Thinking on Uses of Dynamic Analysis for Software Security

ben-holland.com

$ whoami

- 2005 – 2010
  - B.S. in Computer Engineering
  - Wabtec Railway Electronics, Ames Lab (DOE), Rockwell Collins: Software Engineer Intern
- 2010 – 2011
  - B.S. in Computer Science
  - Rockwell Collins: Software Engineer Intern
- 2010 – 2012
  - M.S. in Computer Engineering (Co-major Information Assurance)
  - Thesis: Enabling Open Source Intelligence (OSINT) in private social networks
  - MITRE: Software Engineer Intern
- 2012 – 2015
  - Iowa State University: Research Associate → Assistant Scientist
  - DARPA's APAC and STAC programs
    - Demands impactful and practical software solutions for open security problems
    - Fast-paced, high-stakes, adversarial engagement challenges
- 2015 – 2018
  - Ph.D. in Computer Engineering (Iowa State University)
- 2019 – Present
  - Apogee Research: Senior Research Engineer
  - We are hiring! Online at: apogee-research.com

Disclaimer

- Nobody is endorsing me to say any of the things I am about to say
- I am not representing my employer (but we are hiring!)
- What I am going to say is my opinion and may be controversial among experts
- I am somewhat unavoidably biased towards certain approaches
- I’ll probably ask more questions than I have answers
- I’ll probably even get a few things wrong…

Overview

- What is a program?
- Why do we need program analysis?
- What is dynamic analysis?
- What is the state-of-the-art dynamic analysis?
- How can we do better?

What is a program?

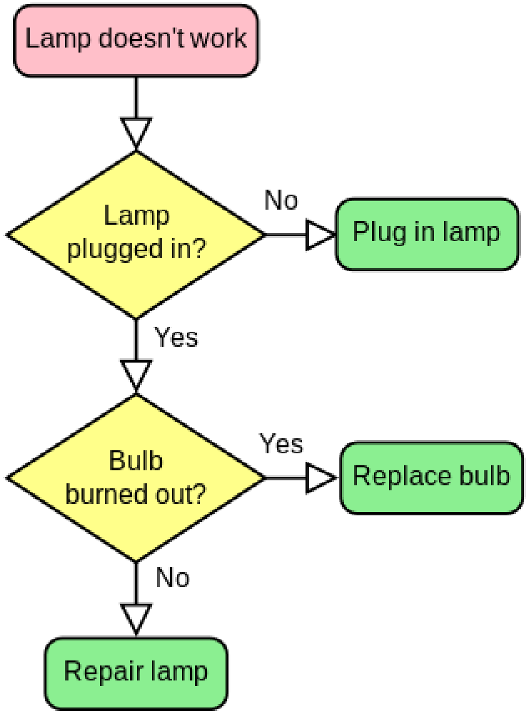
Ice Breaker Exercise: EIL5 “Programming”

- Explain It Like I’m Five (EIL5): What is a computer program?
- Can your explanation intuitively address:
    - What is a program
    - What are the inputs and outputs
    - Complexity of software
    - Programming bugs
    - Security issues

What is a program?

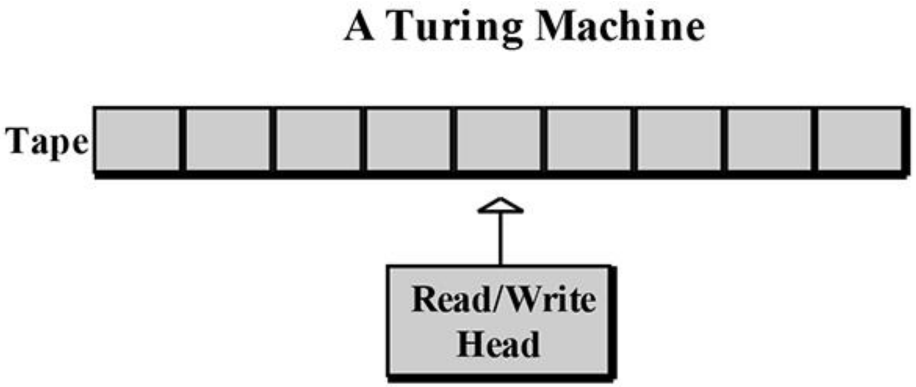
- Common answer: “a set of instructions”
- Better answer: “similar to a cooking recipe”
    - Ordered list of instructions
    - Instructions executable by a cook (i.e. the computer)
    - Instructions specify operators (actions) and operands (data)
        - Example: “add flour to bowl”
        - Operator: *add*
        - Operands: *flour*, *bowl*
    - Instructions can be branching or non-branching
        - Non branching: “add flour to bowl”
        - Branching: *if* “large batch” *then* “add flour to bowl”
    - Instructions can be repeated (i.e. loop)
        - Example: *jump* to first instruction
        - Example: *while* “batter is runny” then “stir batter”

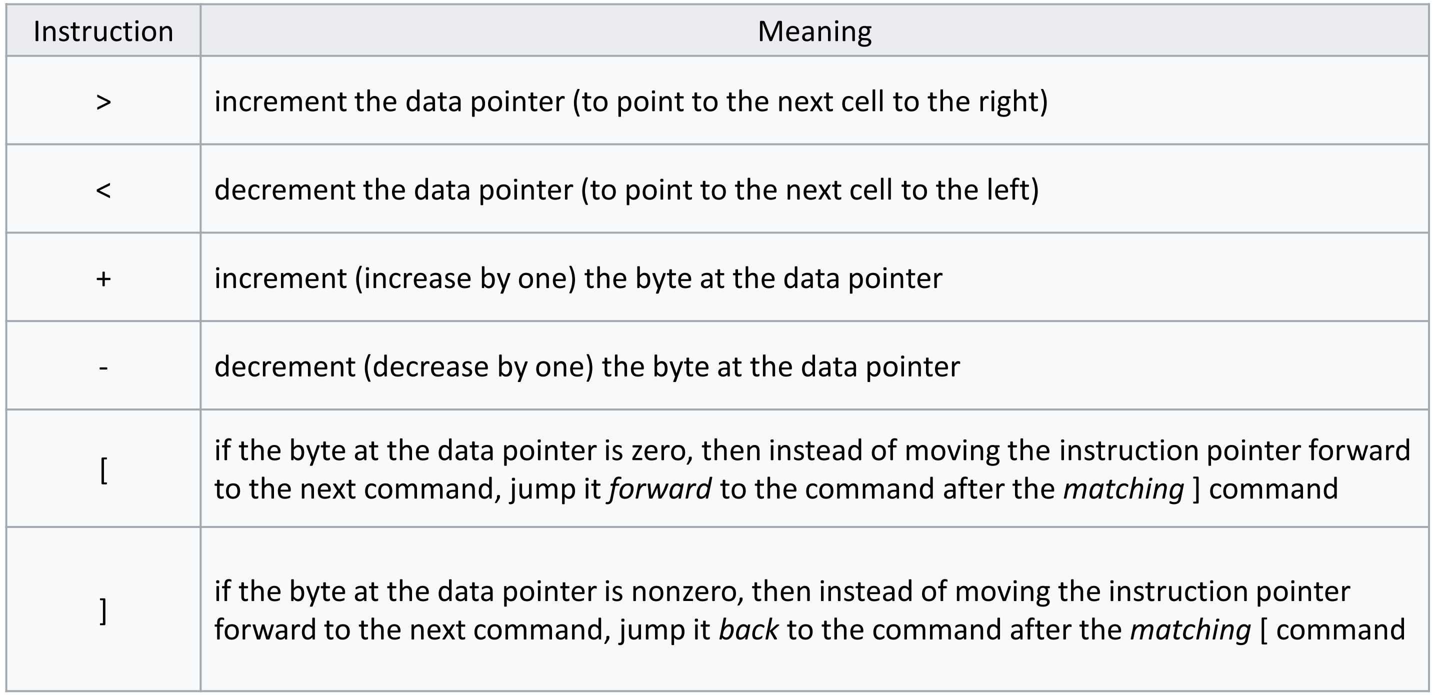
We can visualize programs as flow charts



What is a program?

- Even better answer: Something that can be translated to a set of low level instructions (e.g. Brainf*ck) that control a Turing machine
  - Program: Series of BF instructions
  - Input: Contents on tape
  - Output: Contents on tape

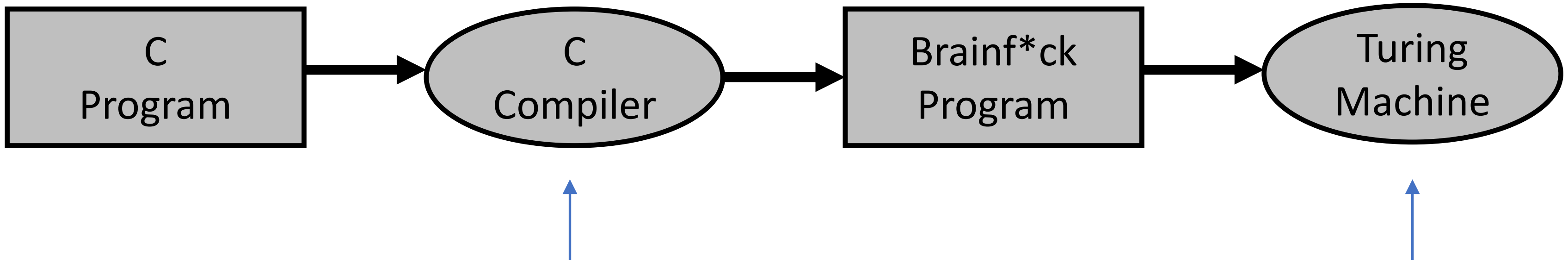




Instruction	Meaning
>	increment the data pointer (to point to the next cell to the right)
<	decrement the data pointer (to point to the next cell to the left)
+	increment (increase by one) the byte at the data pointer
-	decrement (decrease by one) the byte at the data pointer
[	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it *forward* to the command after the *matching*] command
]	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it *back* to the command after the *matching* [command

What is a program?

- Even better answer: Something that can be translated to a set of low level instructions (e.g. Brainf*ck) that control a Turing machine



C to Brainf*ck Compiler

- https://github.com/arthaud/c2bf
- https://www.codeproject.com/Articles/558979/BrainFix-the-language-that-translates-to-fluent-Br

x86 interpreter implemented exactly 100 bytes
https://github.com/peterferrie/brainfuck

Why do we need program analysis?

Why do we need program analysis?

- While humans are currently writing software for machines, it is hopeless for humans alone to audit software at scale
  - Programs have a *staggering* amount of complexity
  - We have *a lot* of programs
  - Programs are changing at a *ridiculous* pace
  - Programs are *infested* with bugs that can last *years*
  - We *still* haven’t learned how to write *correct* software

Programs have a *staggering* amount of complexity

- Branches introduce multiple paths (behaviors) for a program
  - Visually think about each path you could take in a flow chart of the program
- Hypothesis: There are more paths in the Linux kernel than there are atoms in the known universe *(spoiler alert: there are actually many more paths!)*
  - Known universe spans 93 billion light years
  - Estimated to have 500 billion galaxies each with approximately 400 billion stars
    - Estimated that 120 to 300 sextillion (1.2 x 10^{23} to 3.0 x 10^{23}) stars exist
  - On average, each star can weigh about 10^{35} grams
    - Each gram of matter is known to have about 10^{24} protons, or about the same number of hydrogen atoms (since one hydrogen atom has only one proton)
    - Gives us a *high* estimate of atoms in known universe is 10^{86} (one-hundred thousand quadrillion vigintillion)
  - When it sounds like a 1st grader is just making up numbers, then you know it is a big number!

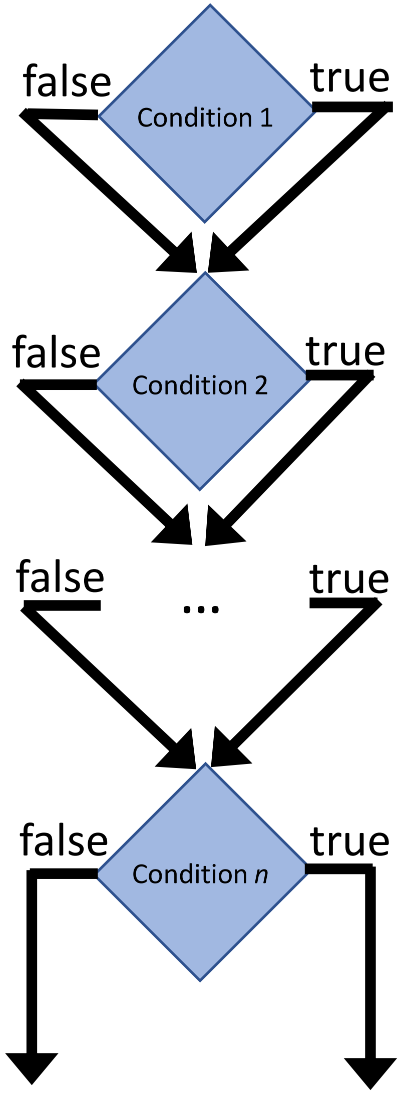
Source: https://www.universetoday.com/36302/atoms-in-the-universe/

Challenge: Path Explosion Problem

- Remember we can draw software as a flow chart…
- A single function in the Linux kernel (*lustre_assert_wire_constants*) has 2^{656} paths with no loops involved!
  - Only 10^{86} atoms in the known universe…
  - $2^{656} \approx 10^{197}$
- Paths are multiplicative across functions…
- Loops test the limits of human comprehension…

