(Offensive) Binary Analysis

Advanced Internet Security

Adrian Dabrowski
Christian Kudera
Georg Merzdovnik
Aljosha Judmayer

News from the Lab

- Challenge 3 ends soon
  - Fastest solve by ”Octal Kansis” (`1day, 0:41:50`)
- Challenge 4 starts tomorrow

News from the Field

- Pwn2Own Tokyo 2018
  - disclosure of a total of 18 Zero-Day Vulnerabilities
  - exploited several devices (iPhone X, Samsung Galaxy S9, Xiaomi Mi6 Phones)
- Samsung Galaxy S9
  - heap overflow in the baseband component to get code execution
- Apple iPhone X running iOS 12.1
  - combination of a JIT bug in the Safari browser together with
  - out-of-bounds access to exfiltrate data from the phone

Overview

1. Binary Analysis Recap
2. Fuzz Testing (Fuzzing)
3. Instrumentation
4. Symbolic Execution

Binary Analysis Recap

Goals of Binary Analysis

- ”Defensive”
  - Program verification
  - Program testing
  - Debugging
- ”Offensive”
  - Reverse engineering
  - Vulnerability detection
  - Exploit generation

Binary Analysis

- Internet Security already covered:
  - Reverse engineering in general
  - Static (Disassembly) vs. Dynamic (Debugging)
  - Mostly manual work
- Now:
  - Let’s look at some techniques that can help us with automation

Fuzz Testing (Fuzzing)

Based on: https://www.slideshare.net/DmitryVyukov/fuzzing-the-new-unit-testing

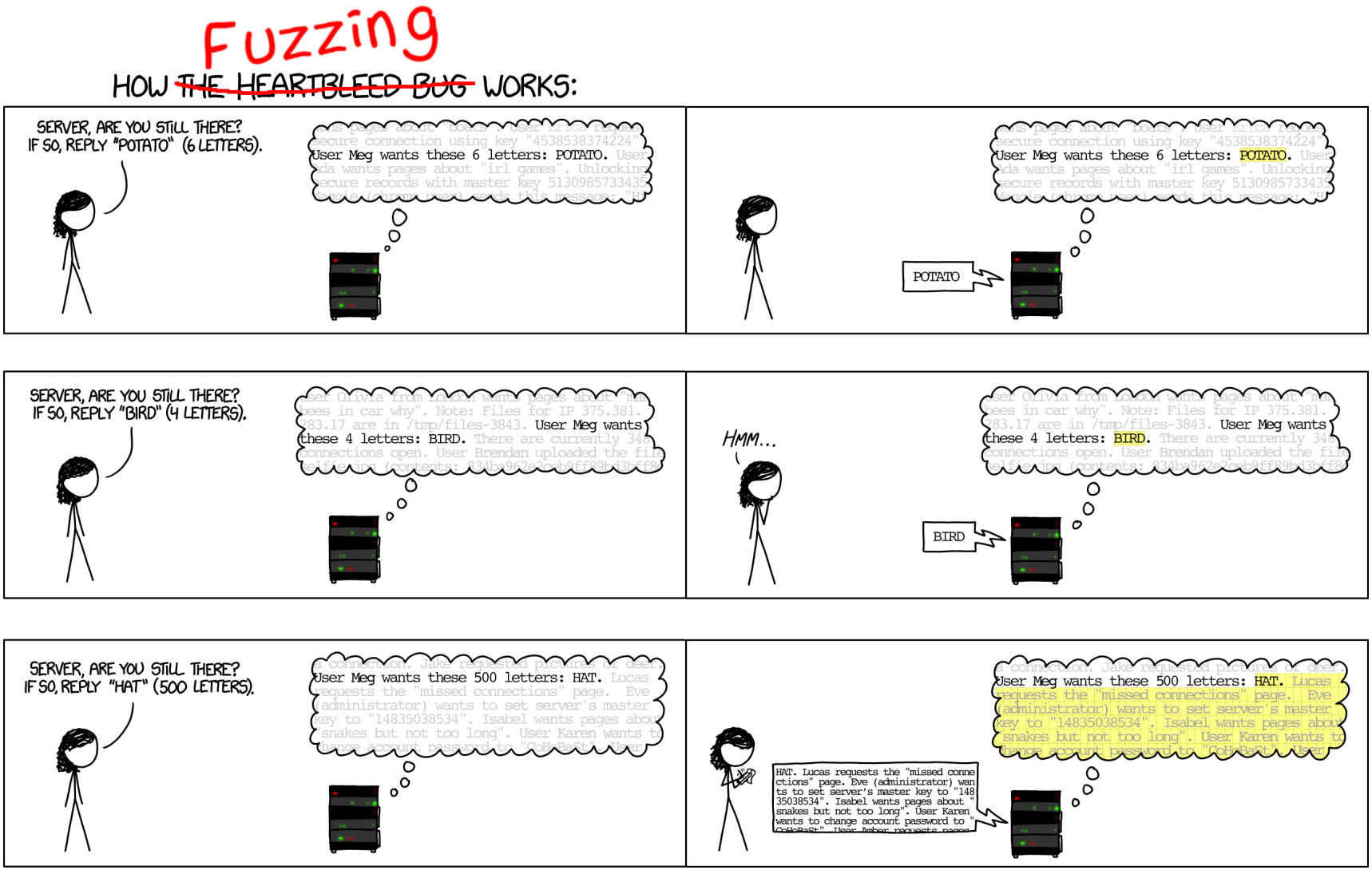
What is Fuzzing

- brute-force vulnerability detection
  - penetrate program with lots and lots of (semi-)random input
  - monitor program for crashes, dead-locks, etc.
- particularly successful in finding protocol/file parsing errors

User Testing vs Fuzzing

- User testing
  - Run program on many normal inputs, look for bad things to happen
  - **Goal:** Prevent normal users from encountering errors
- Fuzzing
  - Run program on many abnormal inputs, look for bad things to happen
  - **Goal:** Prevent attackers from encountering exploitable errors

Fuzzing in a Nutshell



https://xkcd.com/1354/

Fuzzing can find lots of bugs

- With the help of sanitizers[1]:
  - Use-after-free, buffer overflows
  - Uses of uninitialized memory
  - Memory leaks
  - Data races, deadlocks
  - Int/float overflows, bitwise shifts by invalid amount (other UB)
- Plain crashes:
  - NULL dereferences, uncaught exceptions, div-by-zero
- Resource usage bugs:
  - Memory exhaustion, hangs or infinite loops, infinite recursion (stack overflows)
  - Logical bugs

[1] We come back to that later

Not necessarily “random” input

- several ways to create input data
- may be human assisted or automated
- can achieve high levels of code coverage

What can be fuzzed?

- Anything that consumes complex inputs:
  - Parsers of any kind (xml, json, asn.1, pdf, truetype, …)
  - Media codecs (audio, video, raster & vector images, etc)
  - Network protocols (HTTP, RPC, SMTP, MIME…)
  - Crypto (boringssl, openssl)
  - Compression (zip, gzip, bzip2, brotli, …)
  - Formatted output (sprintf, template engines)
  - Compilers and interpreters (Javascript, PHP, Perl, Python, Go, Clang, …)
  - Regular expression matchers (PCRE, RE2, libc's regcomp)
  - Text/UTF processing (icu)
  - Databases (SQLite)
  - Browsers, text editors/processors (Chrome, vim, OpenOffice)
  - OS Kernels (Linux), drivers, supervisors and VMs

- **Must have** for everything that consumes untrusted inputs, open to internet or otherwise security sensitive.

Types of Fuzzers I

- Mutation Based - “Dumb” Fuzzing
  - has no knowledge about the program currently fuzzed (e.g. no information about input structure, protocols,…)
  - mutate existing data samples to create test data
  - Requires smallest amount of work to get running

Types of Fuzzers II

- Generation Based - “Smart” Fuzzing
  - has knowledge (model) about the program and input formats
  - define new tests based on models of the input
  - e.g Program that parses a protocol
    - might never parse certain fields if the structure of the input is not correct
    - but a wrong length field could still produce problematic outputs (see Heartbleed)
  - the greater the level of intelligence, the better the code coverage might be
- Evolutionary
  - generate inputs based on response from program

Input Generation I

- Blind mutation
  - Requires a corpus of representative inputs, apply random mutations to them
  - e.g., ZZUF, Radamsa
- Grammar-based generation
  - Generate random inputs according to grammar rules
  - e.g., Peach, packetdrill, csmith, gosmith, syzkaller
- Grammar reverse-engineering
  - Learn grammar from existing inputs using algorithmic approach of machine learning
  - e.g., go-fuzz

Input Generation II

- Symbolic execution + SAT solver
  - Synthesize inputs with maximum coverage
  - e.g., KLEE - Coverage-guided fuzzers
- Genetic algorithm that strives to maximize code coverage
  - e.g., libFuzzer, AFL, honggfuzz, syzkaller
- Hybrid Approaches

Coverage-guided fuzzing

```
Build the program with code coverage instrumentation;

Collect initial corpus of inputs (optional);

while (true) {
    Choose a random input from corpus and mutate it;

    Run the target program on the input,
        collect code coverage;

    If the input gives new coverage, add mutation
        back to the corpus;
}
```

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

- Requires "{if(" input to crash
  - $\sim 2^{32}$ guesses to crack when blind.

