ? -1 : 1	→	Integers
x=y / 0	→	No integers (undefined)

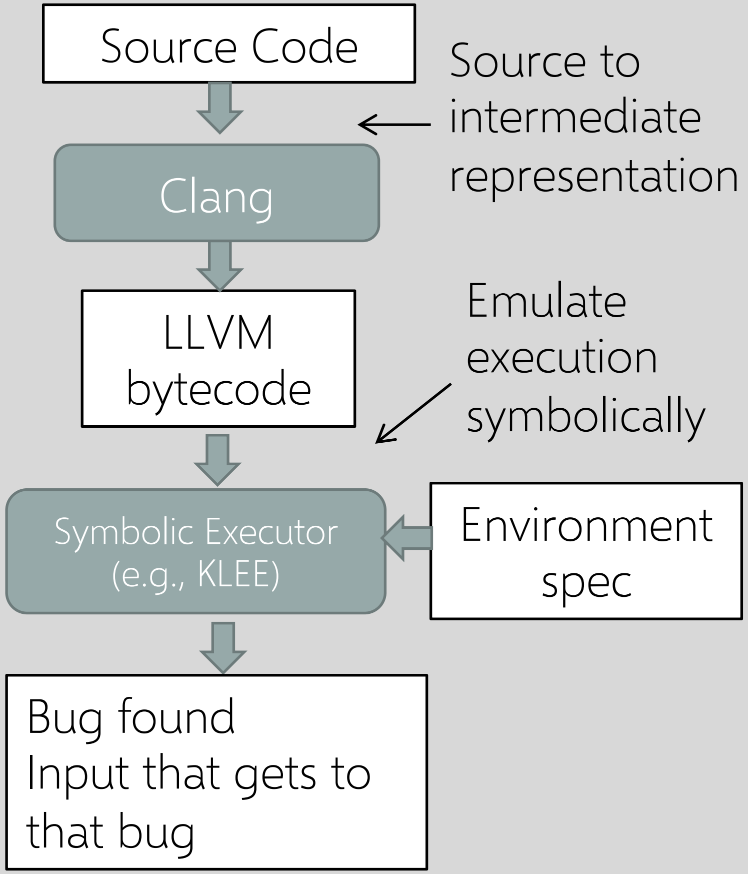
With “Signs” Approximation



Add Path Sensitivity



Symbolic Execution



- Technique for analyzing code paths and finding inputs
- Associate symbols to input variables ("symbolic variable")
- Simulate execution symbolically
  - Update symbolic variable's value appropriately
  - Conditionals add constraints on possible values
- Cast constraints as satisfiability, use SAT solver to find inputs
- Perform security checks at each execution state

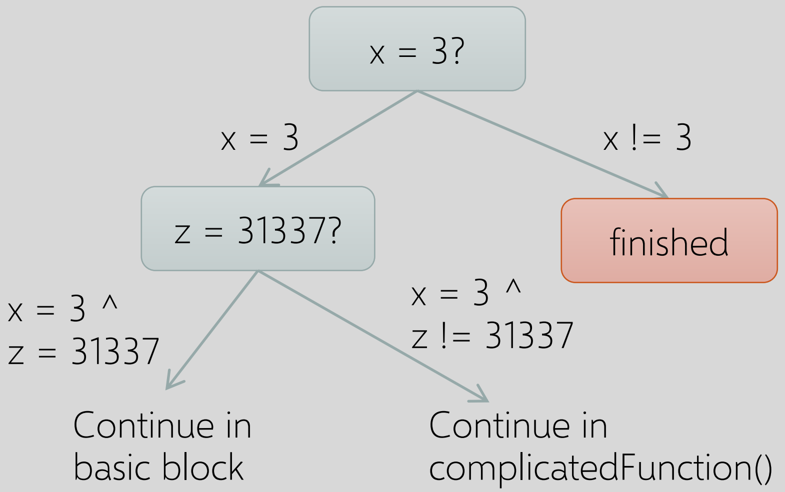
Symbolic Execution

```
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

Is it possible to complete analyses?

- Yes, but only for very simple programs
- Exponential number of paths to explore
- Each branch increases state size of symbolic emulator

Path selection

- Which state to explore next?
- 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)

Example Tools

Approach	Type	Comment
Lexical analyzers	Static analysis	Perform syntactic checks Ex: LINT, RATS, ITS4
Fuzz testing	Dynamic analysis	Run on specially crafted inputs to test
Symbolic execution	Emulated execution	Run program on many inputs at once, by Ex: KLEE, S2E, FiE
Model checking	Static analysis	Abstract program to a model, check that model satisfies security properties Ex: MOPS, SLAM, etc.

Google Address Sanitizer (ASan)

Memory error detector for C/C++ that finds…

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

Google Address Sanitizer (ASan)

LLVM Pass

- Modifies the code to check the shadow state for each memory access and creates poisoned redzones around stack and global objects to detect overflows and underflows

A run-time library that replaces memory management functions

- Replaces malloc, free and related functions, creates poisoned redzones around allocated heap regions, delays the reuse of freed heap regions, and does error reporting

Google Address Sanitizer (ASan)

```
==9901==ERROR: AddressSanitizer: heap-use-after-free on address 0x60700000dfb5 at pc
0x45917b bp 0x7fff4490c700 sp 0x7fff4490c6f8
READ of size 1 at 0x60700000dfb5 thread T0
    #0 0x45917a in main use-after-free.c:5
    #1 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
    #2 0x459074 in _start (a.out+0x459074)
0x60700000dfb5 is located 5 bytes inside of 80-byte region [0x60700000dfb0,0x60700000e000)
freed by thread T0 here:
    #0 0x4441ee in __interceptor_free projects/compiler-rt/lib/asan/asan_malloc_linux.cc:64
    #1 0x45914a in main use-after-free.c:4
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
previously allocated by thread T0 here:
    #0 0x44436e in __interceptor_malloc projects/compiler-rt/lib/asan/asan_malloc_linux.cc:74
    #1 0x45913f in main use-after-free.c:3
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

Summary of Program Analysis

	Pros	Cons
Static	Enables quickly finding bugs at development time Can detect some problems that dynamic misses	Either over or under reports. Misses complex bugs. Generally requires code.
Dynamic	May uncover complex behavior missed by static. Can run on blackbox.	Depends on user input—only checks executed code

Bug Finding is a Big Business

- Grammatech (Cornell startup, 1988)
- Coverity (Stanford startup)
- Fortify

… many, many others

- Also reverse engineers, exploit developers, zero-day markets…