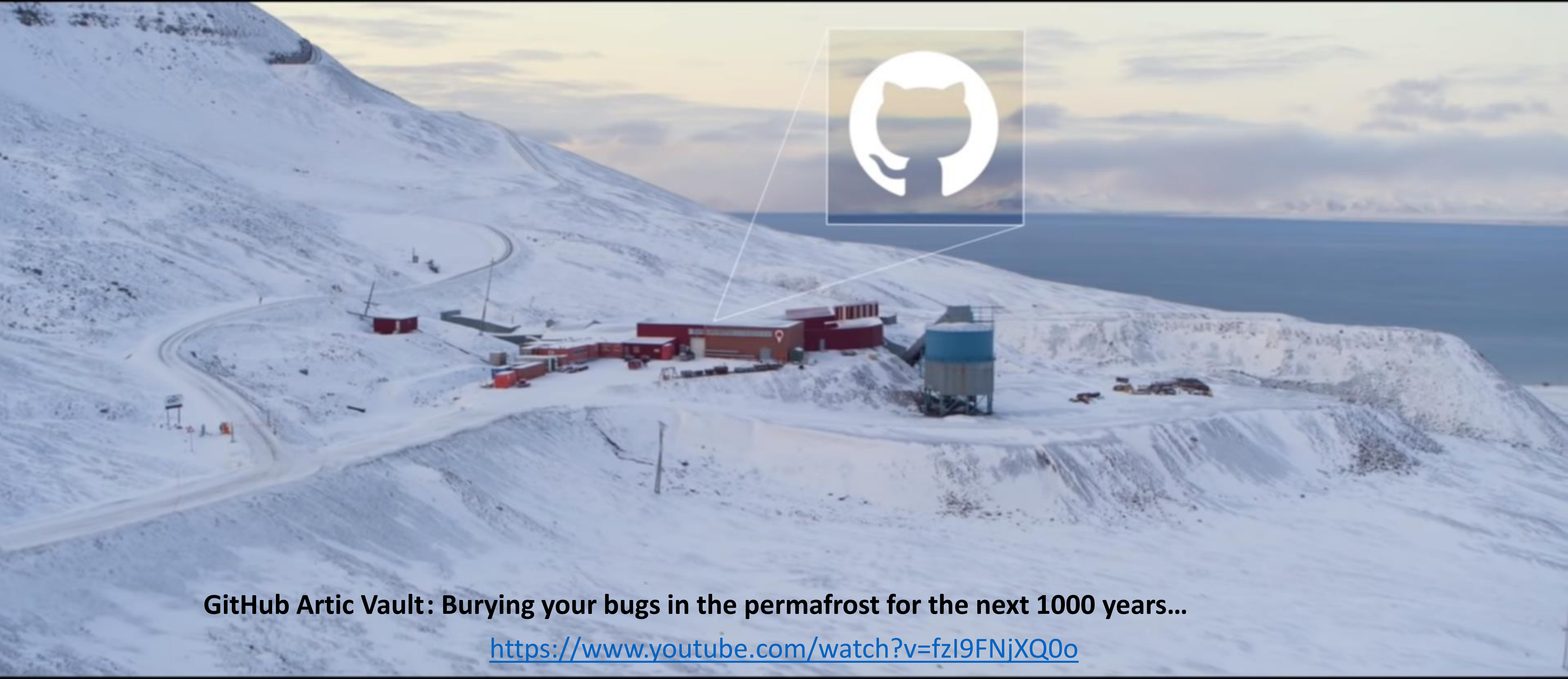
2^n paths!

```
if(condition_1){
   // code block 1
}
if(condition_2){
   // code block 2
}
if(condition_3){
   // code block 3
}
…
if(condition_n){
   // code block n
}
```



We have *a lot* of programs

- Truly we have no idea how many programs there are since software is absolutely ubiquitous
    - Over 700 fully featured programming languages [1]
    - GitHub reached 100 million open source repositories of code in 2018 [2]
    - Estimated that we write 111 billion new lines of code every year [7]
    - Enough programs that GitHub plans to archive source code at the North Pole [3]

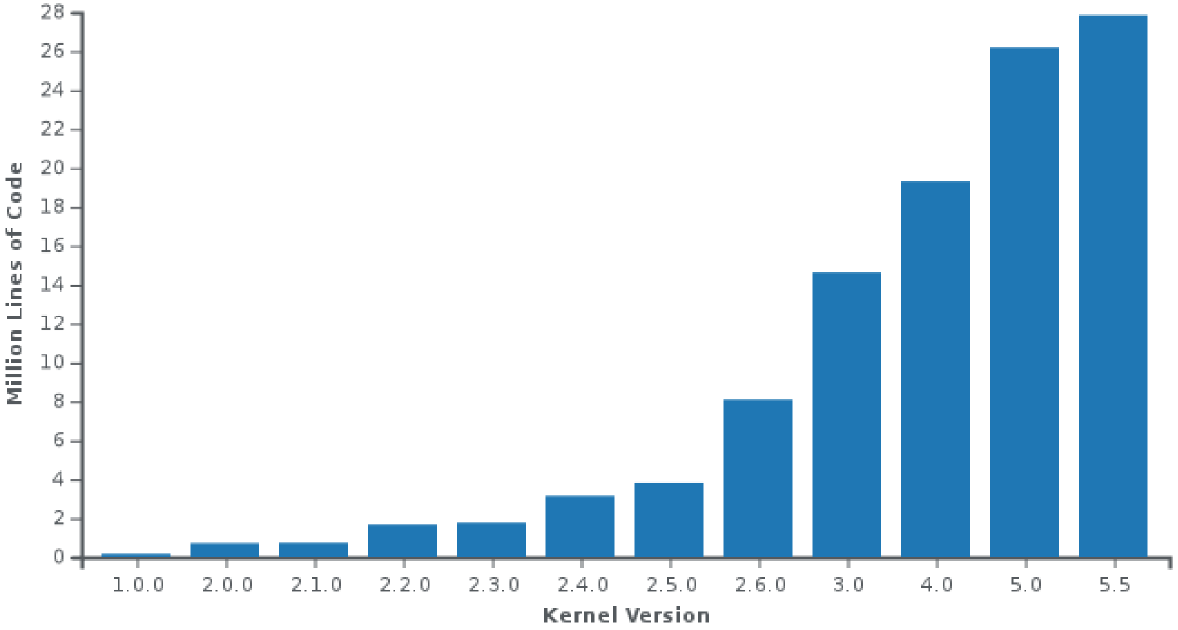


GitHub Artic Vault: Burying your bugs in the permafrost for the next 1000 years…

https://www.youtube.com/watch?v=fzI9FNjXQ0o

Programs are changing at a *ridiculous* pace

- Just the Linux kernel has:
    - 2,246 lines of code changed per day [4]
    - 19,093 lines of code added per day (795 lines added per hour) [4]
    - 19,093 lines of code removed per day [4]
    - Code contributions from over 15,000 developers and 500 companies as of 2017 [5]



Source: https://en.wikipedia.org/wiki/Linux_kernel

Programs are *infested* with bugs that can last *years*

- Software remains infested with bugs creating security vulnerabilities
  - Industry average of 10 to 50 defects per 1,000 lines of code [16]
  - A vulnerability lives in a codebase for an average of 438 days before it is discovered [8]
    - Shellshock was discovered 25 years later after it was created!
  - Zero-day attacks go undetected for an average of 312 days before discovery [9]
  - A security patch is created on average 27 days before the vulnerability is disclosed [8]
    - Organizations take an average of 100-120 days to patch a vulnerability [10]
    - Highest average remediation time of 176 days for financial organizations [13]
  - Exploits have appeared as quickly as 3 days following disclosure [12]
    - Average life expectancy of an exploit is 6.9 years [11]
  - The probability that a vulnerability will be exploited during the first 40-60 days (well before the average remediation period) following disclosure is over 90% [10]

We *still* haven’t learned how to write *correct* software

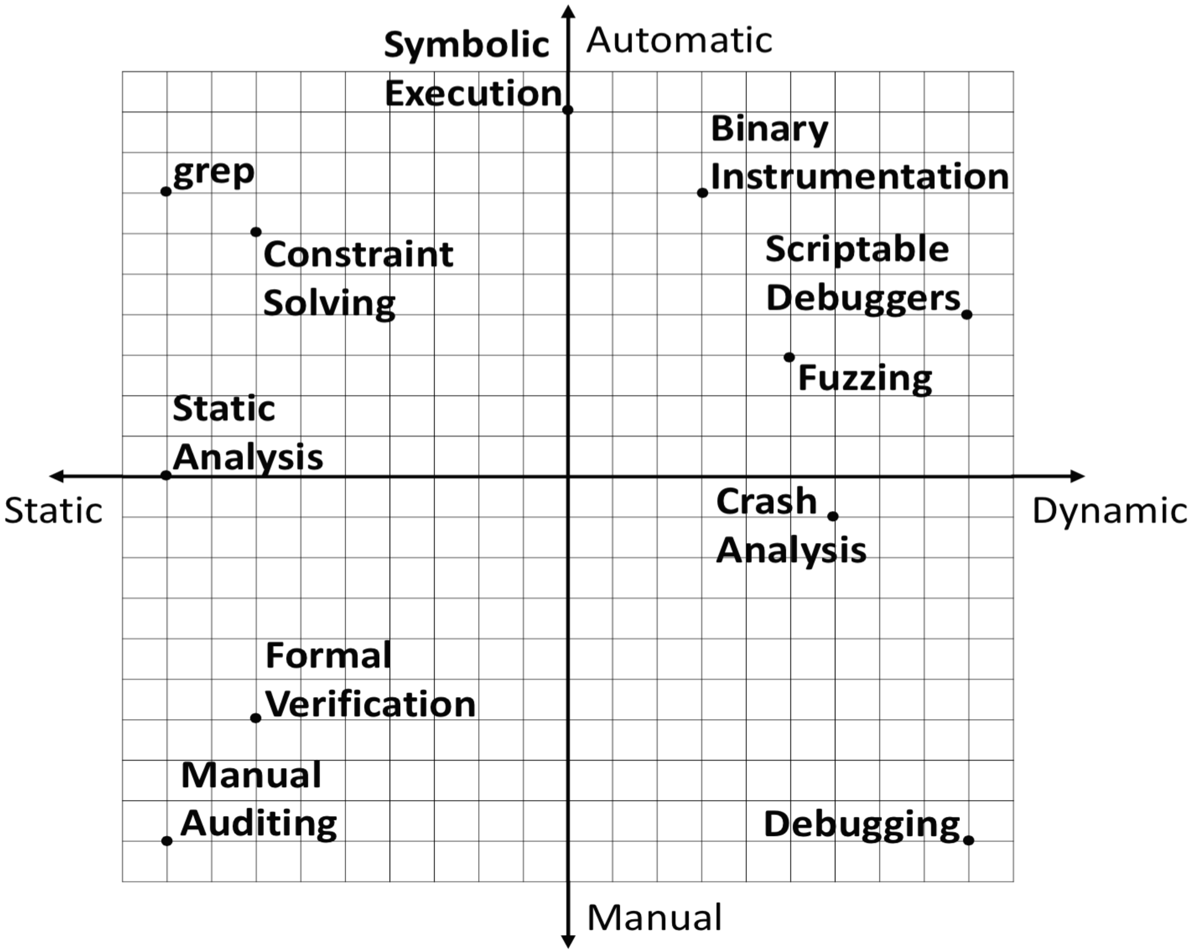
- We keep making the same mistakes…
  - 15-25% of all bug patches in Linux kernel were themselves buggy [14]
  - ~85% of all high severity Android vulnerabilities were violations of low-level data structures [15]
  - 24.24% of all high and critical severity CVEs between 2002-2019 were due to buffer bound issues (my analysis of MITRE CVEs grouped by NIST CWE tags)
    - Buffer overflows vulnerabilities first documented in 1972
    - “Smashing The Stack For Fun and Profit” was published in 1996

What is dynamic analysis?

How do we analyze a program?

- Two main approaches:
    - Static analysis
        - Don’t run the program, dissect the logic and examine program artifacts
        - Advantage: Bird’s eye view of everything that could possibly happen during execution
        - Concern: Number of program behaviors is HUGE
        - Concern: Is it feasible to reach/trigger an artifact of concern?
    - Dynamic analysis
        - Run the program with some inputs and see what it does
        - Advantage: Everything we observe is feasible (we just saw it happen)
        - Concern: Input space is HUGE
        - Concern: Did we test the *interesting* inputs?
- What are we looking for?
    - Bugs: Memory corruption, rounding errors, null pointers, infinite loops, stack overflows, race conditions, memory leaks, business logic flaws, …
    - Not every issue translates to a crash!

A Spectrum of Program Analysis Techniques



Source: Contemporary Automatic Program Analysis,
Julian Cohen, Blackhat 2014

Key Questions for Dynamic Analysis

- What is monitored in the program?
- How are program inputs generated?
- What is being searched for?
- What is executed in the program?

What is monitored in the program?

What is monitored in the program?

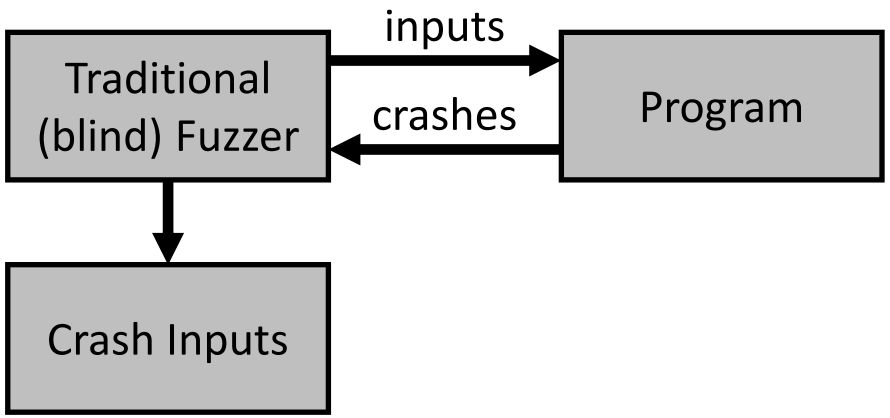
- Blackbox
    - No knowledge of program internals or state
    - Only monitoring inputs/output values or environment changes (e.g. memory)
- Graybox
    - Graybox we can look at *some* parts of the program (e.g. values at branches)
- Whitebox
    - Whitebox we can look at *all* of the program (e.g. we have source code or we can access any of the binary code)

Blackbox Fuzzing

How are inputs generated?

Blind Fuzzing

- Start with a test corpus of well formed program inputs or generate new inputs
- Apply random or systematic mutations to program inputs
- Run program with mutated inputs and observe whether or not the program crashes
- Repeat until the program "crashes"
- Input space
  - Reading data in a loops could make the input space infinite
  - There are 2^n possible inputs for a binary input of length *n*



This is about all we can do without examining program artifacts…

What is being searched for?

What is being searched for?