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

- Requires "{if(" input to crash
  - $\sim 2^{32}$ guesses to crack when blind.
- Coverage-guiding:

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

- Requires "{if(" input to crash
  - $\sim 2^{32}$ guesses to crack when blind.
- Coverage-guiding:
  - Guess "{" in $\sim 2^{8}$, add to corpus.

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

- Requires "{if(" input to crash
  - $\sim 2^{32}$ guesses to crack when blind.
- Coverage-guiding:
  - Guess "{" in $\sim 2^8$, add to corpus.
  - Guess "{i" in $\sim 2^8$, add to corpus.

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

- Requires "{if(" input to crash
  - $\sim 2^{32}$ guesses to crack when blind.
- Coverage-guiding:
  - Guess "{" in $\sim 2^8$, add to corpus.
  - Guess "{i" in $\sim 2^8$, add to corpus.
  - Guess "{if" in $\sim 2^8$, add to corpus.

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

- Requires "{if(" input to crash
  - $\sim 2^{32}$ guesses to crack when blind.
- Coverage-guiding:
  - Guess "{" in $\sim 2^8$, add to corpus.
  - Guess "{i" in $\sim 2^8$, add to corpus.
  - Guess "{if" in $\sim 2^8$, add to corpus.
  - Guess "{if(" in $\sim 2^8$, add to corpus.

Coverage-guiding in action

```
if input[0] == '{' {
  if input[1] == 'i' && input[2] == 'f' {
    if input[3] == '(' {
      input[input[4]] = input[5]; // potential OOB write
    }
  }
}
```

- Requires "{if(" input to crash
  - $\sim 2^{32}$ guesses to crack when blind.
- Coverage-guiding:
  - Guess "{" in $\sim 2^8$, add to corpus.
  - Guess "{i" in $\sim 2^8$, add to corpus.
  - Guess "{if" in $\sim 2^8$, add to corpus.
  - Guess "{if(" in $\sim 2^8$, add to corpus.
  - Total: $\sim 2^{10}$ guesses.

Mutation Based Fuzzing

- Little or no knowledge of the structure of the inputs is assumed
- Requires little to no set up time
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics
- Dependent on the inputs being modified
- May fail for protocols with checksums, those which depend on challenge response, etc.

Mutations

- erase/insert/change/shuffle bit/byte/bytes
- crossover/splice 2 inputs
- insert magic numbers (2^10±1, 2^16±1, 2^31±1, 2^32±1)
- change an ASCII integer (e.g. “123” => “2465357635”)
- insert token from a dictionary
- …

Mutation dictionaries

- User-provided
  - e.g. for HTTP: “HTTP/1.1”, “Host”, “Accept-Encoding”
- Automatically extracted from program
  - memcpy(input, “HTTP/1.1”, 8)

Mutation Based Example: PDF Fuzzing

- Collect .pdf files
  - e.g. search on Google and download
- Use a mutation Fuzzer:
  1. get PDF
  2. mutate the file
  3. open it with viewer
  4. monitor if program crashes

Generation Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing
- Can take significant time to set up

Mutation vs. Generation

Mutation	Generation
Easy setup and automation	Writing the generator can be cumbersome/take a lot of work
Little to no knowledge about the input required	Need the input specification
Limited by initial corpus	Completeness (Better input space coverage)
May fail for protocols with checksums/complexities	Can deal with complexities/dependencies between input elements

Black-, Grey-, Whitebox Fuzzing

- Black-box fuzzer treats the program as black box
  - no knowledge about it’s internal structure
- White-box fuzzer uses program analysis to increase code coverage
  - e.g. using symbolic execution
- Gray-box fuzzer uses instrumentation instead of program analysis to extract information
  - e.g. AFL and libFuzzer track basic block transitions

Challenges in Fuzzing

- Mutation based
  - could run forever.
  - When do we stop?
- Generation based
  - Stops **eventually**
  - But is this enough?
- How to determine if the program did something “bad”?

Code Coverage

- can give a metric to determine how “well” the code was tested
- Coverage flavours:
  - Statement Coverage - which statements have been executed
    - tries to cover all basic blocks in the program at least once
  - Branch coverage - which branches have been taken
    - cover all transitions (edges) in the program at least once
  - Path coverage - which paths were taken
    - coverage of e.g. iterations (edges taken multiple times)

Finding logical bugs

- Not only security/stability
- We could also detect logic bugs/errors

Finding logical bugs

- Not only security/stability
- We could also detect logic bugs/errors
- Problem: We might not know the right result/output!

Finding logical bugs

- Not only security/stability
- We could also detect logic bugs/errors
- Problem: We might not know the right result/output!
- Solution: Be inventive

Finding logical bugs

- sanity checks on results
  - uncompressed image decoder: 100 byte input -> 100 MB output?
  - function returns both error and object, or no error and no object
  - know that some substring must be present in output, but it is not
  - encrypt, check that decryption with wrong key fails
- sometimes we do know the right result
  - any sorting: check that each element is present in output, check that it's not in wrong order
  - building a trie: check size, all elements are present
- include asserts
  - `assert(a == b)`

Finding logical bugs

- Round-trip:
  - encode-decode
  - serialize-deserialize
  - compress-decompress
  - encrypt-decrypt
  - assemble-disassemble
- Checks:
  - decode-encode: check that encode don’t fail
  - decode-encode-decode: check that second decode don’t fail
  - decode-encode-decode: check that decode results are equal
  - encode-decode-encode: check that encode results are equal
- Very powerful technique.

Finding logical bugs

- Comparing two (or more) implementations:
  - check that output is equal
  - or at least check that ok/fail result is the same
    - e.g. gcc and clang both accept or reject the code
- If you don’t want to implement second one yourself, consider:
  - there can be several libraries implementing the same (libxmlFoo vs libxmlBar)
  - implementation in a different language (re2 vs Go’s regexp)
  - compare “fast but complex” with “slow but dumb” (sometimes easy to write)
  - compare different functions (marshalBinary vs marshalText)

Regression testing

- Normally you run fuzzer for a long time.
- But any guided fuzzer accumulates corpus of inputs with max coverage.
- And that’s perfect for regression testing! Just run it once on every change!

Different Fuzzing Projects

- OSS-Fuzz
  - fuzzing as a service by Google for open source projects
- The Fuzzing Project
  - information on fuzzing efforts

Fuzzing Lessons

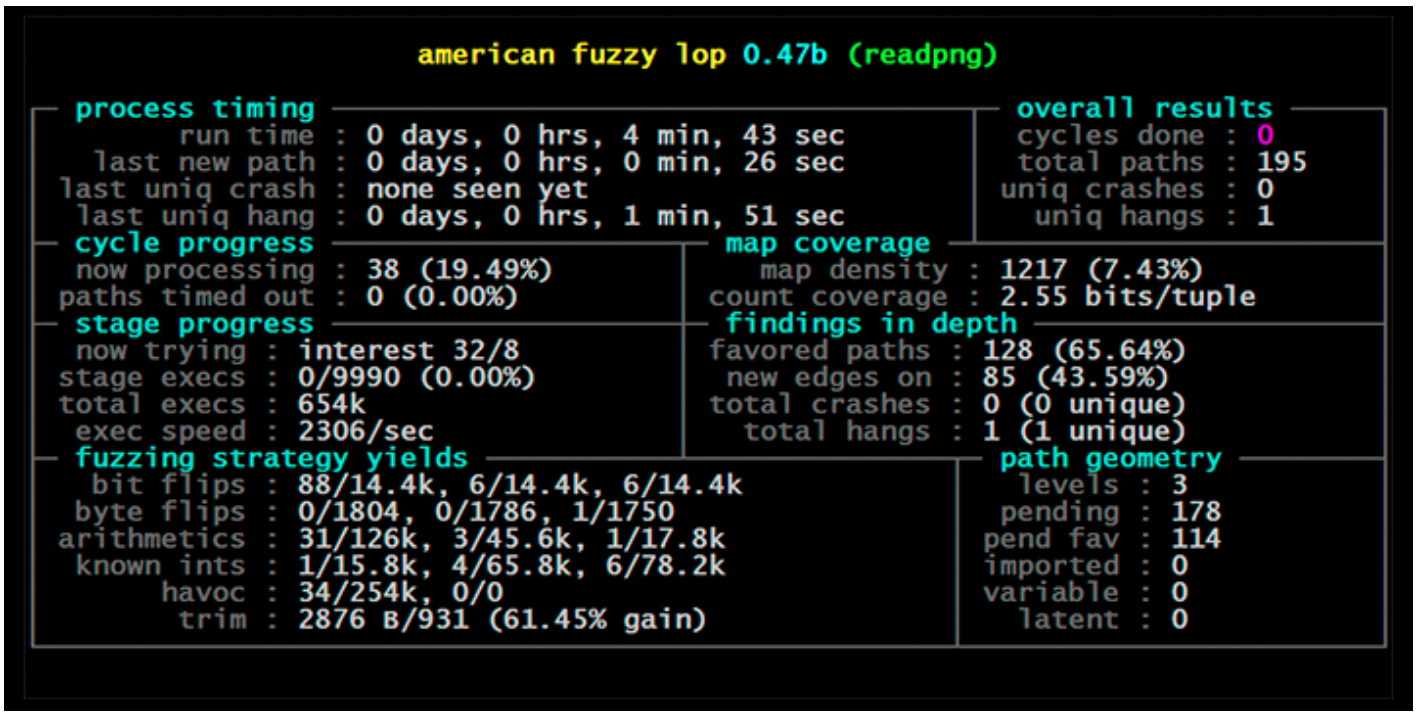
- Protocol knowledge is helpful
  - Generational beats random, better specification make better fuzzers
- Using more fuzzers is better
  - Each one will vary and find different bugs
- The longer you run (typically) the more bugs you'll find
- Guide the process, fix it when it break or fails to reach where you need it to go
- Code coverage can serve as a useful guide

