CS 642: ASLR redux, Fuzzing, Program Analysis

Earlence Fernandes

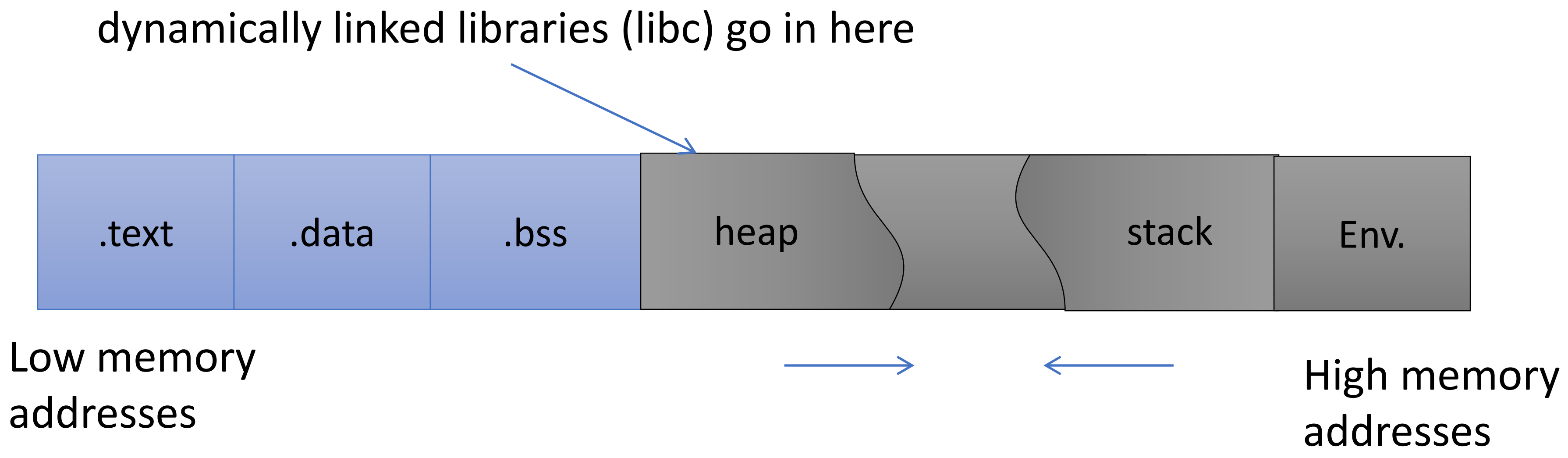
earlence@cs.wisc.edu

UNIVERSITY OF WISCONSIN–MADISON

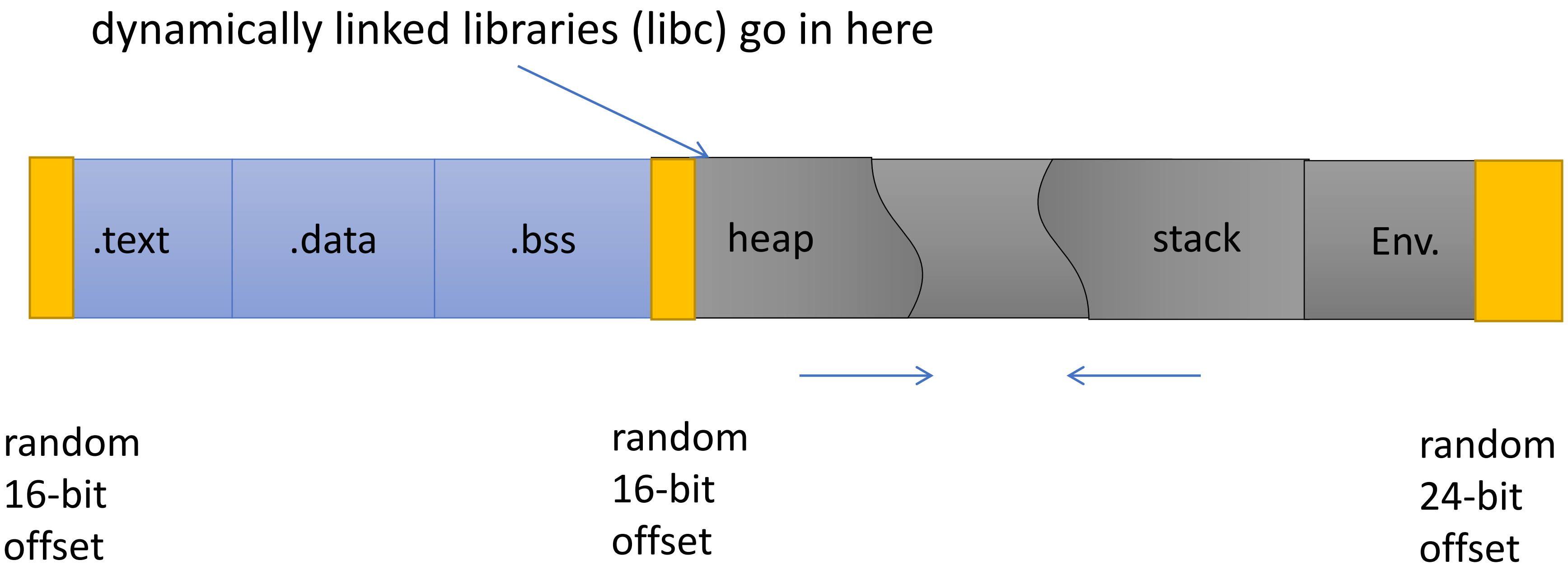
Announcements

- Midterm 2 is Apr 28 between 11 am and 12.15 pm (Central)
- PDF will be released at 10.55 am on canvas (just like homework)
- Write answers on blank pieces of paper
  - Write the question number (e.g., Q 5a) CLEARLY on paper (if not clear, zero)
  - Write answer
  - Scan whole thing using CamScanner (free for Android and iOS)
  - Submit on canvas (just like homework)
- Submit by 12.25 pm
  - 10 minutes extra for scanning
  - After 10 minutes, 1% deduction every 1 minute
- If timezone is an issue, email me immediately
  - Based on timezone poll, this should not be an issue in vast majority
- Study material will be released soon, but study everything between midterm 1 and Apr 23; Blackboard session during test to ask clarifying questions
- Honor system: You can use notes, but no chatting with each other, no searching Internet
  - Cheating on exam = cheating yourself

Address space layout randomization (ASLR)



Address space layout randomization (ASLR)



PaX ASLR (in Linux) implementation,:

- Randomize offsets of three areas
- 16 bits, 16 bits, 24 bits of randomness
- Adds unpredictability… but how much?

Defeating ASLR

- Large **nop** sled with classic buffer overflow (* W^X prevents this)
- Use a vulnerability that can be used to leak address information (e.g., printf arbitrary read)
- Brute force the address