- Crash vs. no crash
- Expected vs. unexpected output
- Tainted vs. untainted output
  - Example: Web app fuzzers provide XSS code as input and monitor for XSS execution
- Harness can translate a domain specific problem to a standard detection output
  - Example: *if(some program state) { crash(); }*

Data Drives Program Execution

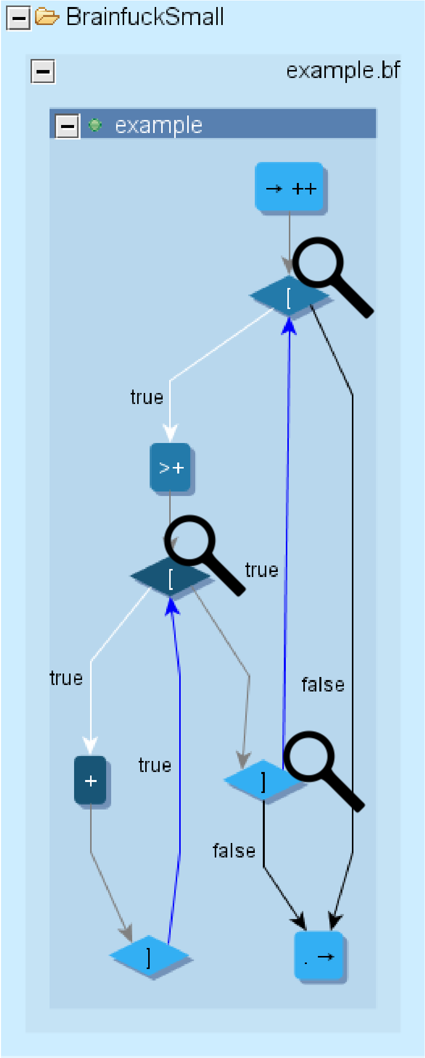
“The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program. When you sort a deck of cards, you’re moving them around, but it’s the numbers on the cards that are telling you where to move them.” - Taylor Hornby, a judge for the Underhanded Crypto Contest

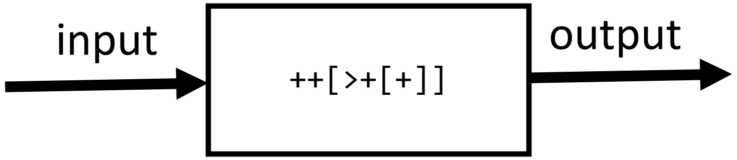
input
Program
output

Graybox Fuzzing

Data Drives Program Execution

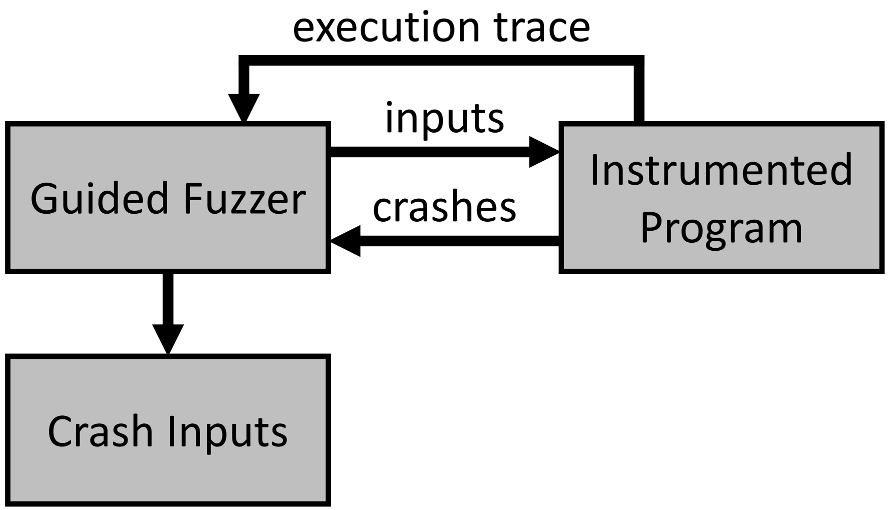
- Use static analysis to look ahead at all program paths
- Monitor which path was taken for a given input
- Correlate information of how changing the input data changes the executed program paths





Guided Fuzzing: Feedback Driven Input Generation

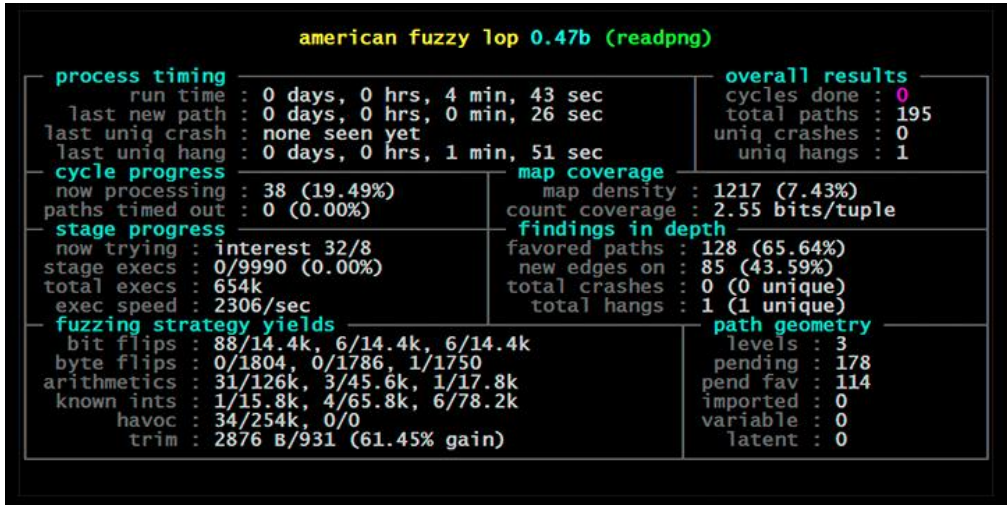
- Start with a test corpus of well formed program inputs
- Apply random or systematic mutations to program inputs
- Instrument the program branch points
- Run the instrumented program with mutated inputs and 1) observe whether or not the program crashes and 2) record the program execution path coverage
- If the input results in new program paths being explored then prioritize mutations of the tested input
- Repeat until the program crashes



Heuristics guide genetic algorithm to generate program inputs that push the fuzzer deeper into the program control flow, avoiding the common pitfalls of fuzzers to only test “shallow” code regions.

AFL (American Fuzzy Lop) Fuzzer

- Recognized as the current industry standard implementation of guided fuzzing
  - Effective mutation strategy to generate new inputs from initial test corpus
  - Lightweight instrumentation at branch points
  - Genetic algorithm promotes mutations of inputs that discover new branch edges
    - Aims to explore all code paths
  - Huge trophy case of bugs found in wild
    - 371+ reported bugs in 161 different programs as of March 2018
    - Tool: http://lcamtuf.coredump.cx/afl/
- A game of economics. AFL tends to “guess” the correct input faster than a smart tool “computes” the correct input

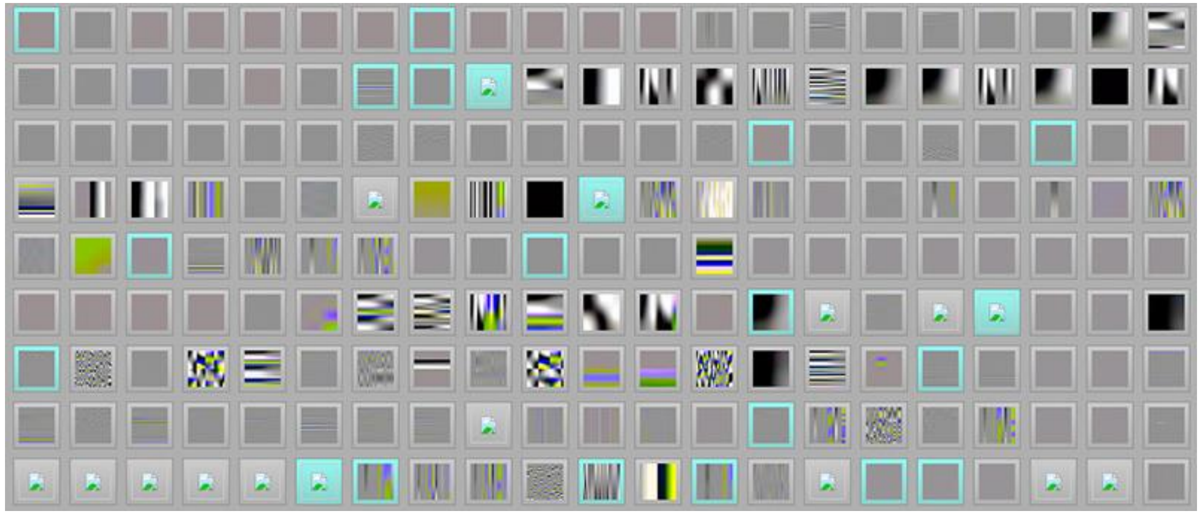


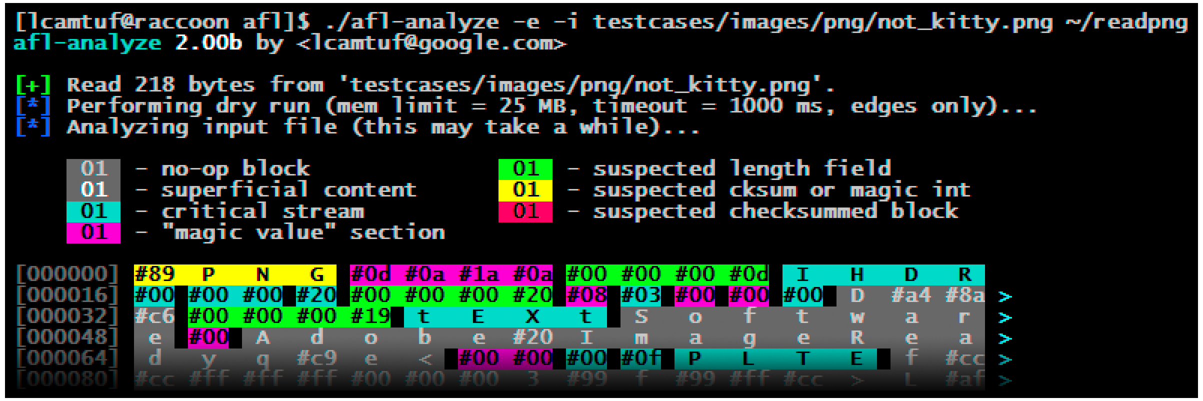
```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec       │  cycles done : 0  │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec       │  total paths : 195│
│ last uniq crash : none seen yet                      │ uniq crashes : 0  │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec       │   uniq hangs : 1  │
├─ cycle progress ─────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 38 (19.49%)    │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)      │ count coverage : 2.55 bits/tuple       │
├─ stage progress ─────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8      │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)     │  new edges on : 85 (43.59%)            │
│ total execs : 654k               │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec           │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k         │    levels : 3          │
│  byte flips : 0/1804, 0/1786, 1/1750             │   pending : 178        │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k          │  pend fav : 114        │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k          │  imported : 0          │
│       havoc : 34/254k, 0/0                       │  variable : 0          │
│        trim : 2876 B/931 (61.45% gain)           │    latent : 0          │
└──────────────────────────────────────────────────┴────────────────────────┘
```

Path Coverage is a Surprisingly Effecting Heuristic

- The heuristic to generate inputs that drives the execution to new paths can be effective
- Impressive given that JPEGs are non-trivial parsers that include Huffman compression and have multiple magic header sequences (e.g. 0xFFD8 and 0xFFD9)
- The inputs that survive share some common properties

Randomly generated JPEGs that actually parse by libjpeg





```
[lcamtuf@raccoon afl]$ ./afl-analyze -e -i testcases/images/png/not_kitty.png ~/readpng
afl-analyze 2.00b by <lcamtuf@google.com>

[+] Read 218 bytes from 'testcases/images/png/not_kitty.png'.
[*] Performing dry run (mem limit = 25 MB, timeout = 1000 ms, edges only)...
[*] Analyzing input file (this may take a while)...

    01  - no-op block                  01  - suspected length field
    01  - superficial content          01  - suspected cksum or magic int
    01  - critical stream              01  - suspected checksummed block
    01  - "magic value" section

[000000] #89  P   N   G  #0d #0a #1a #0a #00 #00 #00 #0d  I   H   D   R
[000016] #00 #00 #00 #20 #00 #00 #00 #20 #08 #03 #00 #00 #00  D  #a4 #8a >
[000032] #c6 #00 #00 #00 #19  t   E   X   t   S   o   f   t   w   a   r  >
[000048]  e  #00  A   d   o   b   e  #20  I   m   a   g   e   R   e   a  >
[000064]  d   y   q  #c9  e   <  #00 #00 #00 #0f  P   L   T   E   f  #cc >
[000080] #cc #ff #ff #ff #00 #00 #00  3  #99  f  #99 #ff #cc  >   L  #af >
```

Sources: https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html
https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html

Fuzzer Harness

- AFL assumes all inputs are binary files accepted by a program
- A main function (program entry point) is required to execute a program
  - A library typically does not have a main method, so one must be provided to make a complete executable program
- Practically needed to translate fuzzer generated inputs to expected program input format
  - Example: AFL generates a file as input. To fuzz a DNS service the harness must translate the generated input file to a DNS packet or data structure a function takes as a parameter

What is the state-of-the-art dynamic analysis?

Whitebox Fuzzing

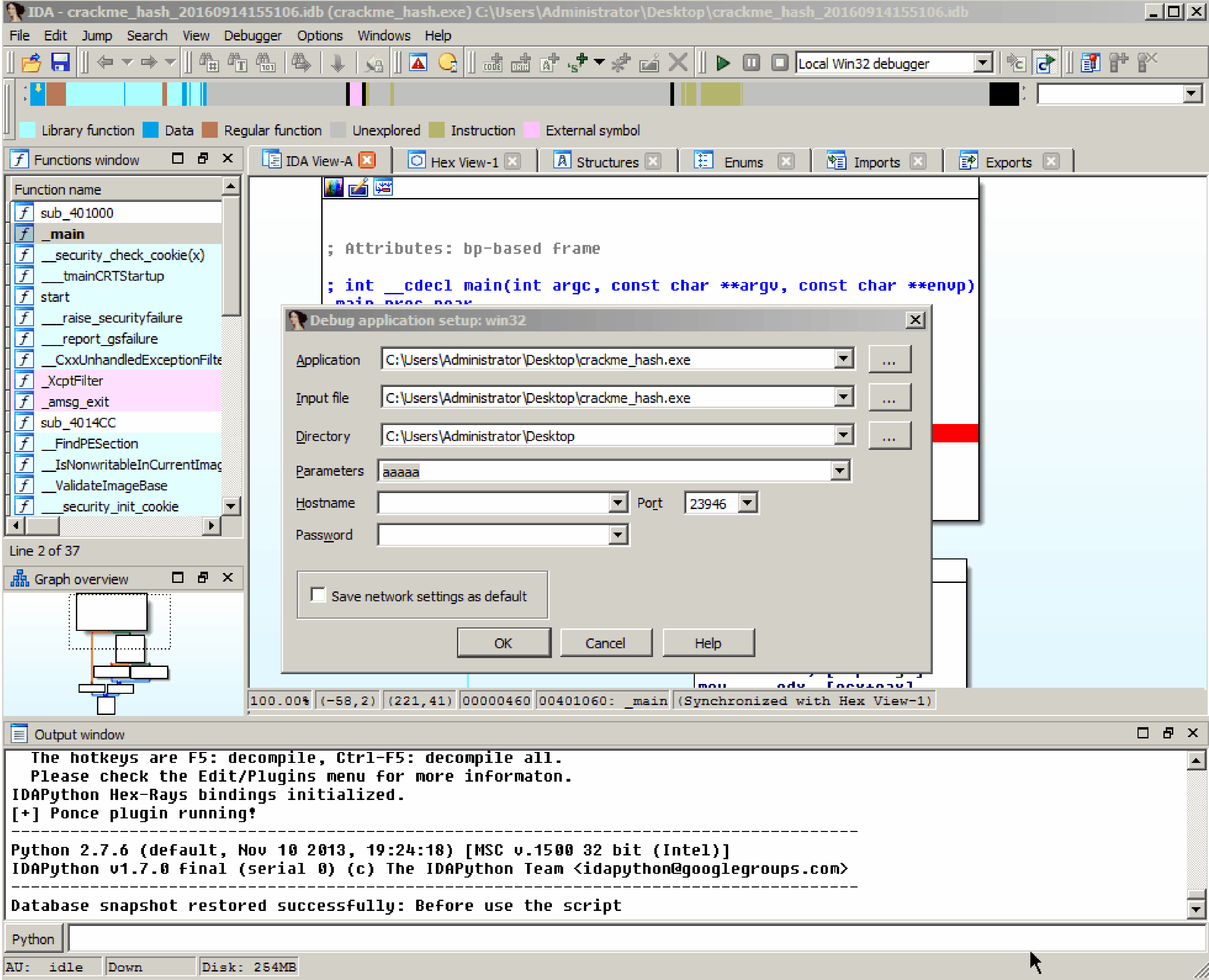
Symbolic Execution

- Replace concrete assignment values with symbolic values
- Perform operations on symbolic values abstractly
- At each branch fork the abstracted logic
  - Dealing with path explosion problem is a challenge!
- Utilize SAT/SMT solvers to determine if the constraints are satisfiable for a path of interest
  - Example: fail occurs if `y * 2 = z = 12` is satisfiable
    - Solve`(y * 2 = 12, y), y = 6` satisfies the constraint
    - Failure occurs when read() returns 6
  - Reasoning about true path of "`if(a * b == c)`" could force analysis to solve prime factorization if `c` is the product of two large primes

```
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    fail();
  } else {
    printf("OK");
  }
}
```