Fuzzing Summary

- Fuzzing is complimentary to any other testing technique
- Fuzzing is mandatory for anything security-related
- Fuzzing finds LOTS of bugs
- Fuzzing is easy to use

Tool Example: American Fuzzy Lop (AFL)

- Security oriented guided fuzzer
- Based on compile time instrumentation
- Uses genetic algorithms to get better code coverage
  - Use together with AddressSantizer for even better results

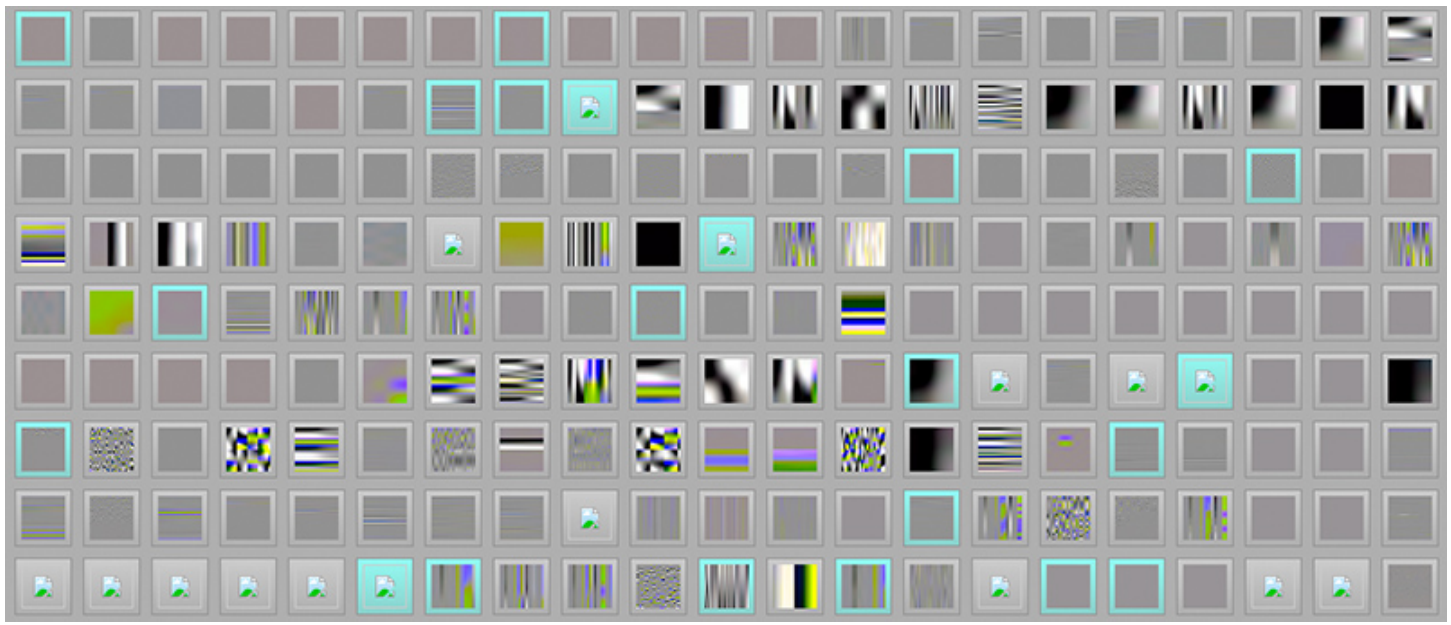


AFL - Why use it

- Simple to use
- No configuration of AFL necessary
  - But usually re-compilation of the target required
- Fast
- Produces good input files (e.g. also for use in other fuzzers)

AFL - Pulling JPEGs out of thin air

- fuzzing of djpeg with AFL
- input was txt with “hello”
- AFL started to produce the first valid JPEG file after about six hours on an 8 core machine.



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

AFL - Usual use case

- You have access to the source code and compile it yourself
- Your are on 32bit or 64bit on Linux/OSX/BSD
- The to-be-fuzzed code (e.g. parser) reads it's input from stdin or from a file
- The input file is usually only max. 10kb
- This covers *a lot* of Linux libraries

AFL - What if prerequisites are not met

- No source code?
  - Try the experimental QEMU instrumentation
- Not on 32/64 bit?
  - There is an experimental ARM version
- Not reading from stdin or file?
  - Maybe your project has a utility command line tool that does read from file
  - Or you write a wrapper to do it
  - Same if you want to test (parts of) network protocol parsers

AFL - Steps of fuzzing I

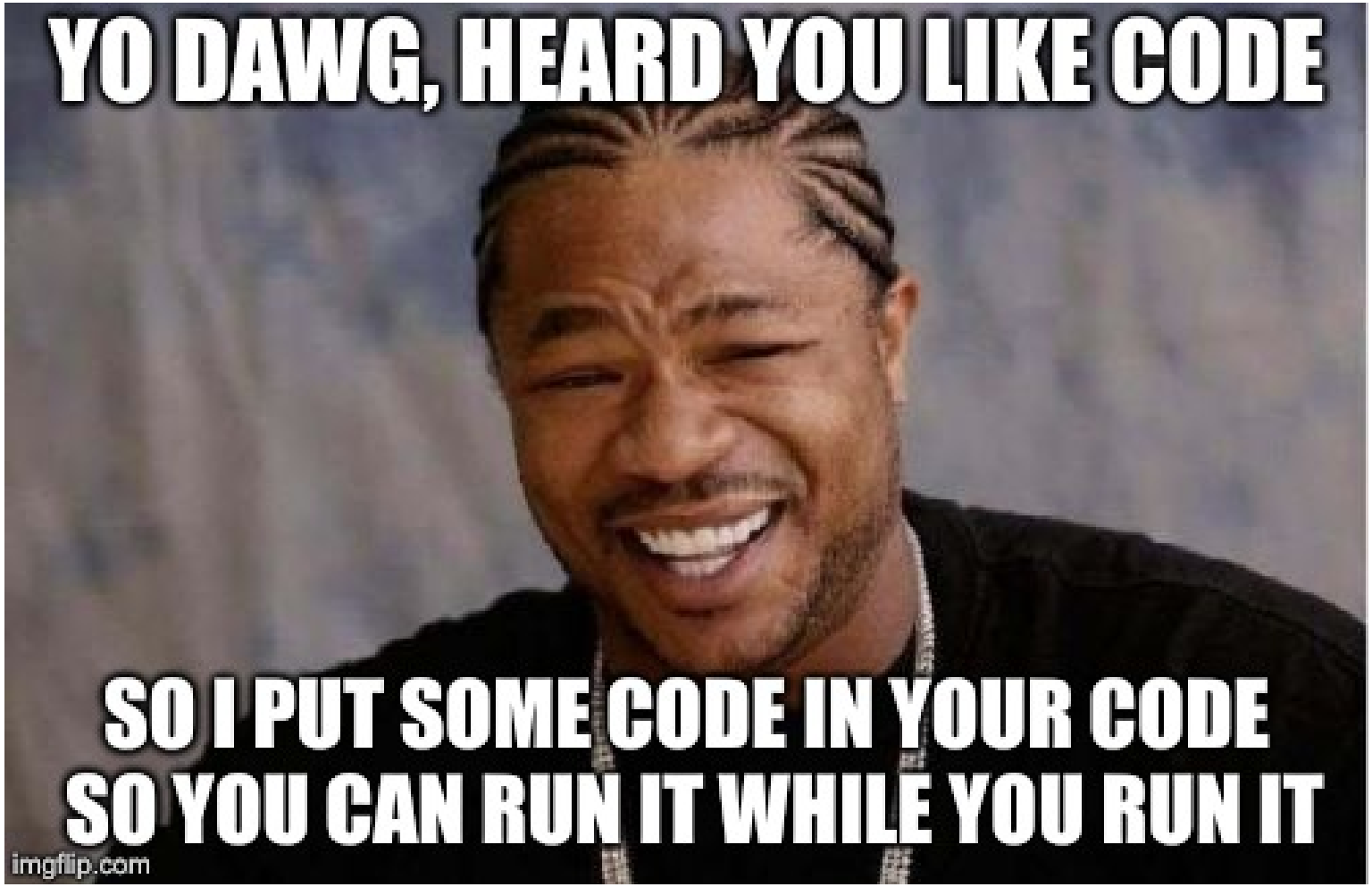
1. Compile/install AFL (once)
2. Compile target project with AFL
   - afl-gcc / afl-g++ / afl-clang / afl-clang++ / (afl-as)
3. Chose target binary to fuzz in project
   - Chose its command line options to make it run fast
4. Chose valid input files that cover a wide variety of possible input files
   - afl-cmin / (afl-showmap)
5. Fuzzing
   - afl-fuzz