  - "on average 216 seconds to compromise Apache running on a Linux PaX ASLR system", from Shacham et al., 2004 paper

Defeating ASLR

Brute-forcing example from reading "On the effectiveness of Address Space Layout Randomization" by Shacham et al.

request

Apache forks
off child process
to handle request

response

Apache web server
with Oracle 9 PL/SQL
module

There is a buffer overflow in
module that helps process request

Defeating ASLR

Brute-forcing example from reading “On the effectiveness of Address Space Layout Randomization” by Shacham et al.

request

Attacker makes a guess of where `usleep()` is located in memory

top of stack (higher addresses)
⋮
0x01010101
0xDEADBEEF
guessed address of `usleep()`
0xDEADBEEF
64 byte buffer, now filled with A’s
⋮
bottom of stack (lower addresses)

Apache web server with Oracle 9 PL/SQL module

Failure will crash the child process immediately and therefore kill connection

Success will crash the child process after sleeping for 0x01010101 microseconds and kill connection

If on 64-bit architecture, such brute-force attack unlikely to work

ASLR

Can also randomize more stuff:

- Instruction set randomization
- per-memory-allocation randomization
- etc.

Read this paper:

https://benpfaff.org/papers/asrandom.pdf

On the Effectiveness of Address-Space Randomization

Hovav Shacham
Stanford University
hovav@cs.stanford.edu

Matthew Page
Stanford University
mpage@stanford.edu

Ben Pfaff
Stanford University
blp@cs.stanford.edu

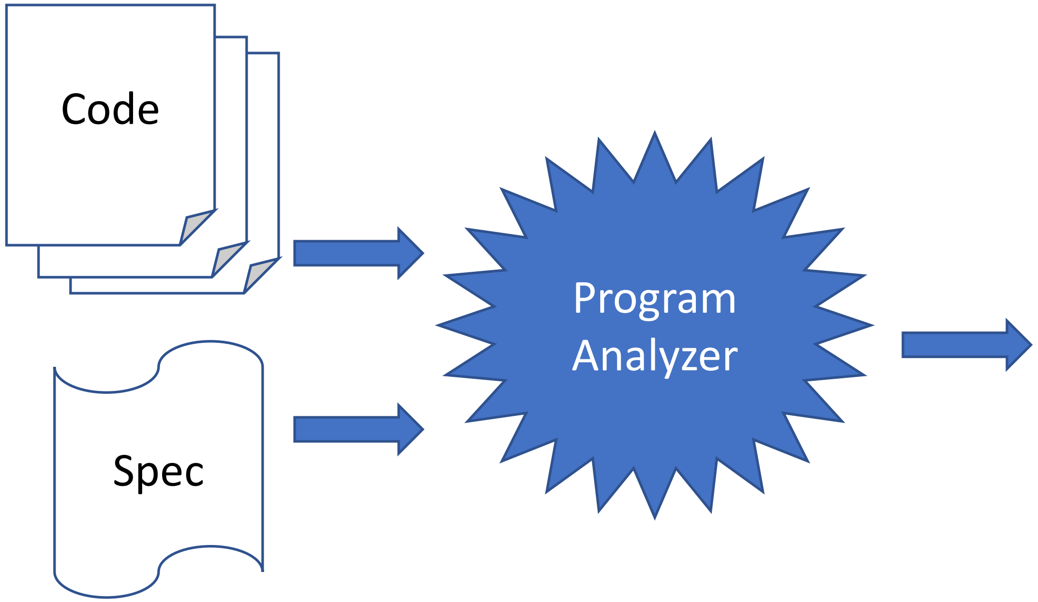
Eu-Jin Goh
Stanford University
eujin@cs.stanford.edu