On what inputs does the code fail?

```c
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
        int i;
        int hash = 0xABCD;

        for (i = 0; ptr[i]; i++)
                hash += ptr[i] ^ serial[i % 5];

        return hash;
}

int main(int ac, char **av)
{
        int ret;

        if (ac != 2)
                return -1;

        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");

        return 0;
}
```



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
```

Debug application setup: win32

Application	C:\Users\Administrator\Desktop\crackme_hash.exe
Input file	C:\Users\Administrator\Desktop\crackme_hash.exe
Directory	C:\Users\Administrator\Desktop
Parameters	aaaaa
Hostname	
Port	23946
Password	

□ Save network settings as default

OK Cancel Help

```
100.00% (-58,2) (221,41) 00000460 00401060: _main (Synchronized with Hex View-1)
```

```
  The hotkeys are F5: decompile, Ctrl-F5: decompile all.
  Please check the Edit/Plugins menu for more informaton.
IDAPython Hex-Rays bindings initialized.
[+] Ponce plugin running!
---------------------------------------------------------------------------------------
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
IDAPython v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
---------------------------------------------------------------------------------------
Database snapshot restored successfully: Before use the script
```

```
AU:  idle   Down   Disk: 254MB
```

https://github.com/illera88/Ponce

Concolic Execution

- A hybrid of dynamic analysis and symbolic execution
  - Perform symbolic execution on some variables and concrete execution on other variables
  - Symbolic variables could be made concrete in order to:
    - Move past symbolic limitations such as challenges in modeling the program environment (example network interaction)
    - Deal with path explosion problem and satisfiability problem by replacing difficult symbolic values with concrete values to simplify analysis
  - Pays cost in time for symbolic computations and execution time of program
- Several well known tools:
  - Angr - http://angr.io
  - KLEE - https://klee.github.io
  - DART - https://dl.acm.org/citation.cfm?id=1065036
  - CREST (formerly CUTE) - https://code.google.com/archive/p/crest
  - Microsoft SAGE - https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf

DARPA’s Cyber Grand Challenge (CGC)

- “Cyber Grand Challenge (CGC) is a contest to build high-performance computers capable of playing in a Capture-the-Flag style cyber-security competition.”
- DEFCON 2016

https://www.darpa.mil/program/cyber-grand-challenge

DARPA’s Cyber Grand Challenge (CGC)

- Fully automatic reasoning to:
  - Detect program vulnerabilities
  - Patch programs to prevent exploitation
  - Develop and execute vulnerability exploits against competitors
- No human players!

CGC

CYBER

GRAND_CHALLENGE

CGC Results (Reading Between the Lines)

- All teams published the same essential combination of strategies
  - Guided fuzzing (nearly every team had modified AFL)
  - Symbolic/concolic execution to assist fuzzer sometimes aided by classical program analyses (points-to, reachability, slicing, etc.)
  - Some state space pruning and prioritization scheme catered to expected vulnerability types
- Effective patches were more often generic patches which addressed the class of vulnerabilities not the one-off vulnerability that was given
  - Example: Adding stack guards for memory protection
- Competitor scores were close!
  - The difference between 1st and 7th place was not substantial
- Classes of vulnerabilities were known *a priori*

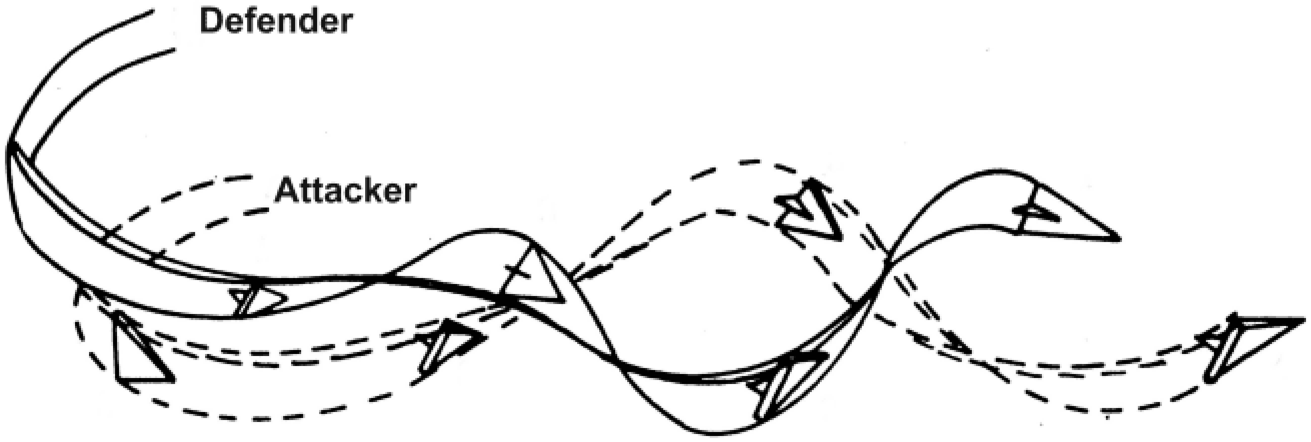
How can we do better?

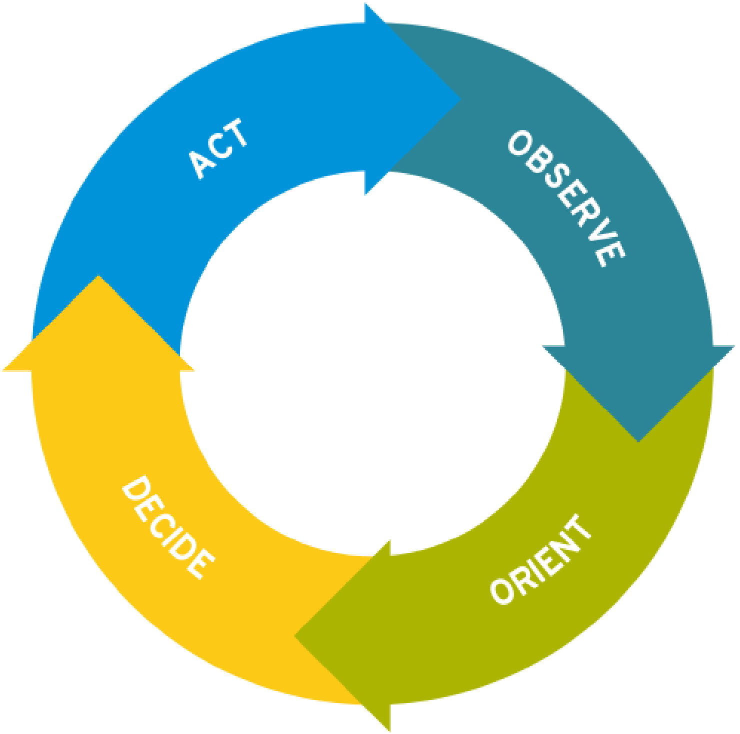
A Simple Observation…

- *Humans armed with even simple tools are still finding bugs that huge racks of super computers can't find…*
  - Case Study: CVE-2002-1337
- Remember that the programs we care about are created by humans
  - Humans naturally imbue code with additional structure (e.g. design patterns)
- Leverage strengths of human + machine
  - Let humans amplify machine reasoning
  - Let machines amplify human reasoning
  - Premise of DARPA CHESS program
- Case Study: Linux lock/unlock pairing
  - Teaching a machine the developer's pattern of using unique types for instances avoids expensive pointer analysis

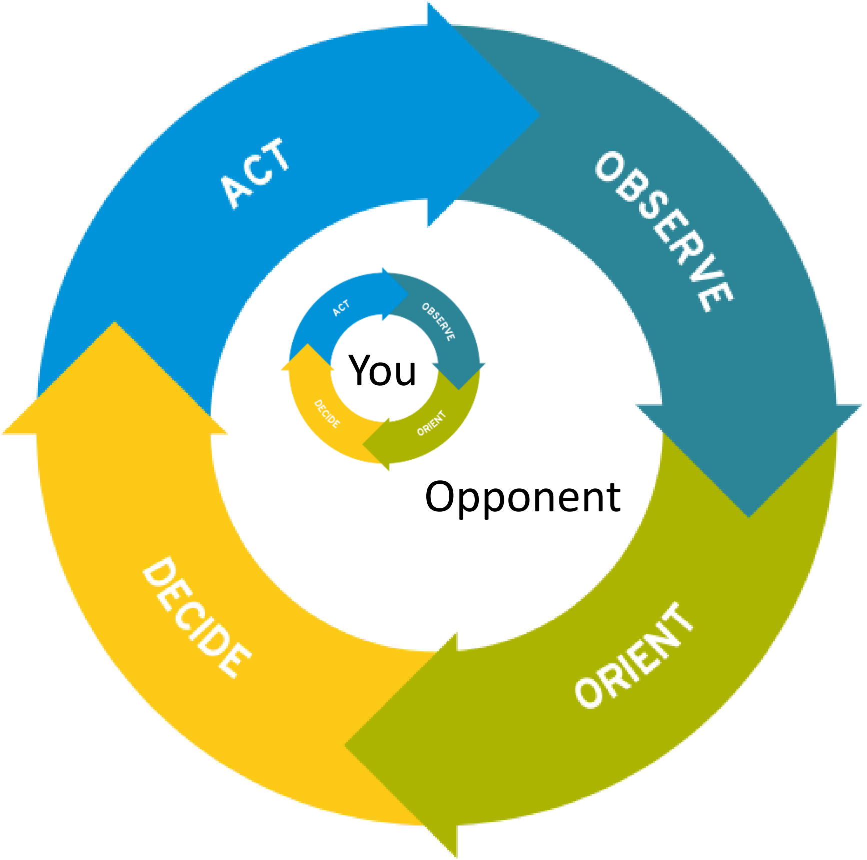
Program Analysis, OODA, and YOU

- “Security is a *process*, not a product” – Bruce Schneier
- Apply John Boyd’s OODA loop to software and security





Program Analysis, OODA, and YOU

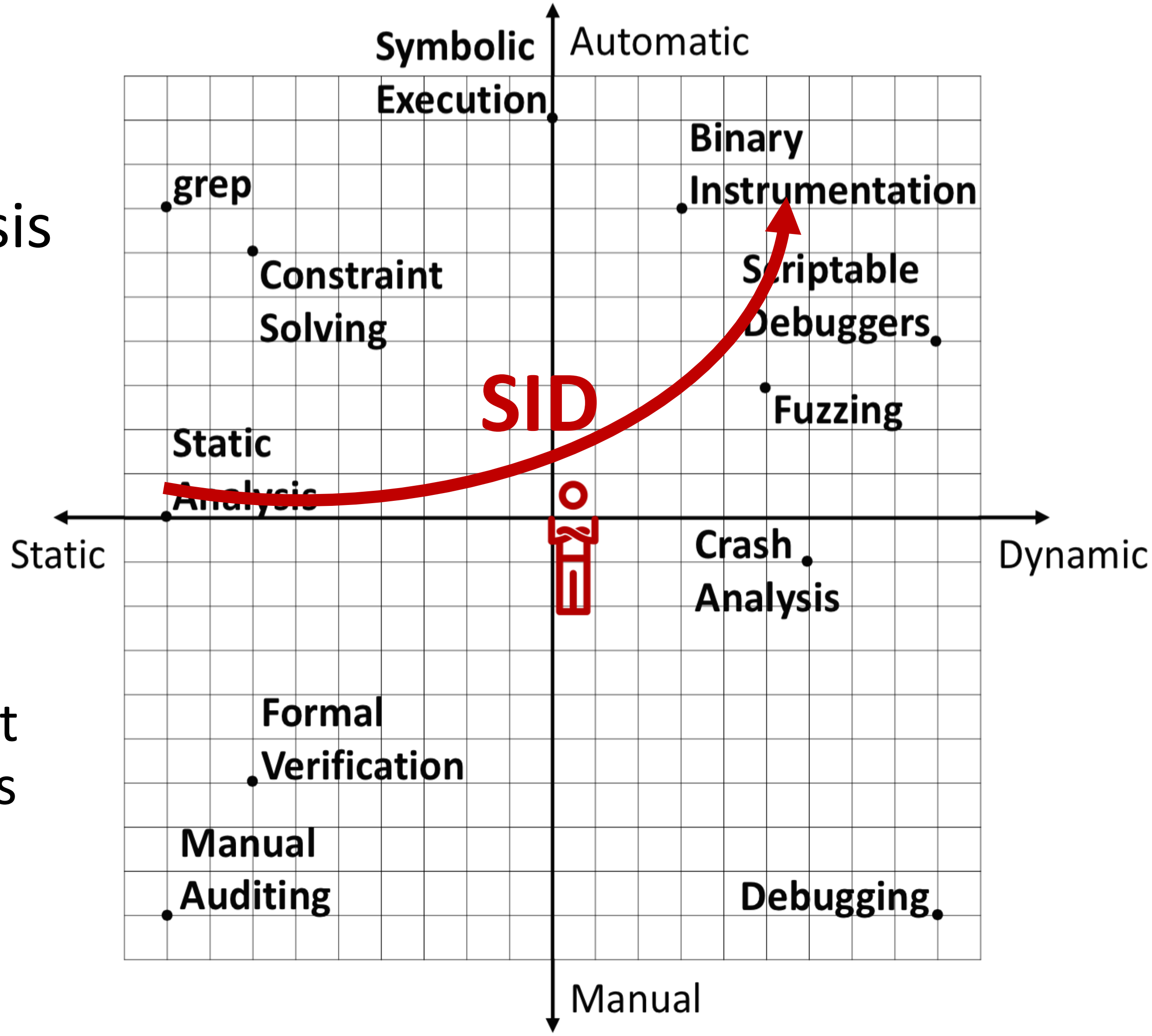


Our opponent is <u>time</u>!

"...IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself."