AFL - Steps of fuzzing II

6. Check how your fuzzer is doing
   - command line UI / afl-whatsup / afl-plot / afl-gotcpu
7. Analyze crashes
   - afl-tmin / triage_crashes.sh / peruvian were rabbit
   - ASAN / valgrind / exploitable gdb plugin / …
8. Have a lot more work than before
   - CVE assignment / responsible disclosure / …

Instrumentation

What is Instrumentation I



What is Instrumentation II

- Is the capability to monitor and modify program behaviour during execution
- source code or binary
- static vs dynamic
- Source/Compile-time Instrumentation
  - instrument the source code of programs
- Binary Instrumentation
  - instrument executables directly

Instrumentation Example

- Assume the following code
  - Obvious problem if len $>$ 10

```
char num[10];
for(x=0;x<len;x++) {

  num[x] = x * 2;
}
```

Instrumentation Example

- Assume the following code
  - We could add debug output

```
char num[10];
for(x=0;x<len;x++) {
  printf("%d", x);
  num[x] = x * 2;
}
```

Instrumentation Example

- Assume the following code
  - Or checks

```
char num[10];
for(x=0;x<len;x++) {
  assert(x < 10);
  num[x] = x * 2;
}
```

Source-based Instrumentation

- Basically every compiler can do this/does this
  - GCC
  - LLVM + Clang
  - …
- E.g adding security checks to software during compilation
- Can also be used for deeper analyses or Bug Hunting

Example: *Sanitizers I

- Sanitizers that can be activated during compilation
- **Pro:** Can be used to detect several issues in the code
- **Con:** they add runtime overhead (slowdown, memory usage) to the binary
- Mostly only for testing, not for productive use

Example: *Sanitizers II

- AddressSanitizer (ASan)
  - memory error detector
    - Out-of-bounds accesses to heap, stack and globals
    - Use-after-free
    - Use-after-return
    - Use-after-scope
    - Double-free, invalid free
    - Memory leaks (experimental)
  - 2x slowdown typical
- ThreadSanitizer (TSan)
  - detects data races and deadlocks
  - 5x-15x slowdown
  - 5x-10x memory overhead

Example: *Sanitizers III

- MemorySanitizer (MSan)
  - detects uninitialized reads
  - 3x slowdown
- UndefinedBehaviourSanitizer (UBSan)
  - Detects undefined behaviour during execution
    - Using misaligned or null pointer
    - Signed integer overflow
    - Conversion to, from, or between floating-point types which would overflow the destination
- LeakSanitizer (LSan)
  - detects memory leaks during run-time

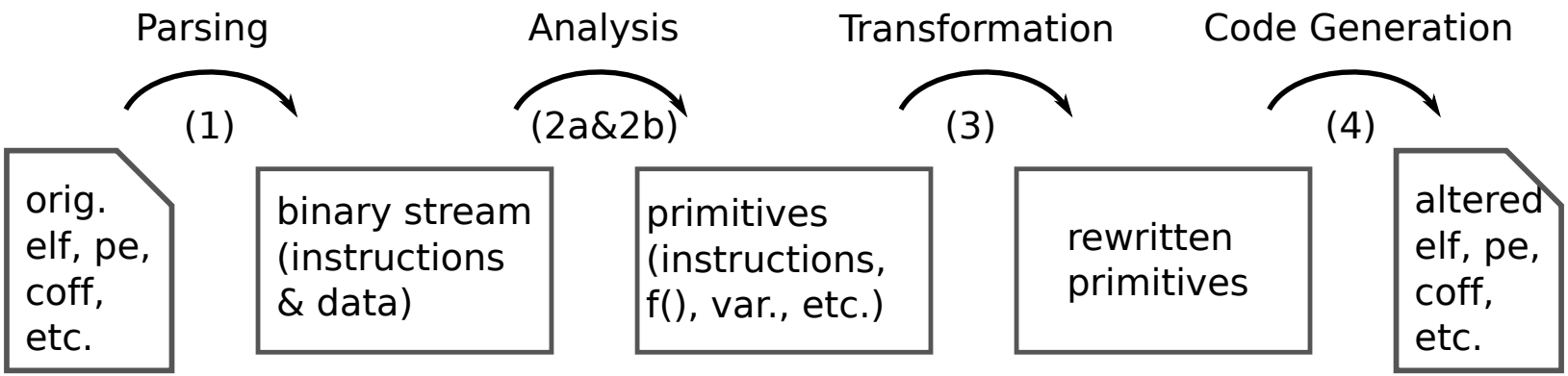
https://github.com/google/sanitizers

Binary Instrumentation

- Insert additional code into binary, without access to the source code
- No need to recompile or relink
- Static vs Dynamic Approaches

Static Binary Instrumentation

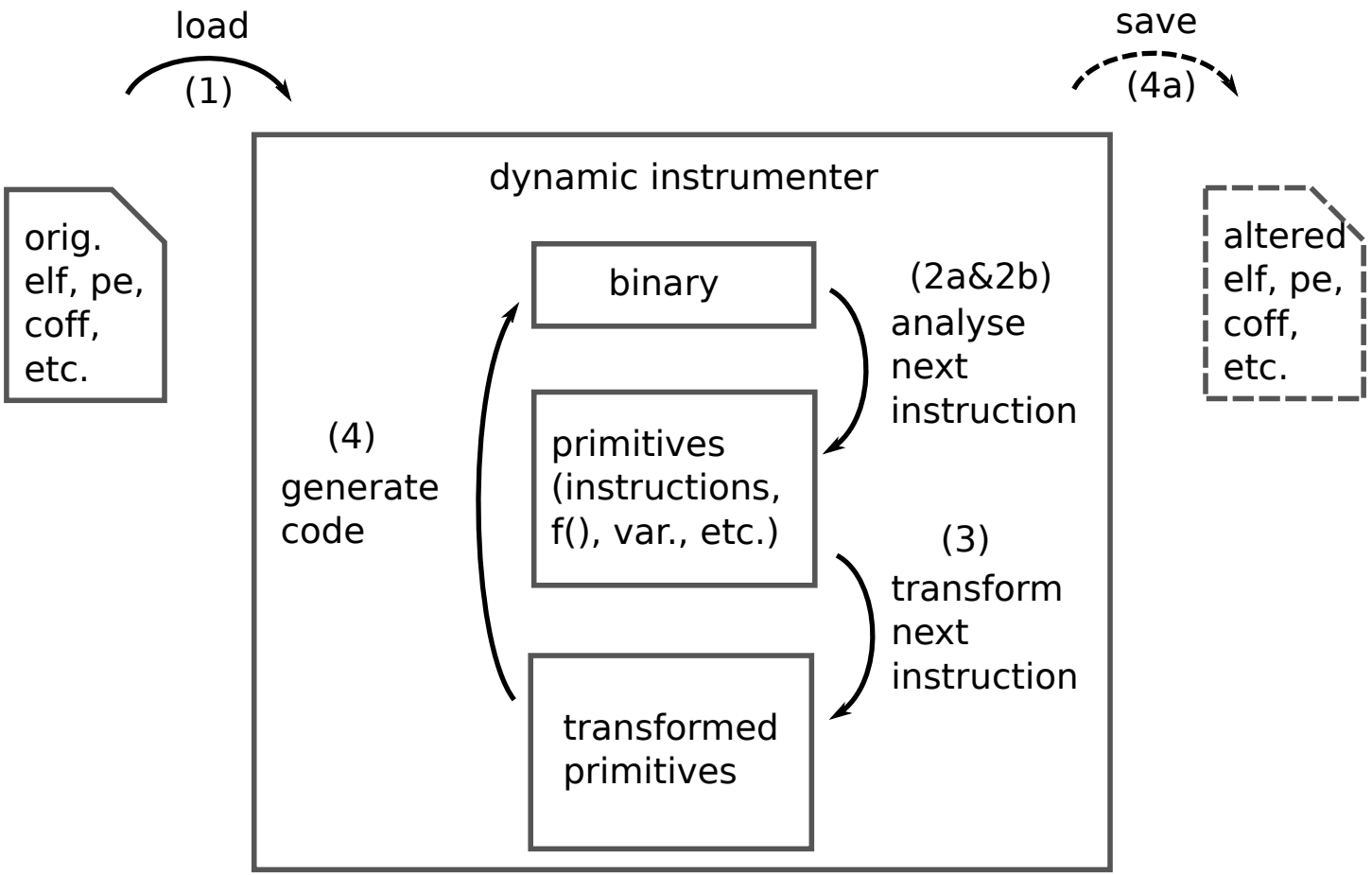
- Works without executing the binary
- Has to deal with all problems involved in static analysis