Nagendra Modadugu
Stanford University
nagendra@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Tooling to find software security issues

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

false alarm

potentially reports many warnings

may emit false alarms

Slide credit: Prof Mitchell Stanford's CS 155

Program analysis: 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 positives

Sound analysis: no false negatives

	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 No False positives False negatives **Decidable**	May not report all errors May report false alarms False negatives False positives **Decidable**

Decidable: A *decision problem* that can be solved by an *algorithm* that halts on all inputs in a finite number of steps.

Undecidable: A problem that cannot be solved for all cases by any *algorithm* whatsoever.

Slide credit: Prof Mitchell Stanford’s CS 155

Program analyzers

Code

Spec

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

false alarm

potentially
reports many
warnings

may emit
false alarms

Manual analysis

- You get a binary or the source code
- You find vulnerabilities

- Experienced analysts according to Dave 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*

```
d <main+9>:     movl   $0xf8,(%esp)
4 <main+16>:    call   0x8048364 <malloc@plt>
9 <main+21>:    mov    %eax,0x14(%esp)
d <main+25>:    movl   $0xf8,(%esp)
4 <main+32>:    call   0x8048364 <malloc@plt>
9 <main+37>:    mov    %eax,0x18(%esp)
d <main+41>:    mov    0x14(%esp),%eax
1 <main+45>:    mov    %eax,(%esp)
4 <main+48>:    call   0x8048354 <free@plt>
9 <main+53>:    mov    0x18(%esp),%eax
d <main+57>:    mov    %eax,(%esp)
0 <main+60>:    call   0x8048354 <free@plt>
5 <main+65>:    movl   $0x200,(%esp)
c <main+72>:    call   0x8048364 <malloc@plt>
1 <main+77>:    mov    %eax,0x1c(%esp)
5 <main+81>:    mov    0xc(%ebp),%eax
8 <main+84>:    add    $0x4,%eax
b <main+87>:    mov    (%eax),%eax
d <main+89>:    movl   $0x1ff,0x8(%esp)
5 <main+97>:    mov    %eax,0x4(%esp)
9 <main+101>:   mov    0x1c(%esp),%eax
d <main+105>:   mov    %eax,(%esp)
0 <main+108>:   call   0x8048334 <strncpy@plt>
5 <main+113>:   mov    0x18(%esp),%eax
9 <main+117>:   mov    %eax,(%esp)
c <main+120>:   call   0x8048354 <free@plt>
1 <main+125>:   mov    0x1c(%esp),%eax
5 <main+129>:   mov    %eax,(%esp)
8 <main+132>:   call   0x8048354 <free@plt>
d <main+137>:   leave
e <main+138>:   ret
sembler dump.
```

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

How to draw an Owl.

"A fun and creative guide for beginners"