– Fred Brooks

A New Approach

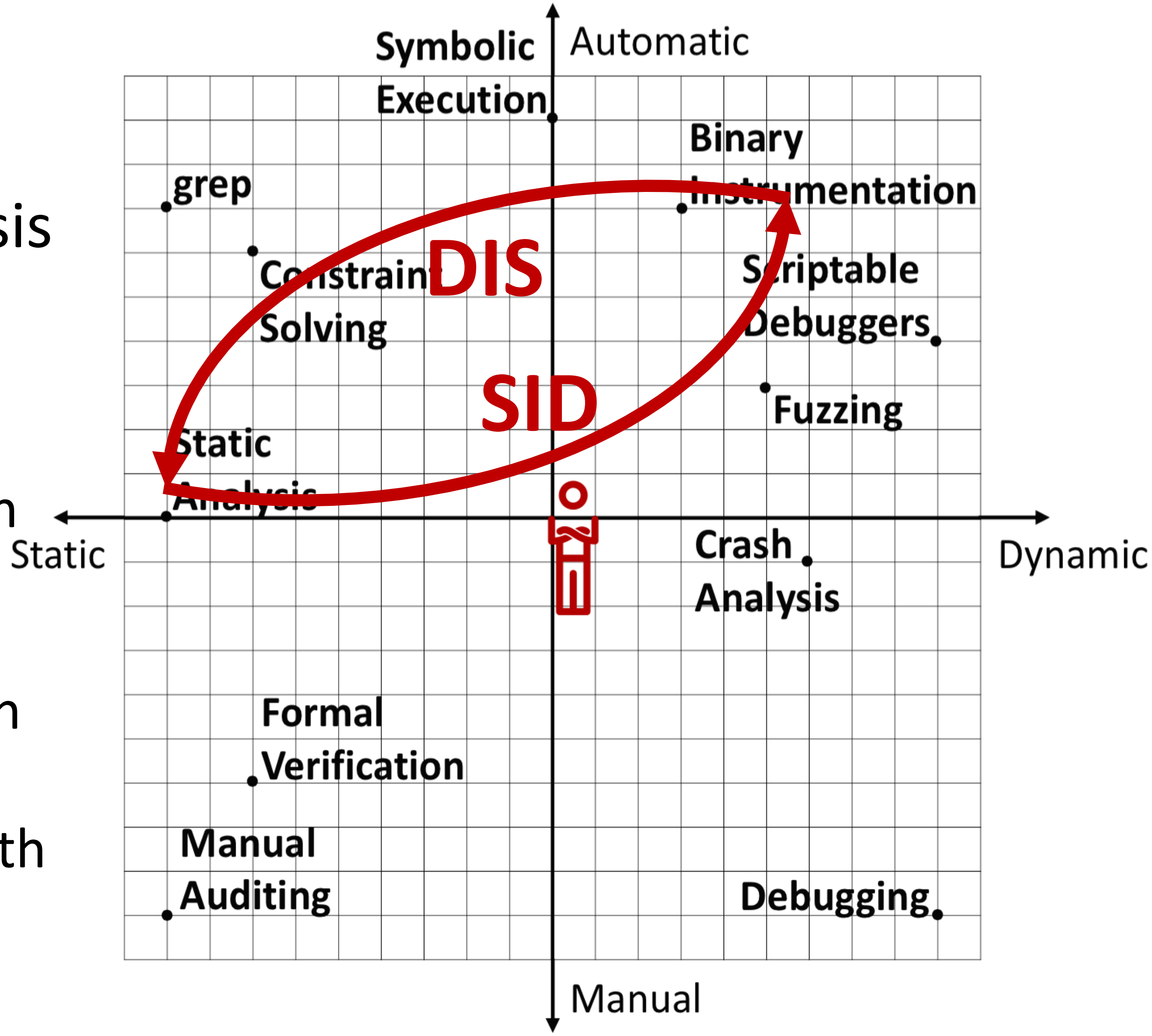
- Statically-informed Dynamic Analysis
  - Human selects events of interest
  - Static computation of relevant behaviors
  - Automatic program modifications prevent execution of irrelevant behaviors
  - Automatic generation of skeleton test harness for targeted dynamic analysis



Spectrum Grid by Julian Cohen [Blackhat 2014]

A New Approach

- Dynamically-informed Static Analysis
  - Human completes test harness by adapting fuzzer inputs to program inputs
  - Guided fuzzer drives input generation on modified program
  - Dynamic invariant detection is performed only on relevant execution traces
  - Static program graph is annotated with behavior-relevant invariants



Spectrum Grid by Julian Cohen [Blackhat 2014]

Revisit: What is monitored in the program?

Program Invariants

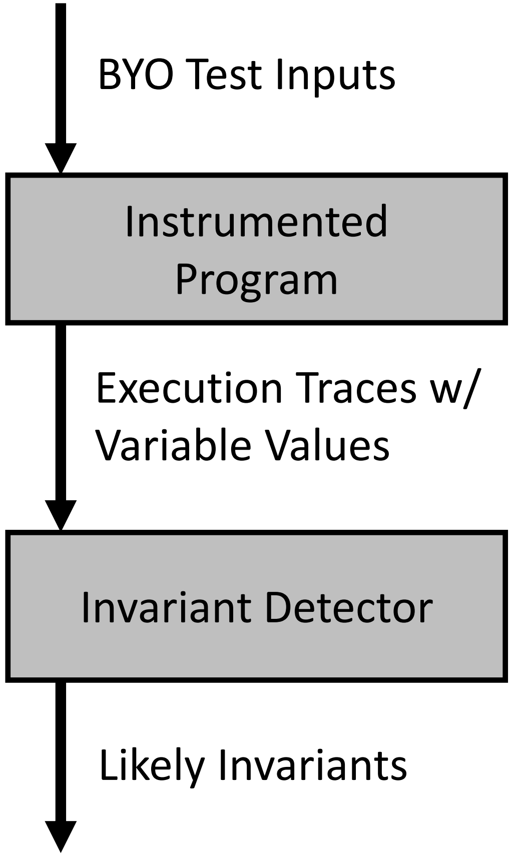
“Programmers have invariants in mind … when they write or otherwise manipulate programs: they have an idea of how the system works or is intended to work, how the data structures are laid out and related to one another, and the like. Regrettably, these notions are rarely written down...” ~ Michael Ernst

Program Invariant:

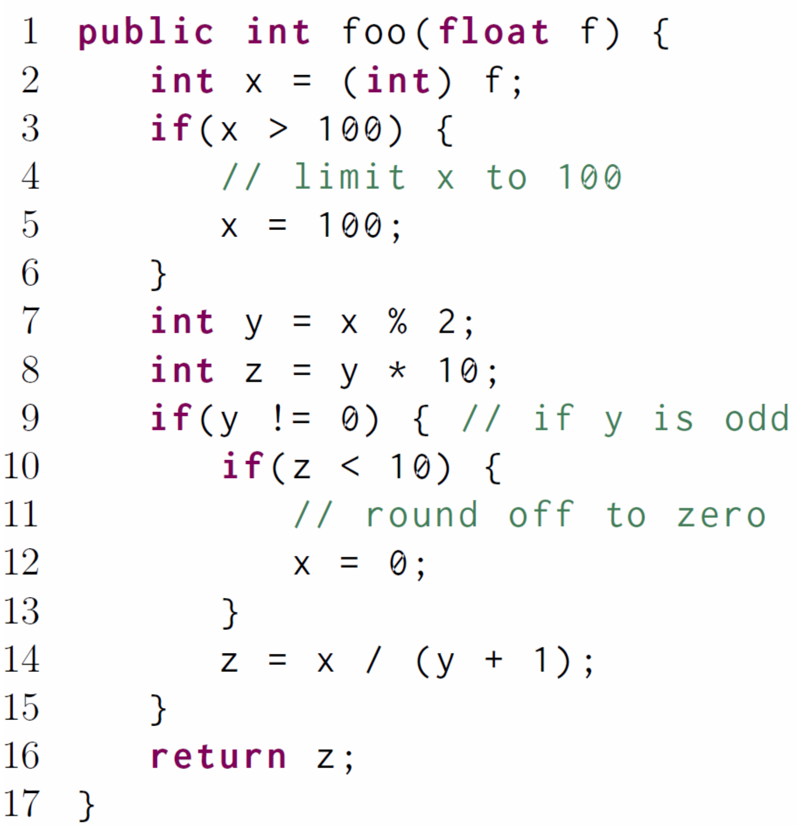
- “a property that is true at a particular program point or points” [Ernst 2000]
- “a property of a program that is always true for every possible runtime state of the program” [MIT OpenCourseWare 6.005]

Dynamic Invariant Detection

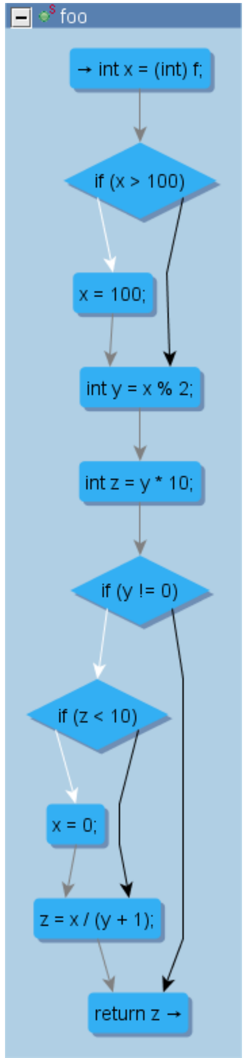
- Daikon: Dynamic Likely Invariant Detection
  - Dynamic analysis only observes feasible paths
  - Program variables are instrumented on all program paths
  - BYO test input strategy, typically used with unit tests or randomized testing
  - Large collection of program invariant patterns (ex: types)
  - Correctness is w.r.t. what was observed. Example: “x > 0” may only be true if negative values were never tested.
  - Can be expensive. Instrumentation adds overhead to execution time and invariant detection must employ many logic tricks in order to scale.
  - Tool: https://plse.cs.washington.edu/daikon/



Recap: Control Flow Graph



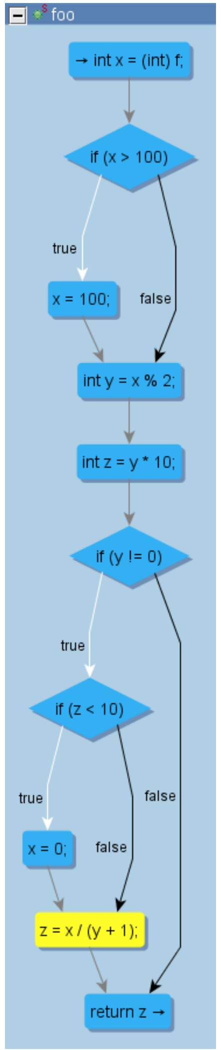
```java
public int foo(float f) {
    int x = (int) f;
    if(x > 100) {
        // limit x to 100
        x = 100;
    }
    int y = x % 2;
    int z = y * 10;
    if(y != 0) { // if y is odd
        if(z < 10) {
            // round off to zero
            x = 0;
        }
        z = x / (y + 1);
    }
    return z;
}
```



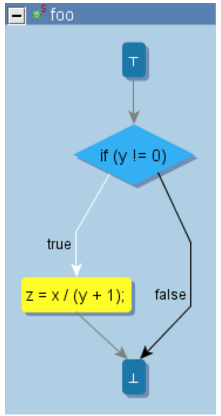
6 program paths

Projected Control Graph

- In 2016, Tamrawi proposed a PCG abstraction
  - Defined a graph homomorphism to efficiently group program behaviors into equivalence classes
  - Parameterized by control flow events of interest
  - Only relevant event statements and necessary conditions are retained



CFG to PCG Transformation



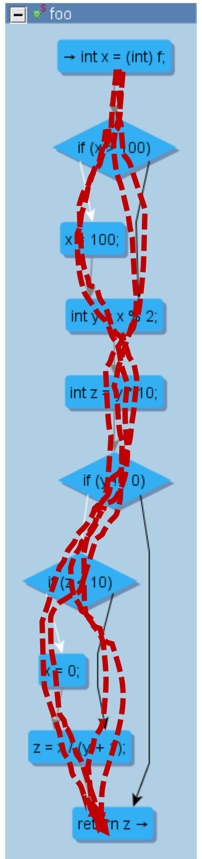
Homomorphic Program Invariants

What are the program invariants that hold true with respect to an equivalence class of control flow paths?

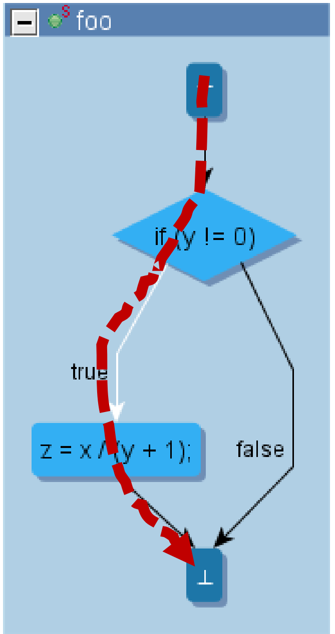
What are the values of y when the true path of the y != 0 branch is taken?

Recall: *y = x % 2;*
y ∈ {-1, 0, 1} for all paths
y ∈ {-1, 1} for the 4 relevant paths

… if y is -1 then division by zero will occur!



4 paths in the control flow graph are equivalent with respect to whether or not a division operation occurs



Revisit: What is executed in the program?

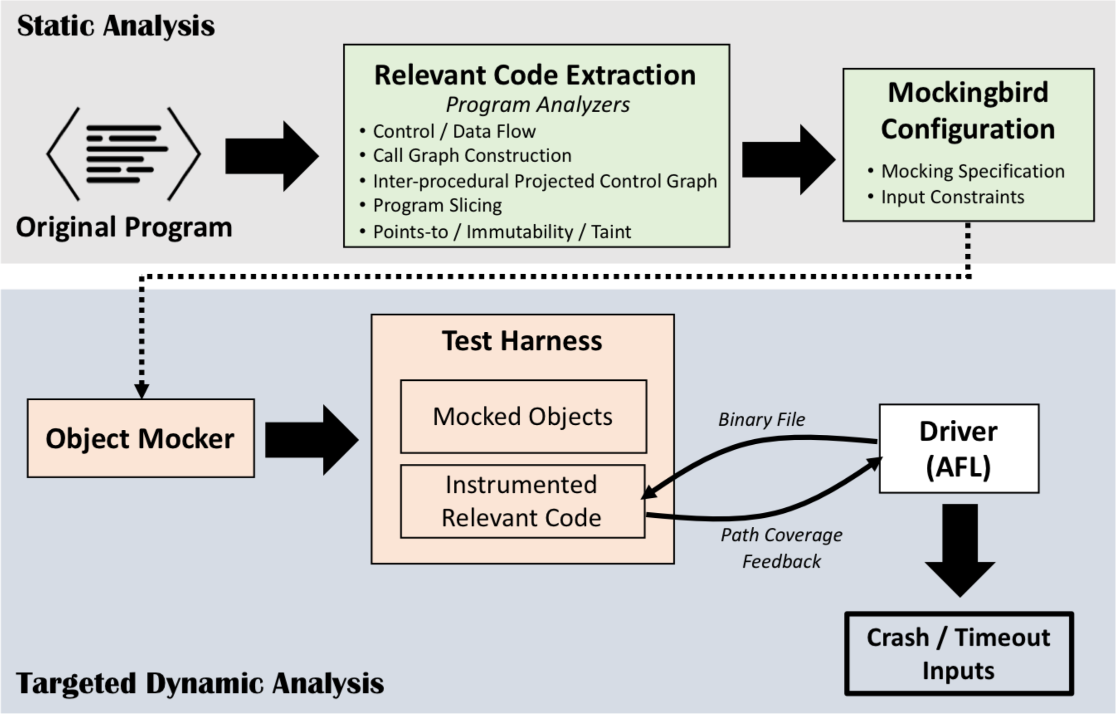
Targeted Fuzzing

- Do we have to fuzz from the start of the program? No!
- Most techniques necessitate manually developing a harness, which is a natural opportunity to “target” fuzzing on a subset of the program
- Some functions are more natural to fuzz then others (ex: library APIs)
  - Helper functions may depend on state of global variables or complex data structures as parameters
- To generically fuzz a function or set of functions the dependencies must be mocked
  - Fuzzing internal program states (mocked dependencies) may ignore a practical constraint on program state enforced at runtime
  - Human reasoning could be used to add fuzzer input generation constraints

Targeted Dynamic Analysis

- Decouples target code from control and data dependencies by replacing objects with mocked objects
  - Global variables
  - Method parameters passed
  - Method return values

- Mocked objects have no dependencies
- Mocked object values can be programmatically stimulated



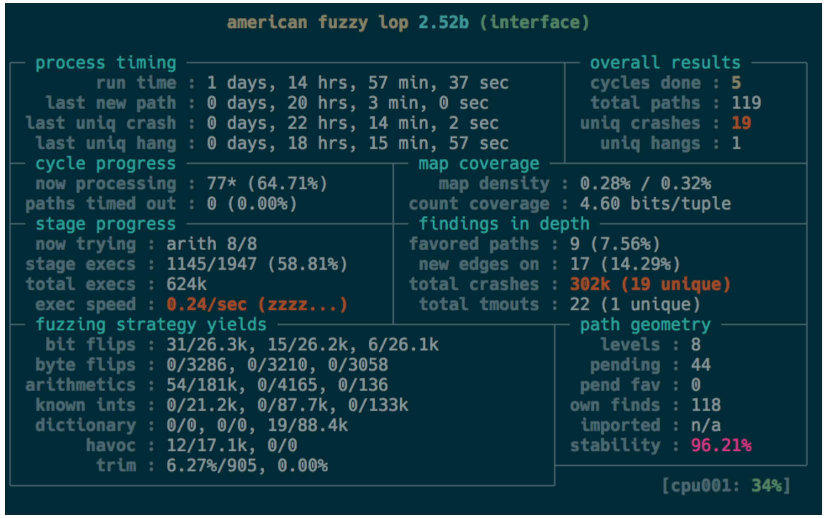
Mockingbird: A Framework for Enabling Targeted Dynamic Analysis of Java Programs.

Example Targeted Dynamic Analysis