Dynamic **B**inary **I**nstrumentation (DBI)

- DBI allows you to insert additional code into a binary during runtime
  - “Hooking” instructions, functions,…
  - Execute your own code before and/or after instructions
- Advantages:
  - Discover Code at runtime
  - Handle dynamically-generated code
    - You can’t do this on the source
  - Attach to/instrument already running processes

DBI at a glance



DBI Use-Cases

- Instrument Binary to extract further information during runtime
  - Inject code into the binary
  - Monitor/Modify State during execution
  - Usually used for:
    - Simulation, Performance, Call graphs,…
  - We use it for:
    - Covert Debugging, Automated DeObfuscation, Taint Analysis, …

DBI Engines

- Several Engines out there
  - Pin
  - DynamoRIO
  - Valgrind
  - Frida
  - Unicorn
  - … (Many More)

Dynamic Taint Analysis

- Sometimes we can’t identify if data is interesting/sensitive when it is written or read
  - e.g., user inputs, contact book, …
- However, we know if data is sensitive when it’s used
  - e.g., change of control flow (return address, function pointer), information leaks, …

Dynamic Taint Analysis - Basic Idea

- Keep track of interesting information
  - (privacy-)sensitive information
  - input from untrusted sources
- Detect when data is used in a sensitive/untrusted way
- Label information with tags
  - e.g. trusted/untrusted, interesting/boring, public/secret
  - Control how the data and labels propagate
    - When copying the data also copy the tag
- Either on binary or source level
- Can be used to check where information is propagated/used in a program

Symbolic Execution

Based on: https://faculty.ist.psu.edu/wu/ist597a-ssa/Pinyao_040815.pdf

SAT/SMT Solvers

- Complex code / obfuscations
  - e.g. Obfuscated Checks of input
  - You want to know which input solves the Problem
  - Sometimes it is hard to understand/reverse the effects of the code
- You can use SAT/SMT Solvers to solve the problem for you
  - e.g. Z3 from Microsoft (there are also python bindings)
- But there is even more…

Symbolic Execution

- Based on Abstract interpretation
  - Usually binary is lifted to some intermediate representation (IR)
  - Reasoning on this abstract interpretation
  - Explore paths in parallel (beware: possible state explosion)
- Try to reason about what inputs will trigger different paths
  - Compared to random fuzzing
- We can use it to:
  - Identify inputs to trigger possible paths through the binary
  - or bug hunting
  - and more

Symbolic Execution

- values can be symbolic formulas over the inputs (not only concrete values)
- Track symbolic state rather than concrete input
- When execution path diverges, fork and add constraints on symbolic values

Goals

Identify semantics of a program

- detect infeasible paths
- generate test inputs
- generating program invariants
- debugging
- repair/patch programs
- finding bugs and vulnerabilities

Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```

Symbolic Execution Example

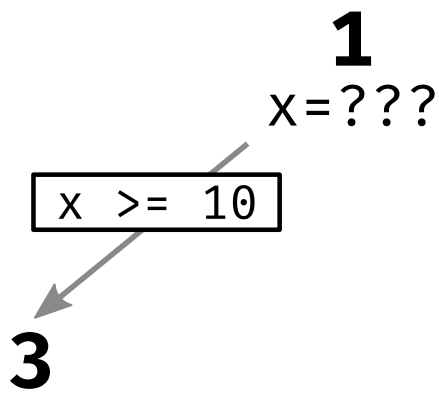
```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```

1

x=???

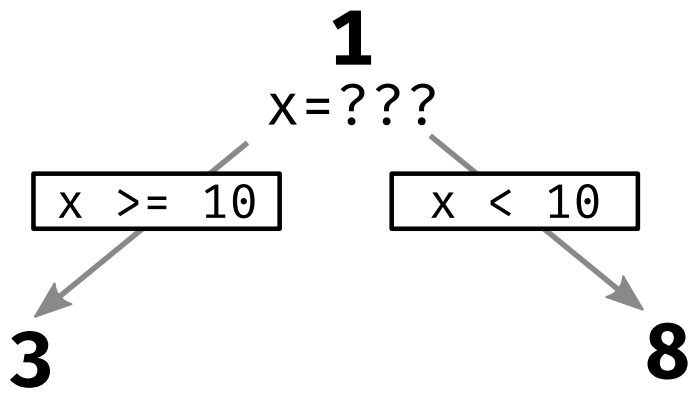
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



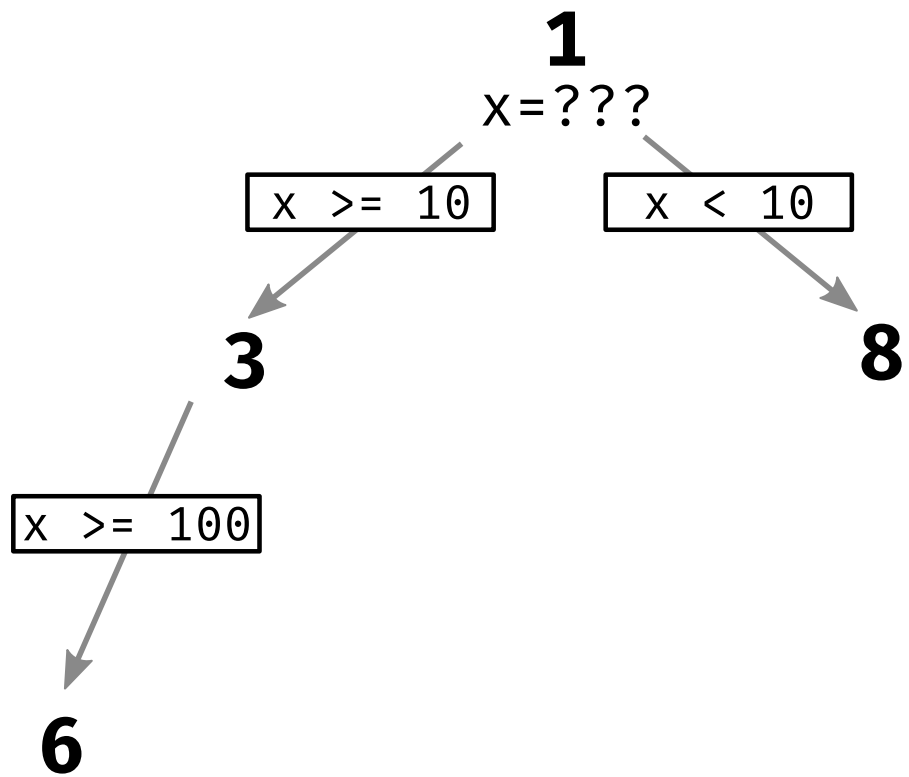
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



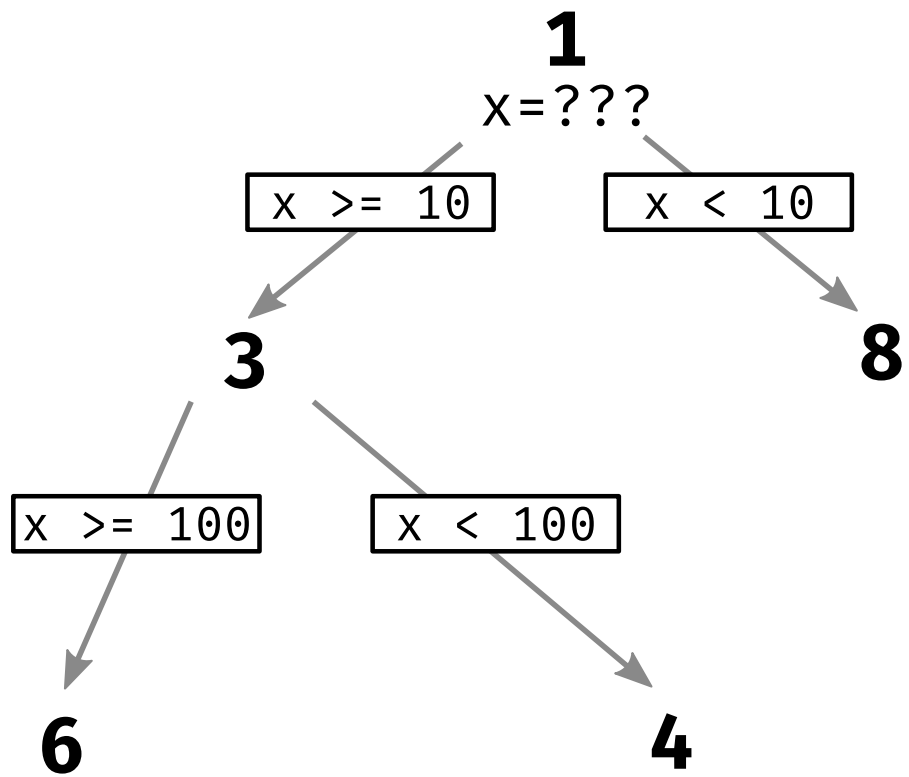
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



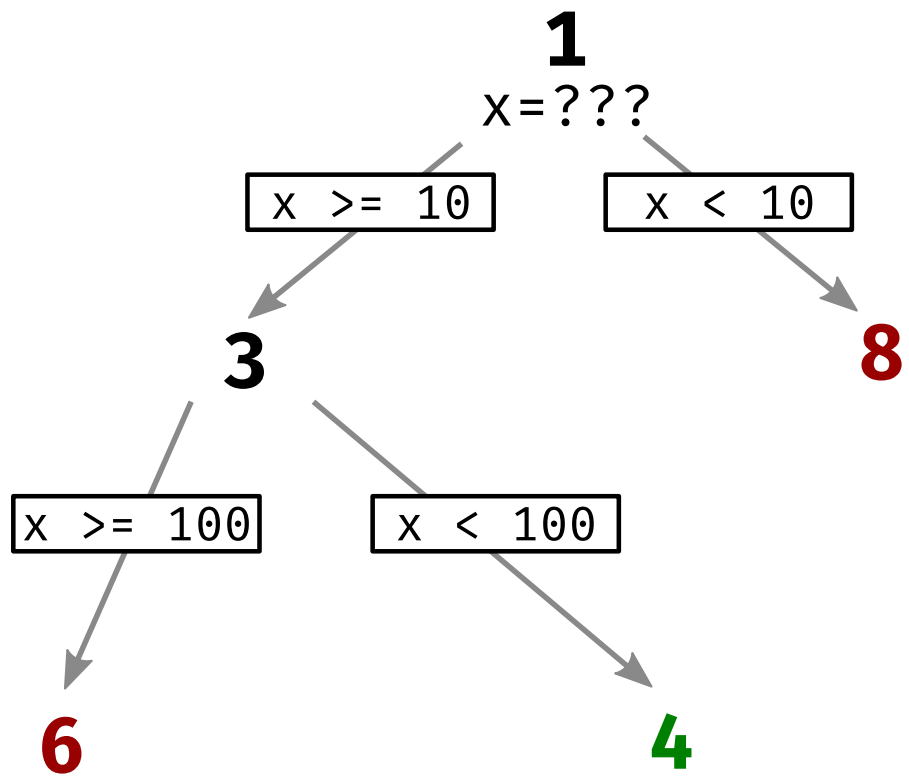
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



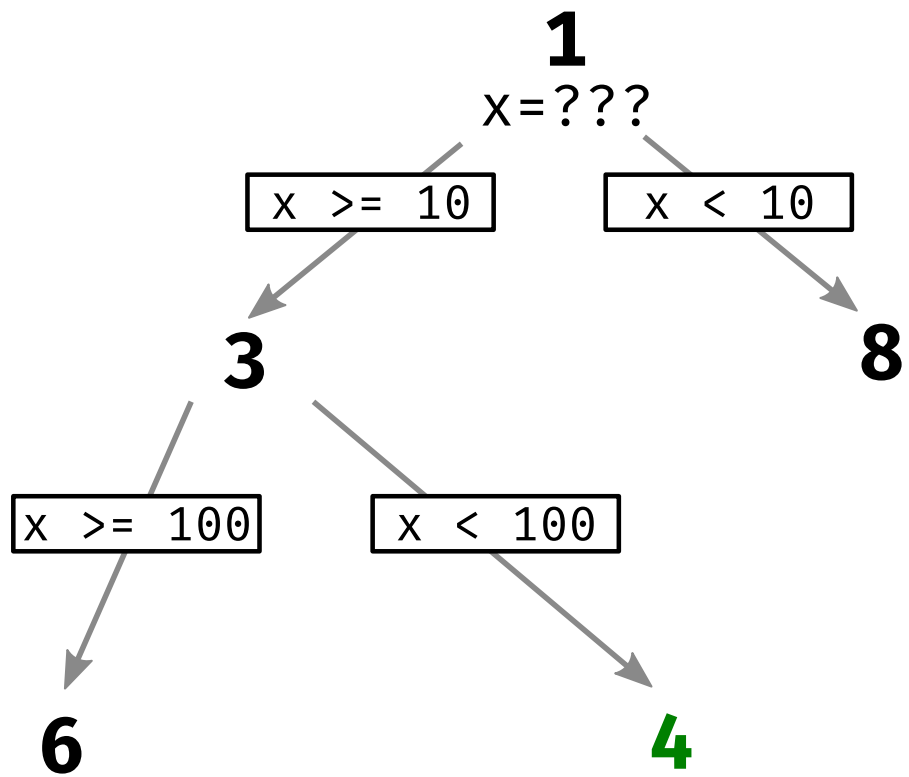
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



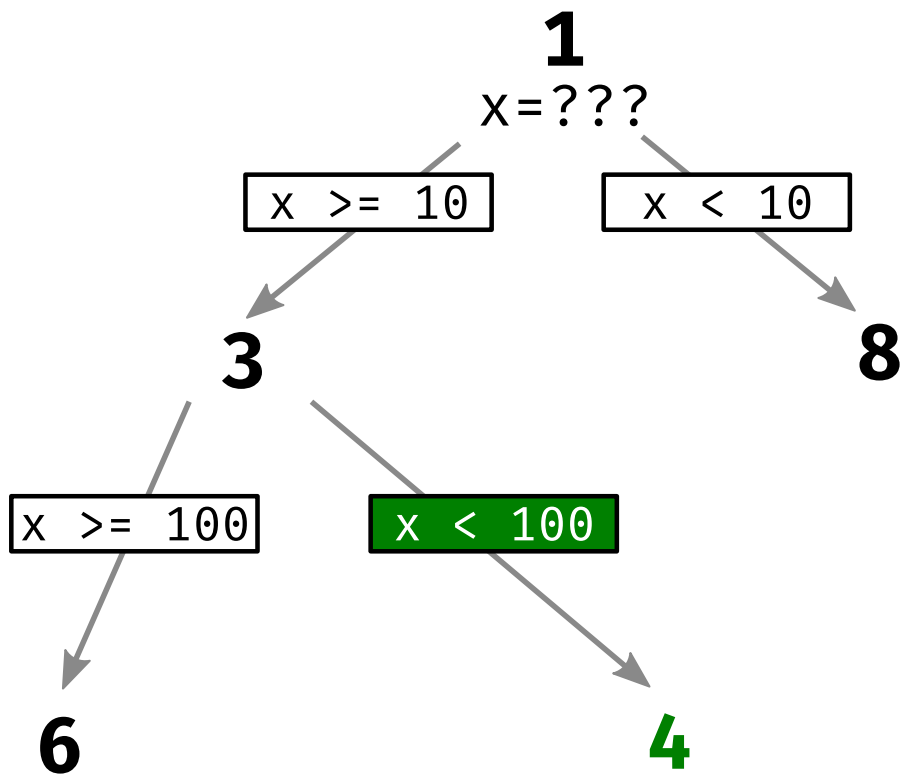
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



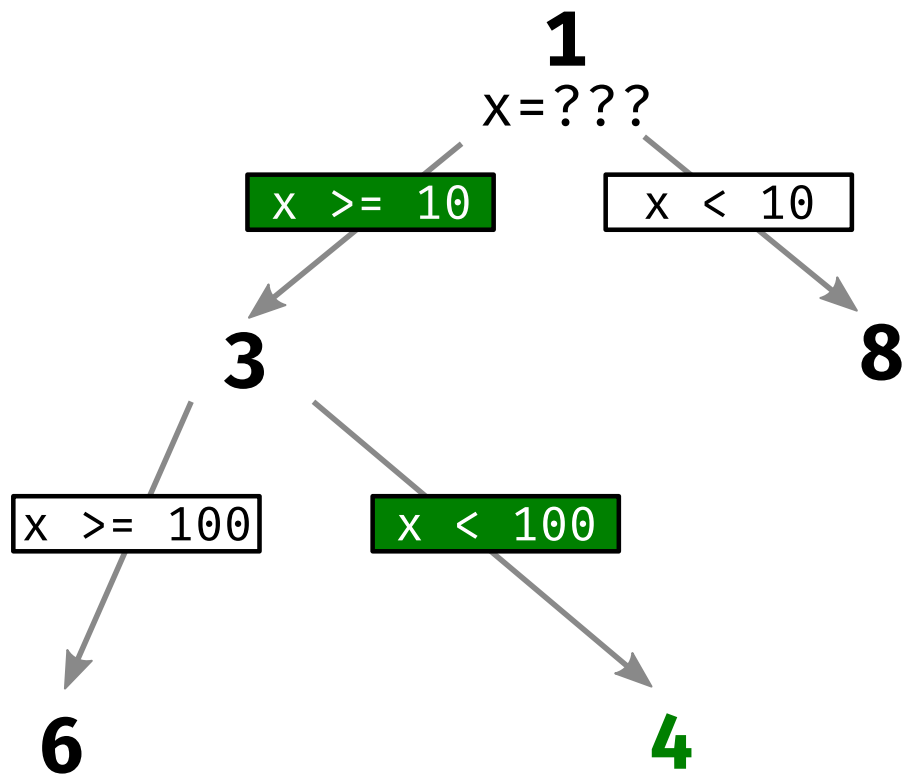
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



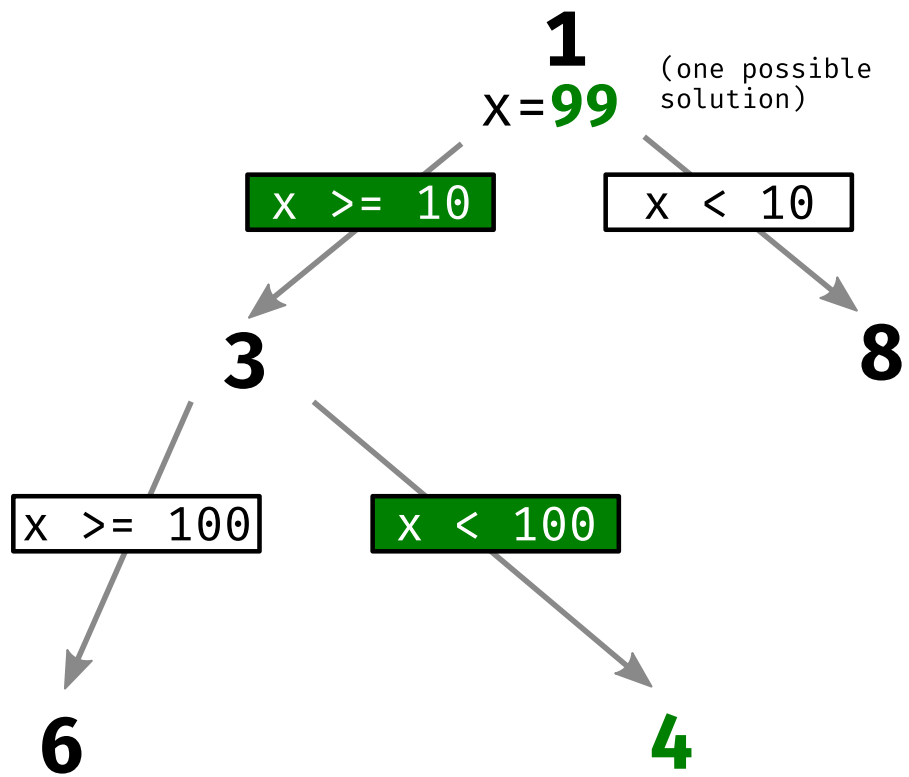
Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



Symbolic Execution Example