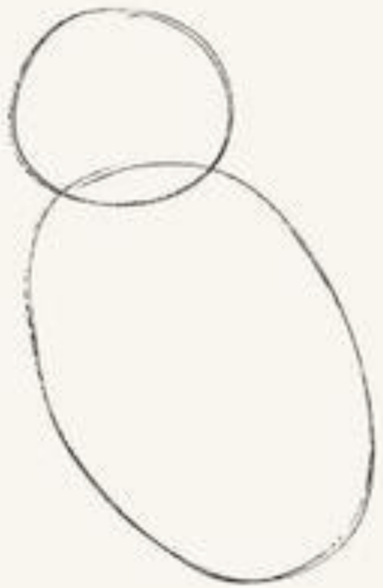


Fig 1. Draw two circles

Fig 2. Draw the rest of the damn Owl

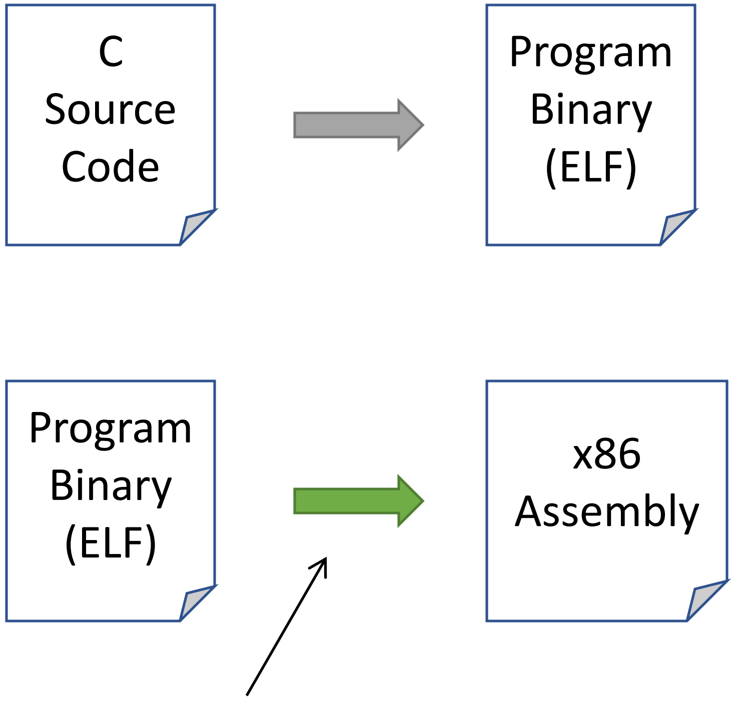
Example program analyzers

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

Disassembly and decompiling

The normal compilation process

What if we start with binary?

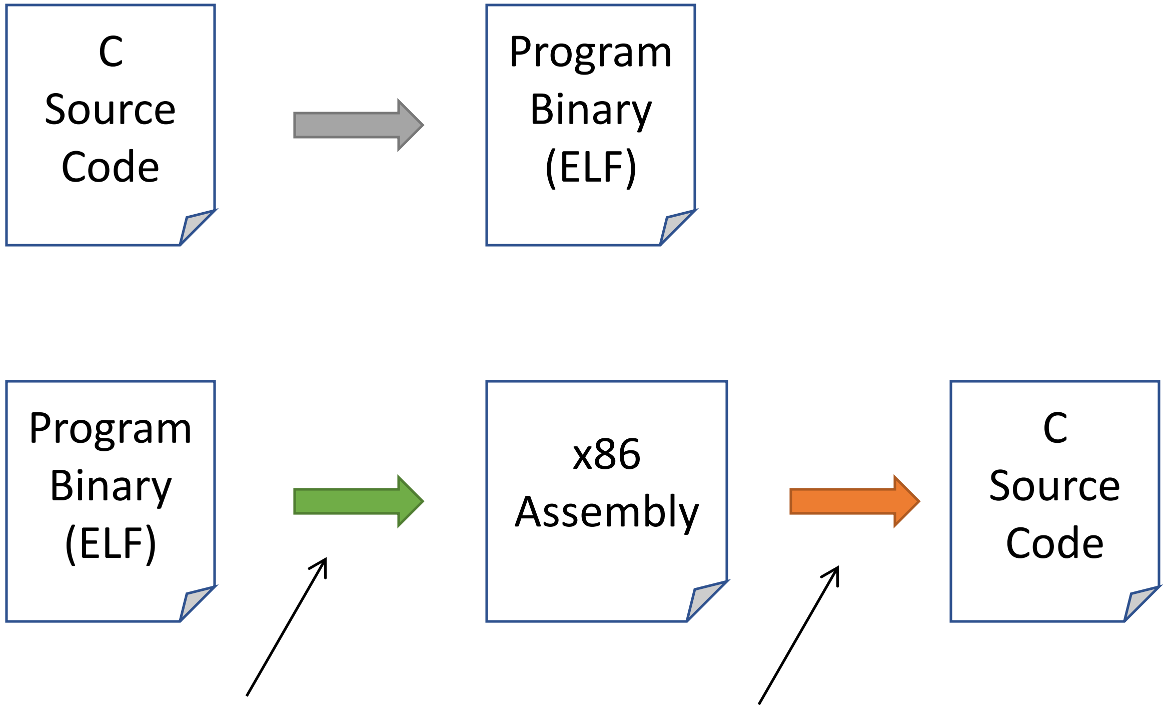


Disassembler
(gdb, IDA Pro, OllyDebug)

Disassembly and decompiling

The normal compilation process

What if we start with binary?