```java
public class Example {
  public static boolean isVowel(char c) {
    return c == 'a' || c == 'e' || c == 'i'
        || c == 'o' || c == 'u' || c == 'y';
  }
  class Pet {
    private String name;
    public Pet(String name) {
      this.name = name;
      sleep(5000);
    }
    public String getName() {
      return name;
    }
    public double getVowelRatio() {
      double vowels = 0;
      String name = getName().toLowerCase();
      for(char c : name.toCharArray()) {
        if(isVowel(c)) {
          vowels++;
        }
      }
      return vowels / (name.length() - vowels);
    }
  }
}
```

Mock Specification

```json
{
  "definition": {
    "class": "Example$Pet",
    "method": "getVowelRatio",
    "instance_variables": [
        {
            "name": "name"
        }
    ]
  },
  "config": {
    "timeout": 1000
  }
}
```

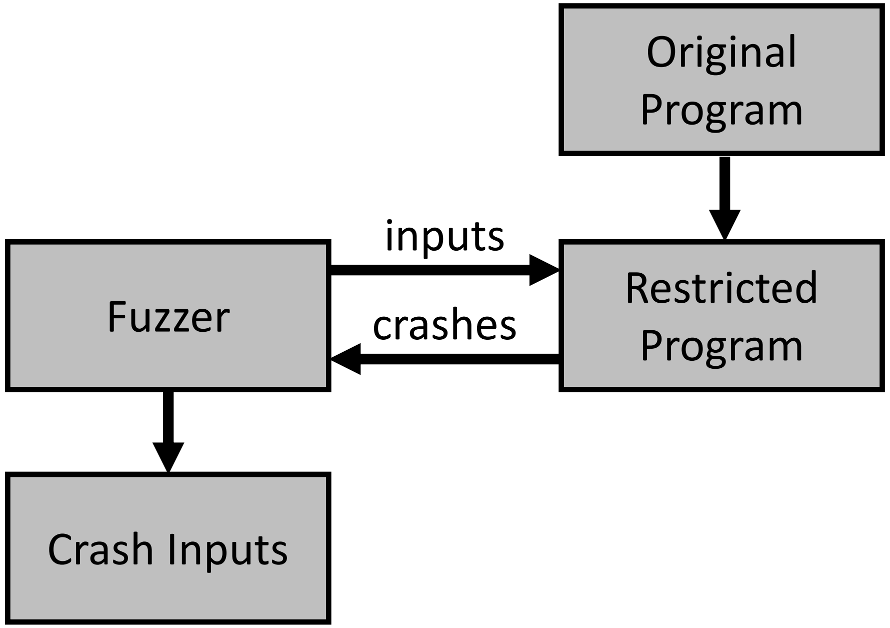


```
                    american fuzzy lop 2.52b (interface)

┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 1 days, 14 hrs, 57 min, 37 sec     │  cycles done : 5      │
│   last new path : 0 days, 20 hrs, 3 min, 0 sec       │  total paths : 119    │
│ last uniq crash : 0 days, 22 hrs, 14 min, 2 sec      │ uniq crashes : 19     │
│  last uniq hang : 0 days, 18 hrs, 15 min, 57 sec     │   uniq hangs : 1      │
├─ cycle progress ────────────────┬─ map coverage ─────┴───────────────────────┤
│  now processing : 77* (64.71%)  │    map density : 0.28% / 0.32%             │
│ paths timed out : 0 (0.00%)     │ count coverage : 4.60 bits/tuple           │
├─ stage progress ────────────────┼─ findings in depth ────────────────────────┤
│  now trying : arith 8/8         │ favored paths : 9 (7.56%)                  │
│ stage execs : 1145/1947 (58.81%)│  new edges on : 17 (14.29%)                │
│ total execs : 624k              │ total crashes : 302k (19 unique)           │
│  exec speed : 0.24/sec (zzzz...)│  total tmouts : 22 (1 unique)              │
├─ fuzzing strategy yields ───────┴─────────────────┬─ path geometry ────────┤
│   bit flips : 31/26.3k, 15/26.2k, 6/26.1k          │    levels : 8          │
│  byte flips : 0/3286, 0/3210, 0/3058               │   pending : 44         │
│ arithmetics : 54/181k, 0/4165, 0/136               │  pend fav : 0          │
│  known ints : 0/21.2k, 0/87.7k, 0/133k             │ own finds : 118        │
│  dictionary : 0/0, 0/0, 19/88.4k                   │  imported : n/a        │
│       havoc : 12/17.1k, 0/0                        │ stability : 96.21%     │
│        trim : 6.27%/905, 0.00%                     ├────────────────────────┘
└────────────────────────────────────────────────────┘          [cpu001: 34%]
```

Revisit (Again): What is executed in the program?

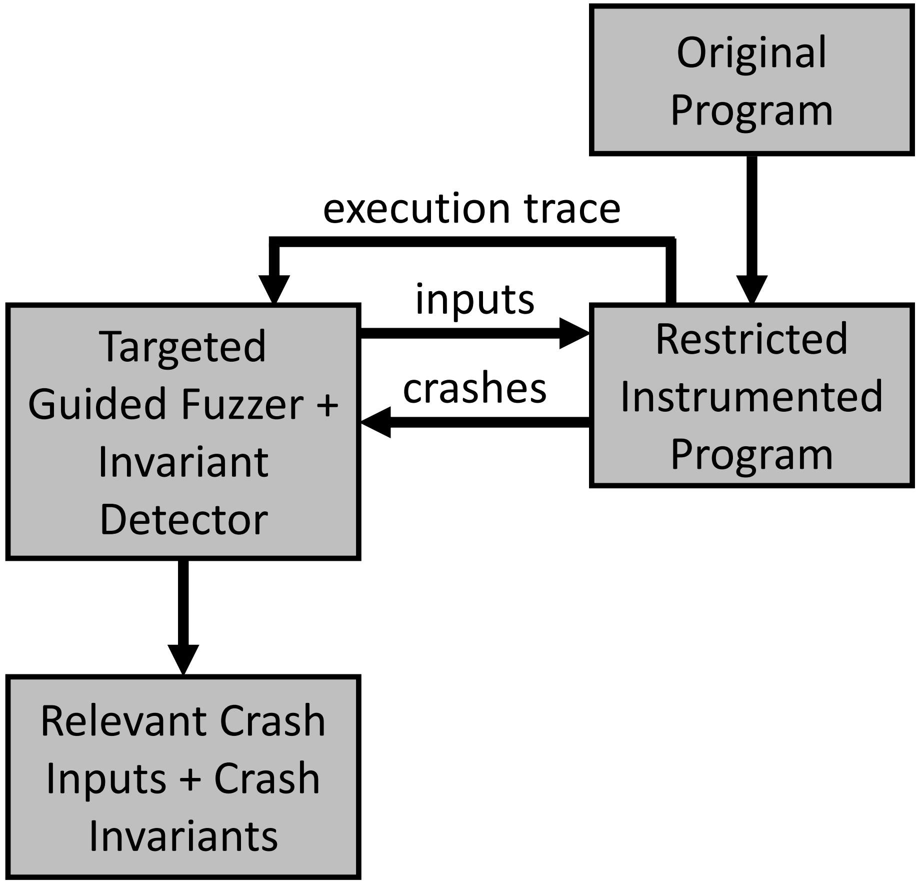
Restricted Program Fuzzing

- A fuzzer could be made smarter by changing the program being fuzzed
- Statically compute a program restricted to code relevant to a crash
  - Assumes some knowledge of relevant crash events can be specified *a priori*
  - Example: Compute program slice and retain only crash relevant program statements
  - Example: Compute PCG of crash events and abort on paths that do not lead to crash events
- Faster execution
- Subset of all program behaviors



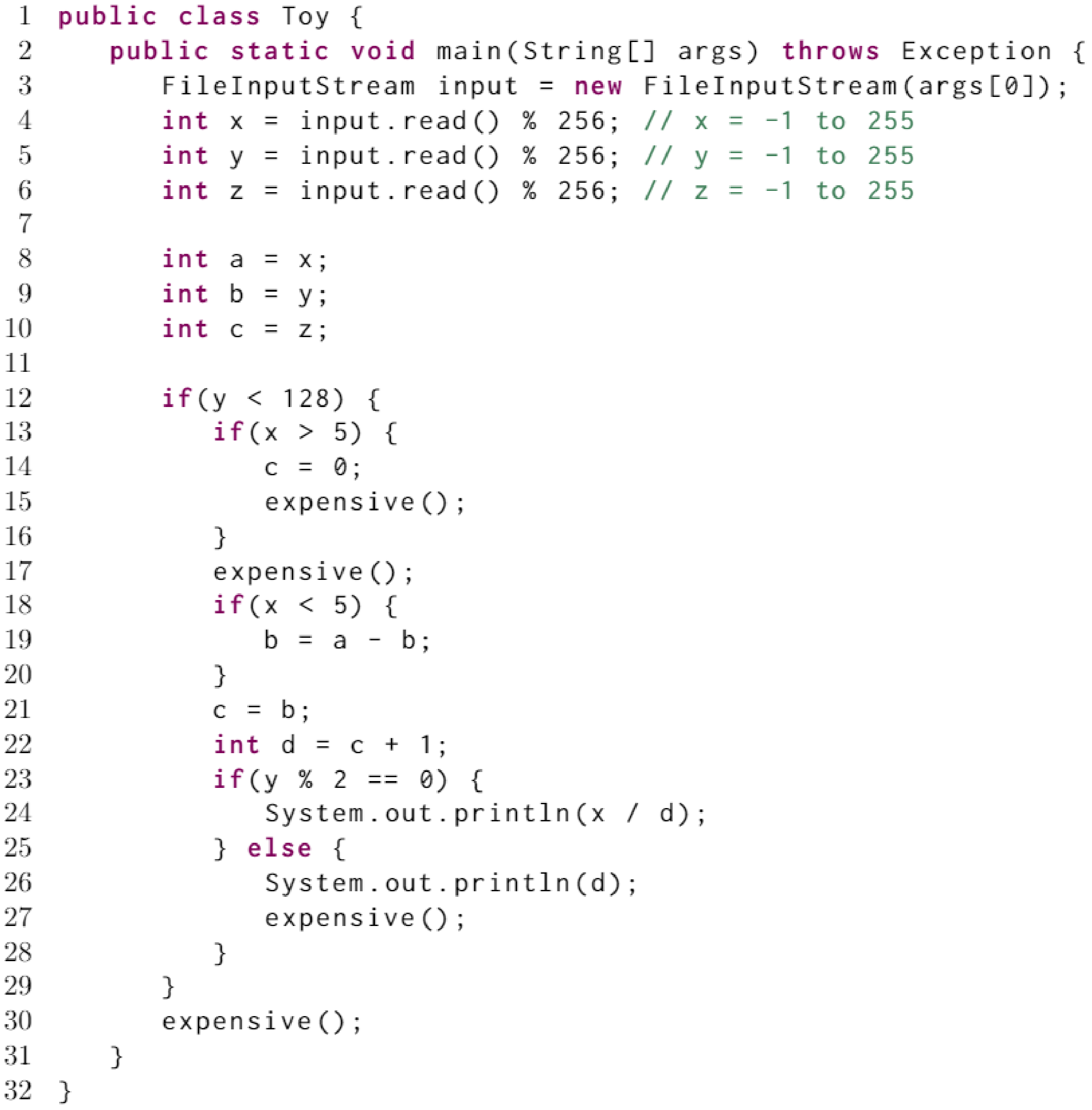
New Approach Workflow

- *Human*: Identify potentially interesting program behaviors
- *Machine:* Generate a program that *restricts* execution to the identified behaviors and *instruments* program points on interesting paths
- *Human*: *Targets* dynamic analysis at a particular program entry point
- *Machine*: Fuzz and produce invariants
  - Generate test inputs for given entry point and guide test generation based on execution feedback
  - Output *invariants* of interesting program behaviors
- *Human: Repeat* process to improve human comprehension of program


Original Program
execution trace
inputs
crashes
Targeted Guided Fuzzer + Invariant Detector
Restricted Instrumented Program
Relevant Crash Inputs + Crash Invariants

Motivating Example

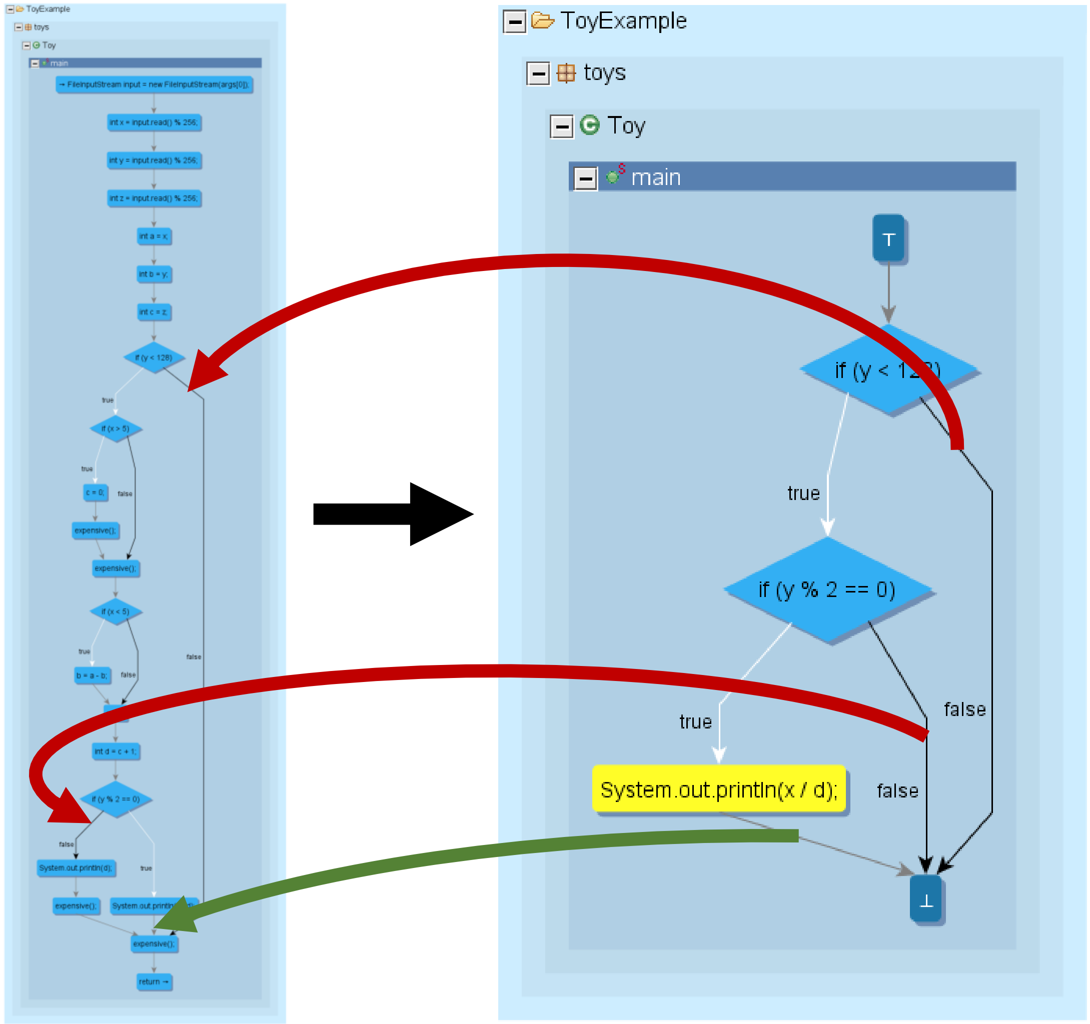
- Is this program bug free?
- Could a division by zero error occur on line 24?
- What conditions are relevant to verifying the program?
- If the program is buggy, what inputs are required to produce the error?
- What input constraints must be satisfied to produce the error?
- Is there a family of buggy inputs?