```
1:   x = int(input())
2:   if x >= 10:
3:     if x < 100:
4:       print "You win!"
5:     else:
6:       print "You lose!"
7:   else:
8:     print "You lose!"
```



Challenges

- Path explosion
  - we can hit it with better computers and more hardware
  - mix symbolic with concrete execution
- Modeling program statements and environment handling
- Powerful constraint solvers

Path explosion: Branches

```
1        if(input()==true){
2            x = x+1;
3        }
4        if(input()==true){
5            x = x+2;
6        }
7        if(input()==true){
8            x = x+4;
9        }
10       assert(x <= 7);
```

- How many paths are there?

Path explosion: Branches

```
1        if(input()==true){
2            x = x+1;
3        }
4        if(input()==true){
5            x = x+2;
6        }
7        if(input()==true){
8            x = x+4;
9        }
10       assert(x <= 7);
```

- How many paths are there?
  - 2^3

Path explosion: Branches

```
1        if(input()==true){
2            x = x+1;
3        }
4        if(input()==true){
5            x = x+2;
6        }
7        if(input()==true){
8            x = x+4;
9        }
10       assert(x <= 7);
```

- How many paths are there?
  - 2^3
  - Exponential in branching structures

Path explosion: Loops

```
1        int x=input()
2        while (x){
3           ... // do something
4        }
```

- Loops are also a problem

Path explosion: Loops

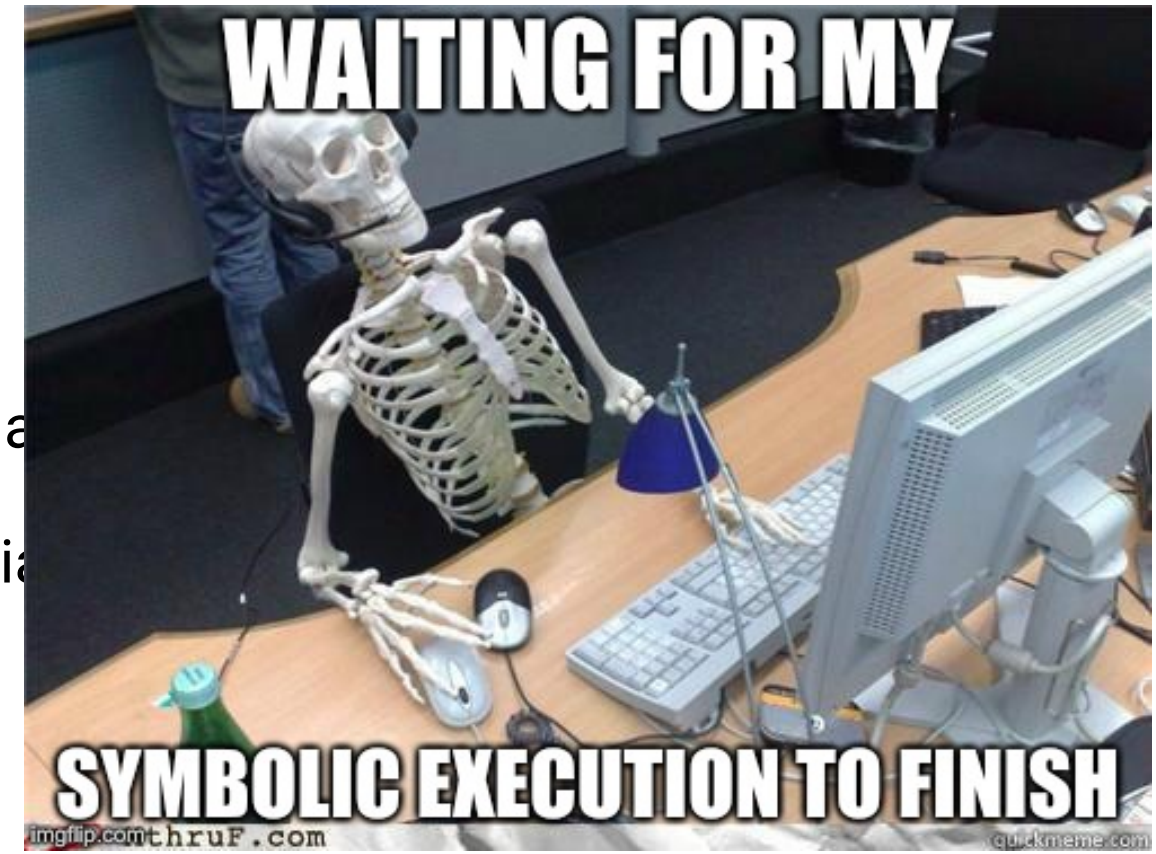
```
1        int x=input()
2        while (x){
3           ... // do something
4        }
```

- Loops are also a problem
- Potentially infinite number of iterations

Path explosion: Loops