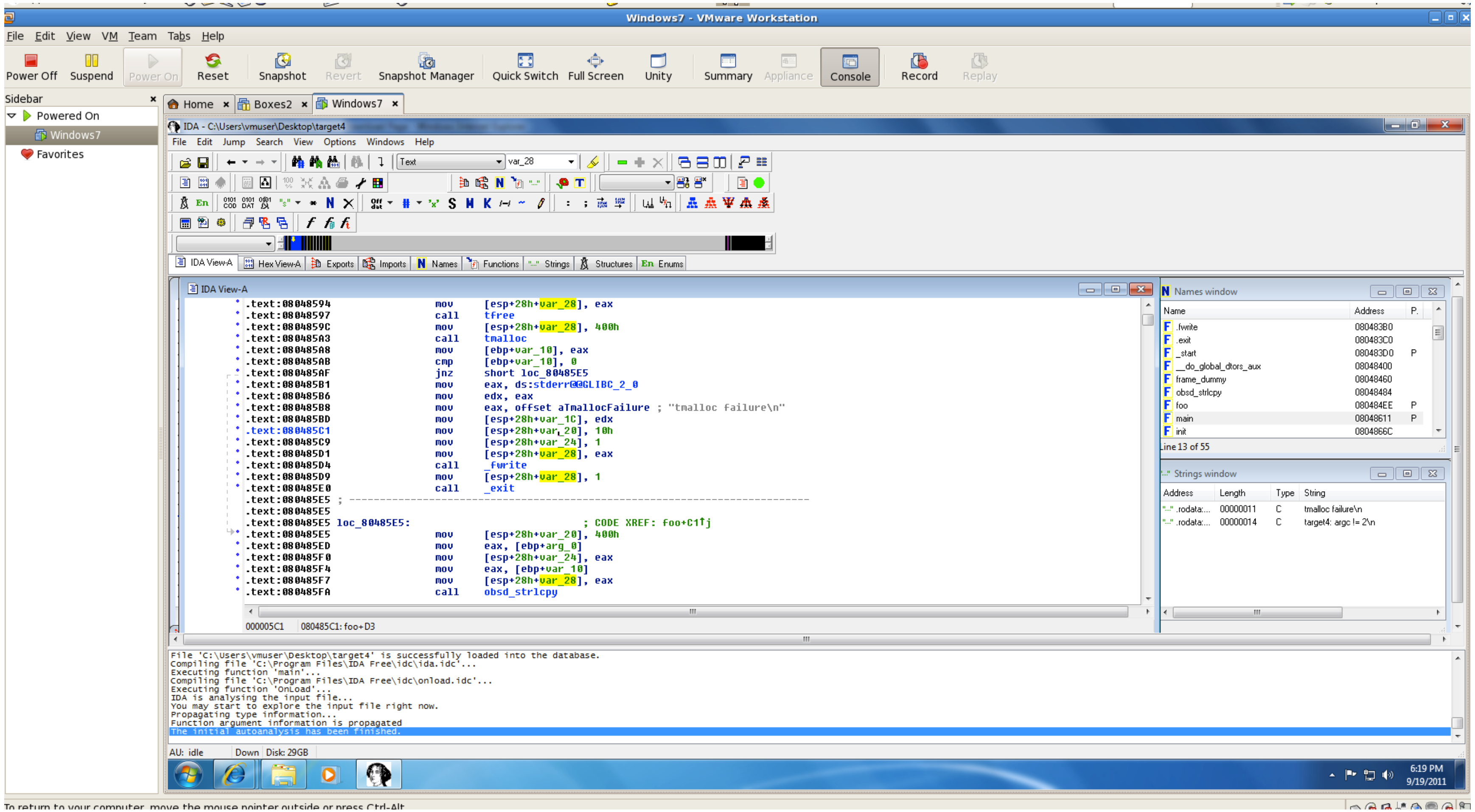


Disassembler
(gdb, IDA Pro, OllyDebug)