```
 1 public class Toy {
 2     public static void main(String[] args) throws Exception {
 3         FileInputStream input = new FileInputStream(args[0]);
 4         int x = input.read() % 256; // x = -1 to 255
 5         int y = input.read() % 256; // y = -1 to 255
 6         int z = input.read() % 256; // z = -1 to 255
 7
 8         int a = x;
 9         int b = y;
10         int c = z;
11
12         if(y < 128) {
13             if(x > 5) {
14                 c = 0;
15                 expensive();
16             }
17             expensive();
18             if(x < 5) {
19                 b = a - b;
20             }
21             c = b;
22             int d = c + 1;
23             if(y % 2 == 0) {
24                 System.out.println(x / d);
25             } else {
26                 System.out.println(d);
27                 expensive();
28             }
29         }
30         expensive();
31     }
32 }
```

Program Modifications (1)

- Technique 1 - Aborting Irrelevant Path Execution
  - Only modification needed to compute behavior-relevant invariants
  - Inject an abort signal at the start of an irrelevant path
  - Insert an *abort-irrelevant* signal before any statement in the CFG that is a successor of a branch reachable from a reverse step in the PCG from the ⊥, omitting events
  - Optionally, insert an *abort-relevant* signal after events reachable in a reverse step of the PCG from the ⊥

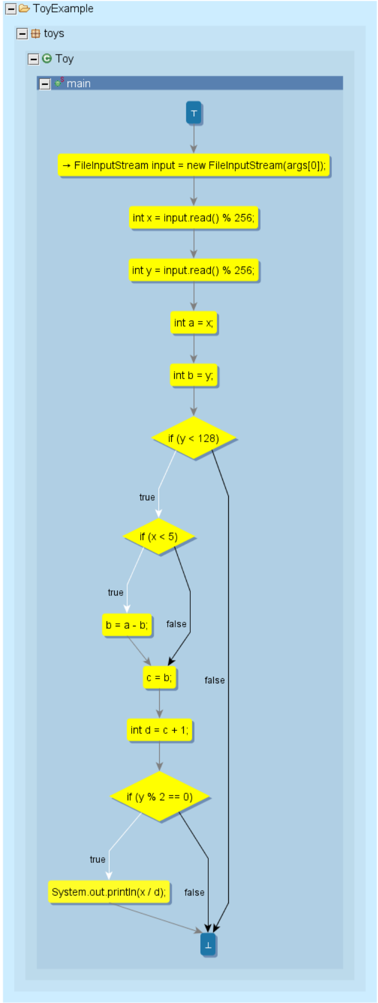


CFG

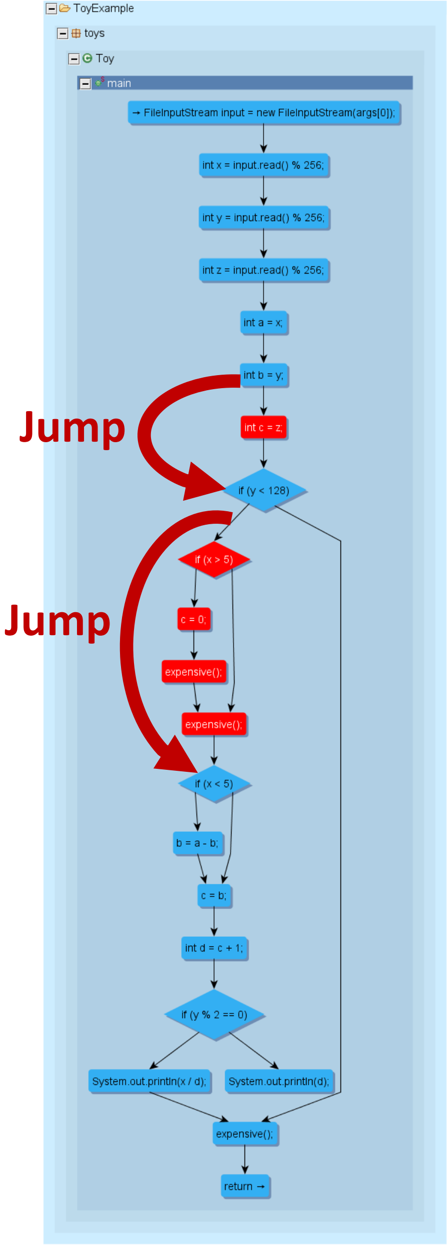
PCG(E), E=Division Statement

Program Modifications (2)

- Technique 2 - Eliding Irrelevant Statements
  - ***Not strictly necessary***
  - Improves fuzzing speed
  - Program slice computes relevant control and data flow events
  - Elide irrelevant statements by injecting a pair of goto and label statements.
  - Specifically, for any edge in the PCG that is not in the CFG add a label before the successor node and a goto label statement after the predecessor node.



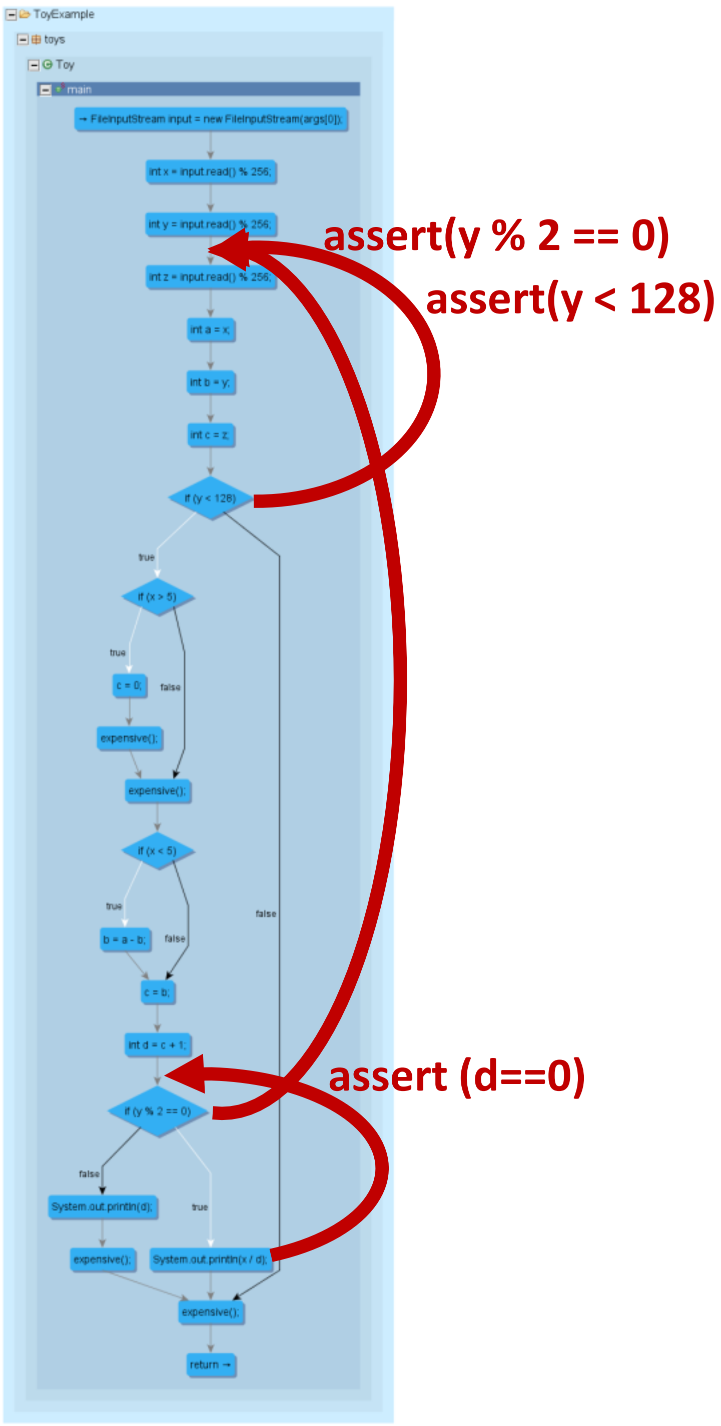
PCG of Relevant Statements



CFG with Irrelevant Statements Marked

Program Modifications (3)

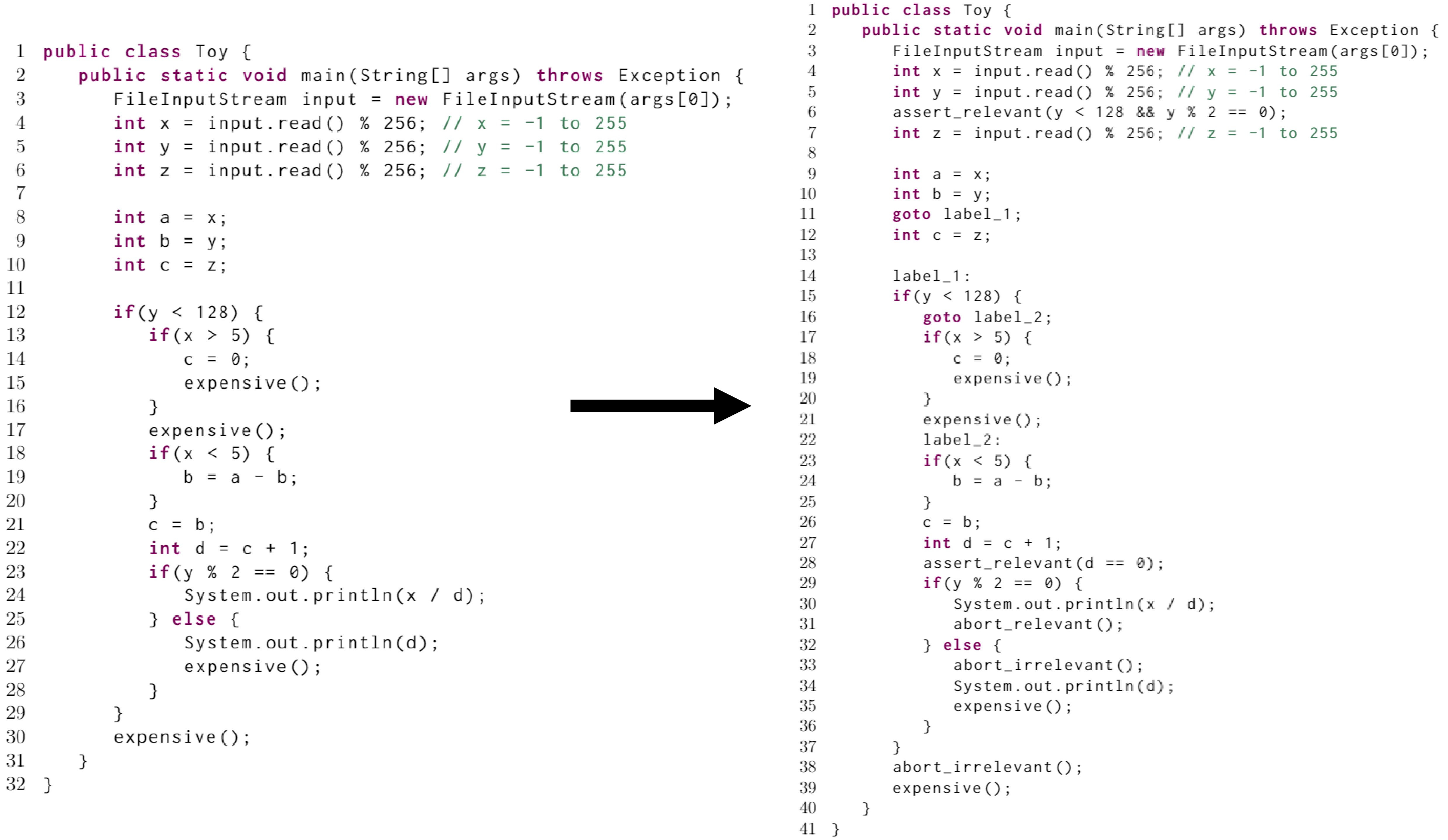
- Technique 3 – Injecting Fail Early Assertions
  - *Not strictly necessary*
    - Can be used to further restrict relevance to a value at a statement. Example assert(d!=0) before the statement print(x / d).
  - Can also be used to improve fuzzing speed by preventing execution of relevant statements.
  - Specifically, for each condition in the PCG, insert an *assert-relevant(condition)* statement at the location of the last reaching definition of the condition variables.



assert(y % 2 == 0)

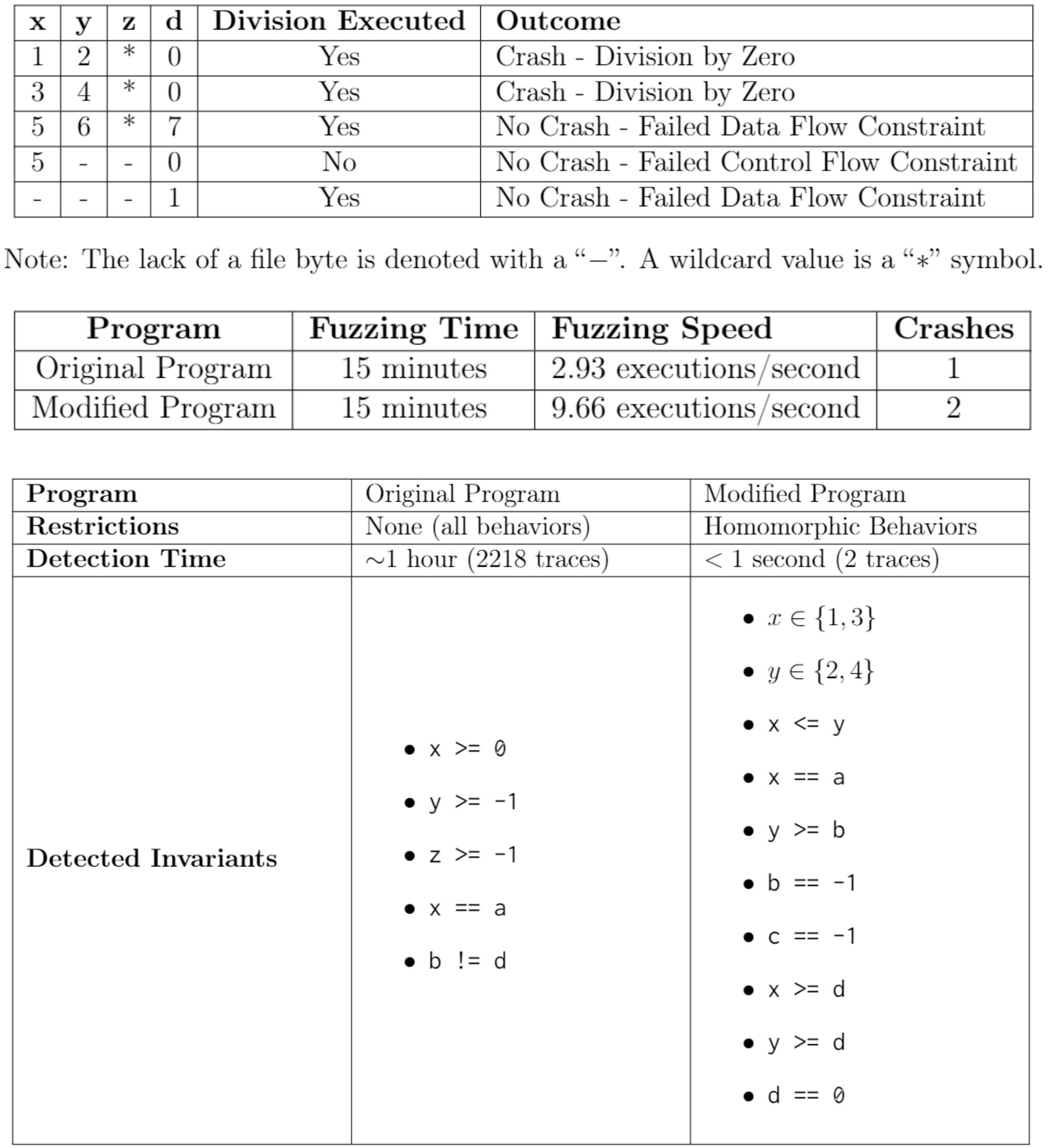
assert(y < 128)

assert (d==0)