```
1
2
3
4
```

- Loops a
- Potentia

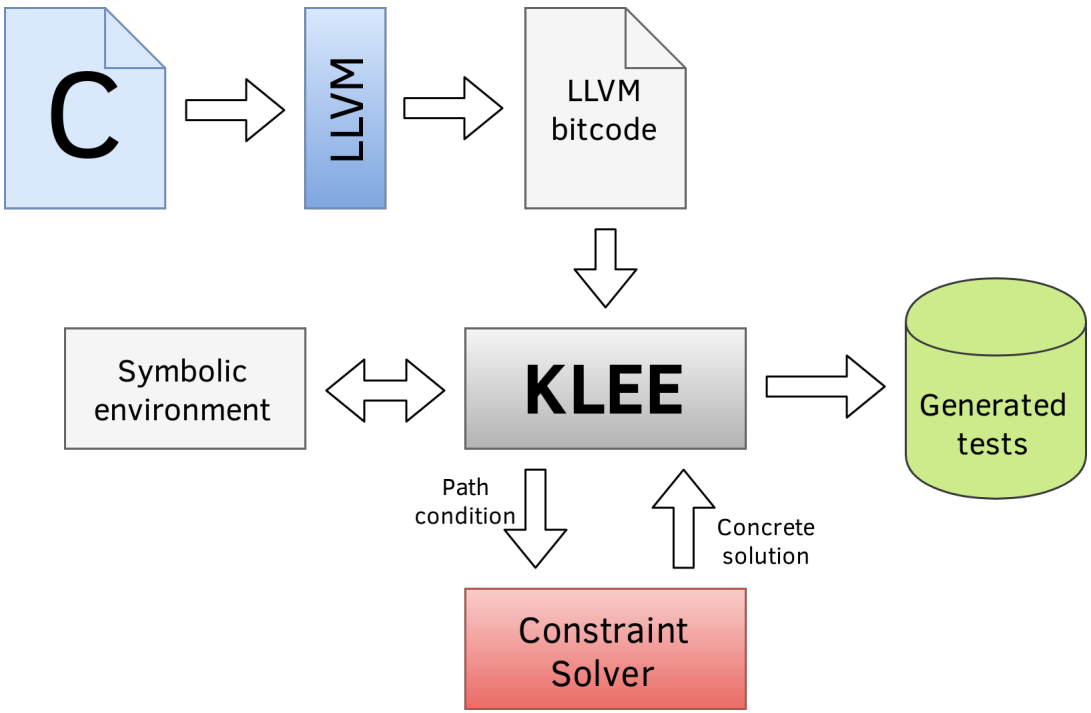


Environment Handling: the Problem

- System/Library Calls
  - Inputs/outputs need to be included into analysis
- Pointers and Memory
  - Analysis might need to be aware of the memory layout (e.g. heap)
- Input data (files, stdin) and command line arguments

Tool Example: KLEE

- Symbolic Virtual Machine based on LLVM compiler infrastructure
- First widely used symbolic execution tool



https://klee.github.io/

KLEE - Path explosion

- Provides several strategies to mitigate the problem
- DFS
  - search can get stuck in long running loops
- BFS
  - very slow to determine properties for a path with many branches
- Random search
  - reproducibility problem
- Coverage guided search
  - tries to reach everything, but might never be able to get to certain statements

KLEE - Environment handling

- Create simple versions of library/system calls or concretize values
  - for libc call: KLEE compiles the uclibc and links to the target program
  - other library calls: concretize symbolic value and use JIT to do library call
- Limit loop execution
  - execution time
  - instructions
- Simulate system calls
  - e.g. for file system calls exists a simple file system model in KLEE

Concolic Execution

- A combination between “**Conc**rete” and “Symb**olic**” Execution
  - Or dynamic symbolic execution
- Uses both techniques to solve a constraint path
  - Symbolic execution creates new concrete inputs to maximize code coverage
- Instrument Program to do symbolic execution as program runs
  - Shadow concrete program state with symbolic variables
- Explore one path at a time, start to finish
  - Can always rely on a concrete underlying value
- Can be used for Bug hunting, automatic exploit generation,…

Concolic Execution - Benefits

- Solve complex formulas
  - `x == (y*y) mod 50`, unsolvable if both x and y are symbolic
  - if we concretize y to its concrete value, now solvable
  - Angr does this!
- External library call and system call
  - E.g., `fd=open(filename)`
  - Set filename to its concrete value “/tmp/abc.txt”
  - Execute the system call concretely
  - Set fd to be concrete after the system call return
  - High level idea of S2E!

Online vs Offline

- Online
  - When encounter a new symbolic branch, solve predicates for both directions
  - If both directions are feasible, fork the execution state (concrete and symbolic)
- Offline (or trace-based)
  - Choose an input and execute the program, collect execution trace
  - Compute path constraints from the trace
  - Negate each conjunct, solve the new path constraint, and get a new input
  - Given the new input to the program and execution again

Online vs Offline

	Online	Offline
Efficiency	High	Low
Implementation difficulty	High	Low
Symbolic State	Quickly explodes	No state management

How to execute symbolically? I

- Trace based
  - BAP: Use Pintraceto collect execution trace, and then convert the trace into BAP IL (derived from VEX)
  - BitBlaze: Use tracecapplugin to collect execution trace, Convert the trace into Vine IR
  - Low efficiency and possibly very long trace!
- Dynamic Instrumentation
  - S2E:
    - Run in QEMU with two machines (concrete and symbolic) simultaneously
    - Convert TCG IR to LLVM Bitcode
  - KLEE:
    - Compile C/C++ into LLVM Bitcode
    - Add instrumentation on LLVM Bitcode

How to execute symbolically? II

- Complete Interpretation or Simulation
  - Interpret binary execution and add symbolic execution
  - **Angr:** convert each instruction into VEX, and interpret each VEX statement in Python
  - **Pro:** full control, easy to implement
  - **Con:** low efficiency by nature. All instructions must be interpreted, no matter if symbolic variables are involved or not. For long execution trace, it will take very long time!

How to deal with state explosion

- State merging and pruning
- Targeted search
  - find an interesting target
  - at each branch point, take the direction “closer” to the target
  - need to define “closer” through fitness function (e.g. distance in terms of edges in the CFG)
- Combine online and concrete re-execution
  - e.g., Mayhem
- Combine symbolic execution with evolutionary fuzzing
  - e.g., Driller

Mayhem

Combine online symbolic execution and concrete re-execution

- Perform online symbolic execution in BFS fashion
- When it reaches a limit, store the symbolic states on disk
- Pick one state to continue. To do so, solve the path constraint, and use it as input to re-execute the program up to the current state
- Start to perform online execution from this state

Driller

Combine symbolic execution with evolutionary fuzzing

- Evolutionary fuzzing drives the path selection
  - AFL
  - Share the seeds with symbolic execution
- Symbolic execution takes each seed and perform a very localized path exploration
  - Angr
  - Generate new inputs and feed them back to the fuzzer
- Problems
  - Most of these new inputs will be unfortunately dropped
  - Some seeds lead to very long trace, take very long time to execute in Angr, and impossible to solve

Tool Example: Angr

- Control-flow graph recovery.
- Symbolic execution.
- Automatic ROP chain building using angrop.
- Automatically binaries hardening using patcherex.
- Automatic exploit generation (for DECREE and simple Linux binaries) using rex.
- Use angr-management, a (very alpha state!) GUI for angr, to analyze binaries!
- Achieve cyber-autonomy in the comfort of your own home, using Mechanical Phish, the third-place winner of the DARPA Cyber Grand Challenge.

https://angr.io/

Summary

1. Binary Analysis Recap
2. Fuzz Testing (Fuzzing)
3. Instrumentation
4. Symbolic Execution

Next Challenge Starts Tomorrow

References I

- Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.”. In: *OSDI*. Vol. 8. 2008, pp. 209–224.
- Sang Kil Cha et al. “Unleashing mayhem on binary code”. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 380–394.
- Pinyao Guo. *Symbolic execution*. URL: https://faculty.ist.psu.edu/wu/ist597a-ssa/Pinyao_040815.pdf.
- Yan Shoshitaishvili et al. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy*. 2016.

References II


Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.”. In: *NDSS*. Vol. 16. 2016, pp. 1–16.


Dmitry Vyukov. *Fuzzing - the new unit testing*. URL: https://github.com/GopherConRu/talks/blob/master/2018/Fuzzing%20-%20the%20new%20unit%20testing%20-%20Dmitry%20Vyukov.pdf.


Heng Yin. *CS 260-001*. URL: https://www.cs.ucr.edu/~heng/teaching/cs260-winter2017/index.html.