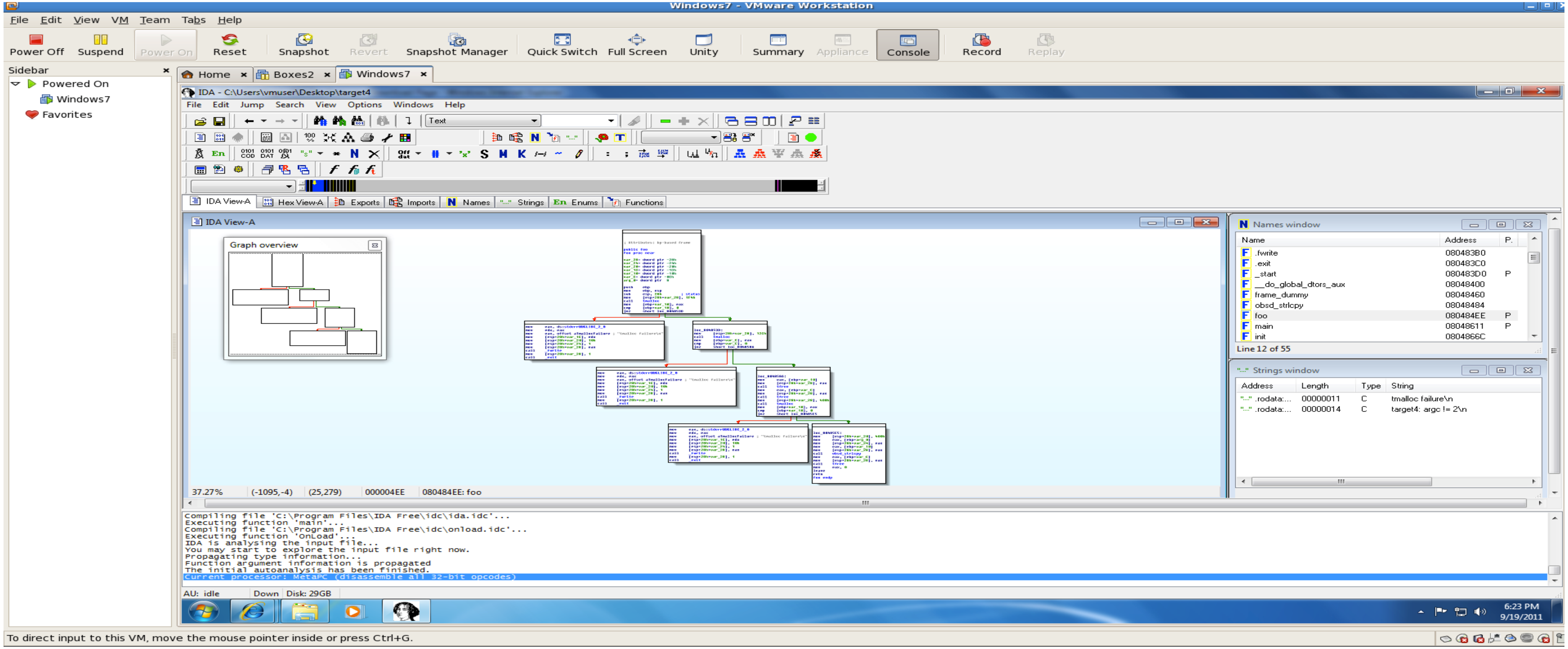
Decompiler
(IDA Pro has one)

Very complex, usually poor results

Tool example: IDA Pro



Tool example: IDA Pro



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( 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);
   }
}
```

Example tools / approaches

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

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)

Flawfinder

Circa 1990's technology:

shouldn't work for reasonable modern codebases

(… but probably will)

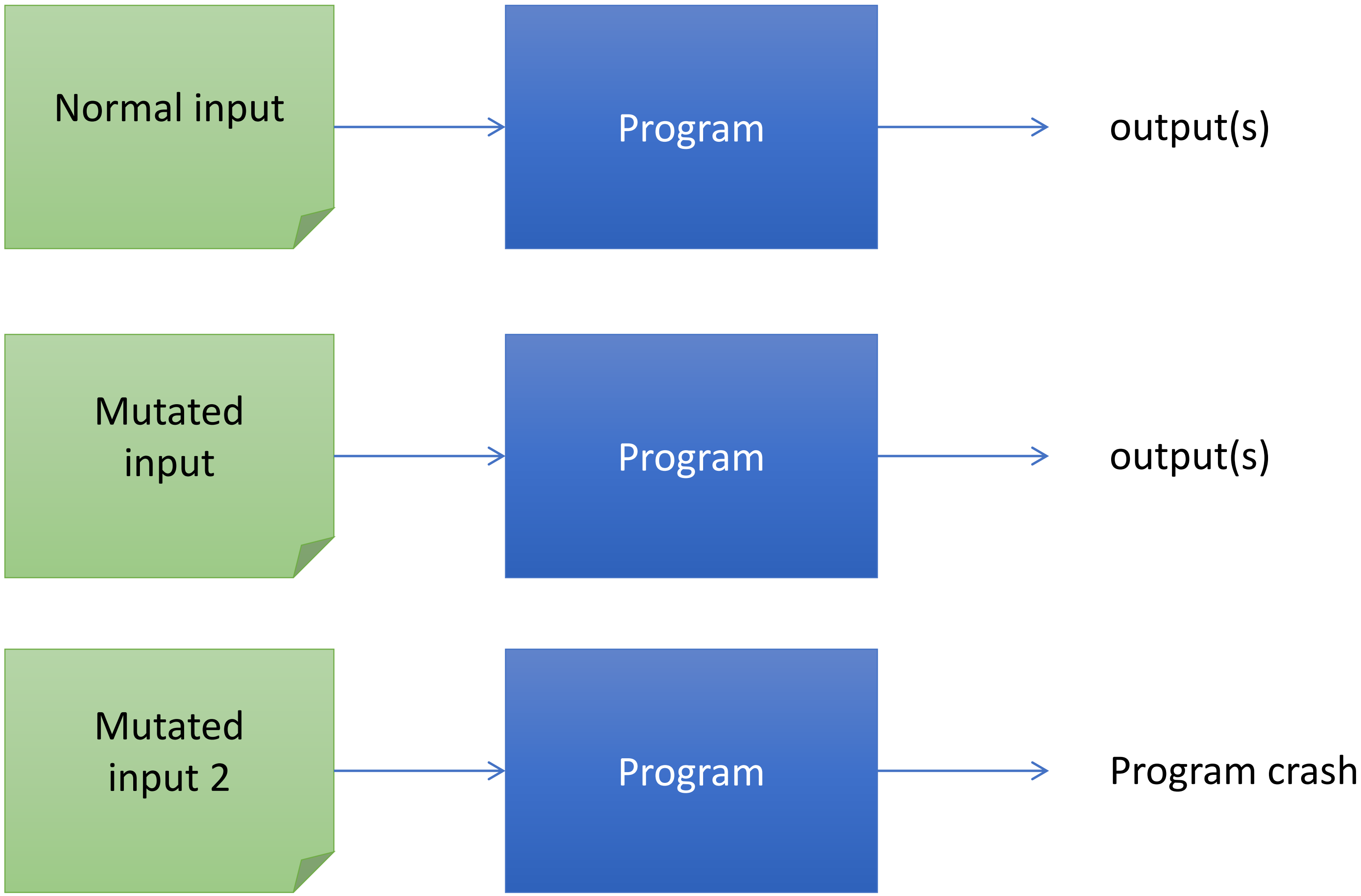
Dynamic analysis: Fuzzing

"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."

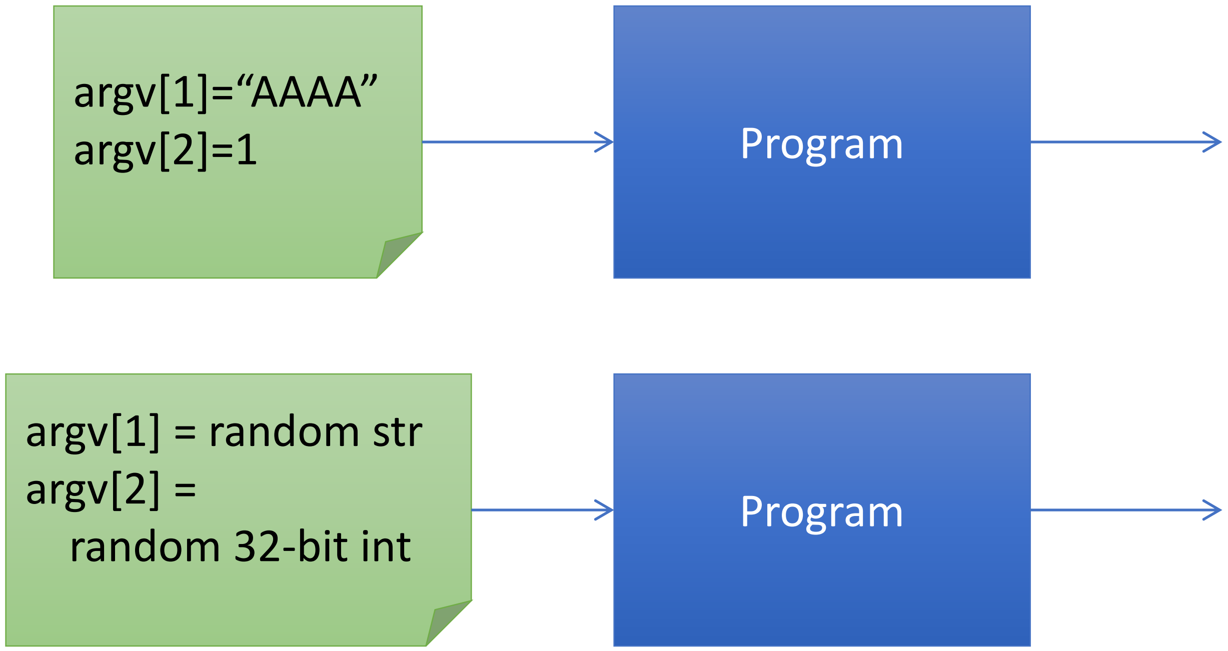
-- Wikipedia (http://en.wikipedia.org/wiki/Fuzz_testing)

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




```
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

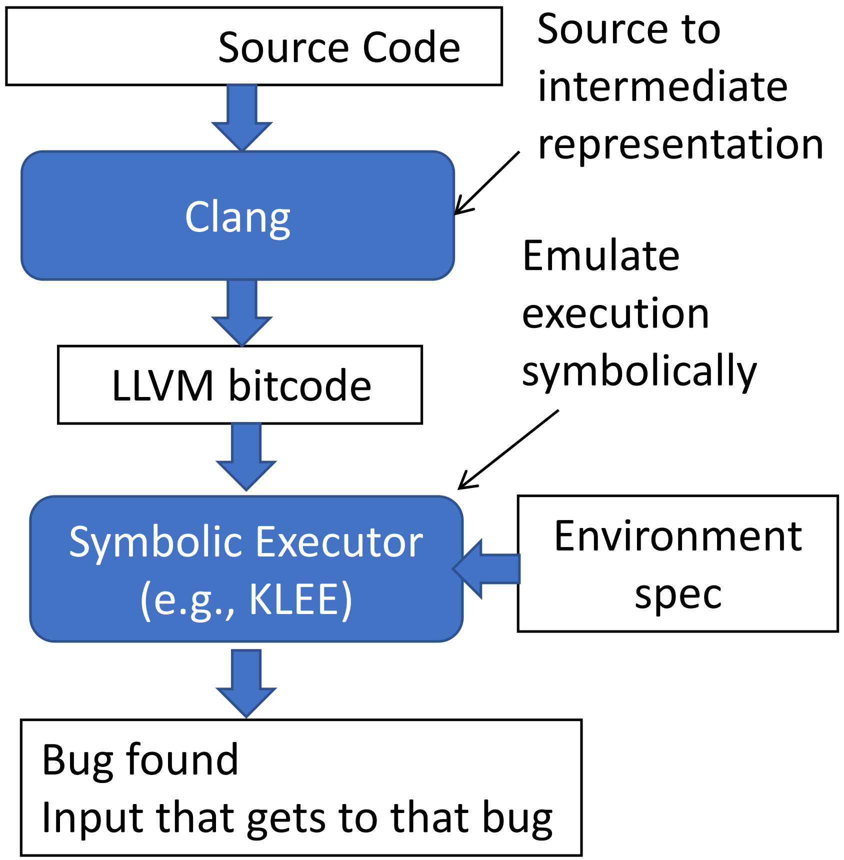
Code coverage and fuzzing

- 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 (AAAAAAAAA…)
    - randomly change items
- Generative
  - Start with specification of protocol, file format
  - Build test case files from it
    - Rarely used parts of spec

Example tools / approaches

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.

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
- Perform security checks at each execution state

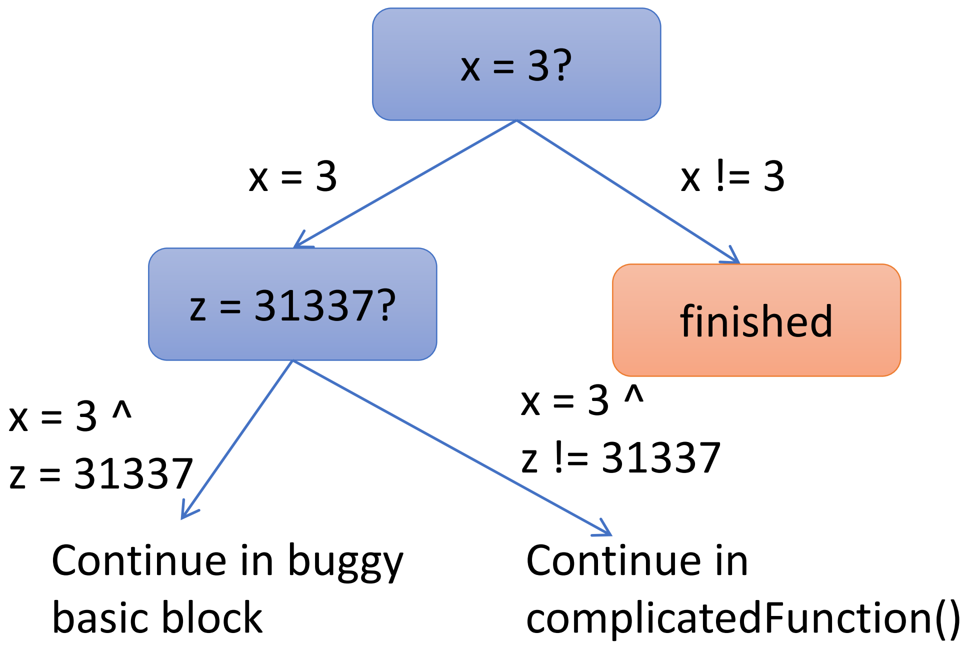
Symbolic execution

```
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

- 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)

White-box fuzz testing

- Start with real input and
  - Perform symbolic execution of program
  - Gather constraints (control flow) along way
  - Systematically negate constraints backwards
  - Eventually this yields a new input
- Repeat
- In-use at Microsoft

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