```java
public class Toy {
   public static void main(String[] args) throws Exception {
      FileInputStream input = new FileInputStream(args[0]);
      int x = input.read() % 256; // x = -1 to 255
      int y = input.read() % 256; // y = -1 to 255
      int z = input.read() % 256; // z = -1 to 255

      int a = x;
      int b = y;
      int c = z;

      if(y < 128) {
         if(x > 5) {
            c = 0;
            expensive();
         }
         expensive();
         if(x < 5) {
            b = a - b;
         }
         c = b;
         int d = c + 1;
         if(y % 2 == 0) {
            System.out.println(x / d);
         } else {
            System.out.println(d);
            expensive();
         }
      }
      expensive();
   }
}
```

```java
public class Toy {
   public static void main(String[] args) throws Exception {
      FileInputStream input = new FileInputStream(args[0]);
      int x = input.read() % 256; // x = -1 to 255
      int y = input.read() % 256; // y = -1 to 255
      assert_relevant(y < 128 && y % 2 == 0);
      int z = input.read() % 256; // z = -1 to 255

      int a = x;
      int b = y;
      goto label_1;
      int c = z;

      label_1:
      if(y < 128) {
         goto label_2;
         if(x > 5) {
            c = 0;
            expensive();
         }
         expensive();
         label_2:
         if(x < 5) {
            b = a - b;
         }
         c = b;
         int d = c + 1;
         assert_relevant(d == 0);
         if(y % 2 == 0) {
            System.out.println(x / d);
            abort_relevant();
         } else {
            abort_irrelevant();
            System.out.println(d);
            expensive();
         }
      }
      abort_irrelevant();
      expensive();
   }
}
```

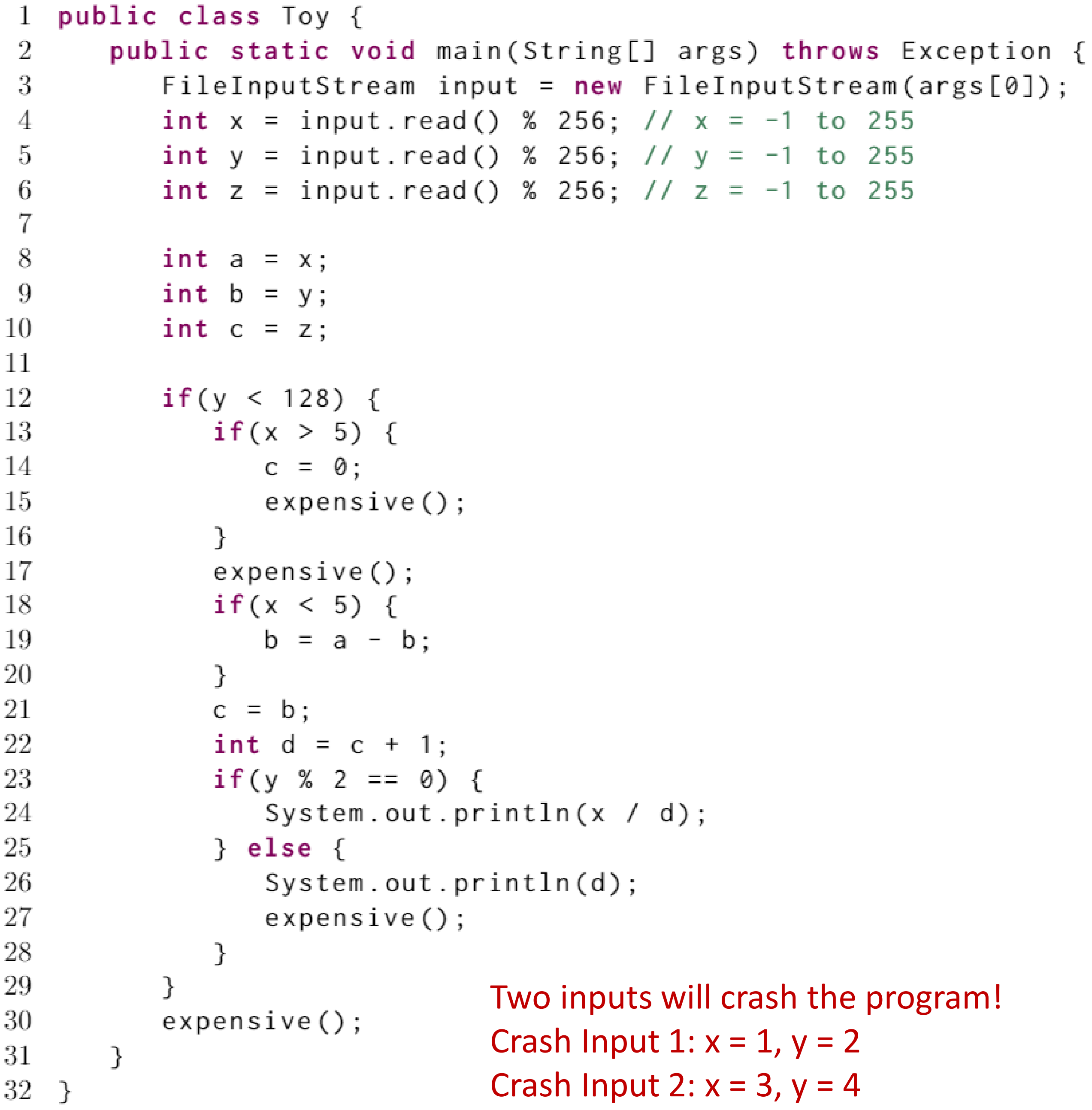


x	y	z	d	Division Executed	Outcome
1	2	*	0	Yes	Crash - Division by Zero
3	4	*	0	Yes	Crash - Division by Zero
5	6	*	7	Yes	No Crash - Failed Data Flow Constraint
5	-	-	0	No	No Crash - Failed Control Flow Constraint
-	-	-	1	Yes	No Crash - Failed Data Flow Constraint

Note: The lack of a file byte is denoted with a "–". A wildcard value is a "∗" symbol.

Program	Fuzzing Time	Fuzzing Speed	Crashes
Original Program	15 minutes	2.93 executions/second	1
Modified Program	15 minutes	9.66 executions/second	2

Program	Original Program	Modified Program
Restrictions	None (all behaviors)	Homomorphic Behaviors
Detection Time	~1 hour (2218 traces)	< 1 second (2 traces)
Detected Invariants	• x >= 0 • y >= -1 • z >= -1 • x == a • b != d	• $x \in \{1, 3\}$ • $y \in \{2, 4\}$ • x <= y • x == a • y >= b • b == -1 • c == -1 • x >= d • y >= d • d == 0



```java
public class Toy {
    public static void main(String[] args) throws Exception {
        FileInputStream input = new FileInputStream(args[0]);
        int x = input.read() % 256; // x = -1 to 255
        int y = input.read() % 256; // y = -1 to 255
        int z = input.read() % 256; // z = -1 to 255

        int a = x;
        int b = y;
        int c = z;

        if(y < 128) {
            if(x > 5) {
                c = 0;
                expensive();
            }
            expensive();
            if(x < 5) {
                b = a - b;
            }
            c = b;
            int d = c + 1;
            if(y % 2 == 0) {
                System.out.println(x / d);
            } else {
                System.out.println(d);
                expensive();
            }
        }
        expensive();
    }
}
```

Two inputs will crash the program!
Crash Input 1: x = 1, y = 2
Crash Input 2: x = 3, y = 4

x	y	z	d	Division Executed	Outcome
1	2	*	0	Yes	Crash - Division by Zero
3	4	*	0	Yes	Crash - Division by Zero
5	6	*	7	Yes	No Crash - Failed Data Flow Constraint
5	-	-	0	No	No Crash - Failed Control Flow Constraint
-	-	-	1	Yes	No Crash - Failed Data Flow Constraint

Note: The lack of a file byte is denoted with a "–". A wildcard value is a "∗" symbol.

Program	Fuzzing Time	Fuzzing Speed	Crashes
Original Program	15 minutes	2.93 executions/second	1
Modified Program	15 minutes	9.66 executions/second	2

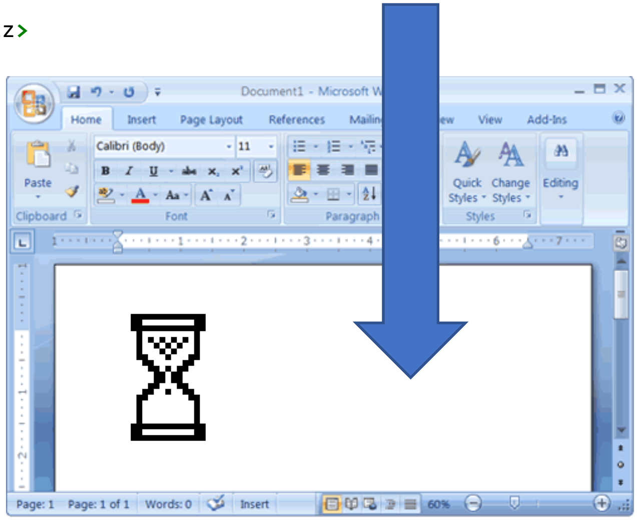
Program	Original Program	Modified Program
Restrictions	None (all behaviors)	Homomorphic Behaviors
Detection Time	~1 hour (2218 traces)	< 1 second (2 traces)
Detected Invariants	• x >= 0 • y >= -1 • z >= -1 • x == a • b != d	• $x \in \{1, 3\}$ • $y \in \{2, 4\}$ • x <= y • x == a • y >= b • b == -1 • c == -1 • x >= d • y >= d • d == 0

```java
public class Toy {
    public static void main(String[] args) throws Exception {
        FileInputStream input = new FileInputStream(args[0]);
        int x = input.read() % 256; // x = -1 to 255
        int y = input.read() % 256; // y = -1 to 255
        assert_relevant(y < 128 && y % 2 == 0);
        int z = input.read() % 256; // z = -1 to 255

        int a = x;
        int b = y;
        goto label_1;
        int c = z;

        label_1:
        if(y < 128) {
            goto label_2;
            if(x > 5) {
                c = 0;
                expensive();
            }
            expensive();
            label_2:
            if(x < 5) {
                b = a - b;
            }
            c = b;
            int d = c + 1;
            assert_relevant(d == 0);
            if(y % 2 == 0) {
                System.out.println(x / d);
                abort_relevant();
            } else {
                abort_irrelevant();
                System.out.println(d);
                expensive();
            }
        }
        abort_irrelevant();
        expensive();
    }
}
```

Case Study: BraidIt DARPA Challenge App

- Space/Time Analysis for Cybersecurity (STAC)
- Scenario: Detect algorithmic complexity (AC) and side channel (SC) vulnerabilities in a compiled bytecode applications
- Measured with respect to execution time or volatile/non-volatile memory space and an attacker input budget
    - Example: Send 1k byte request to cause 300 sec runtime execution
    - Example: Measure the response times of 100 requests to learn private key

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

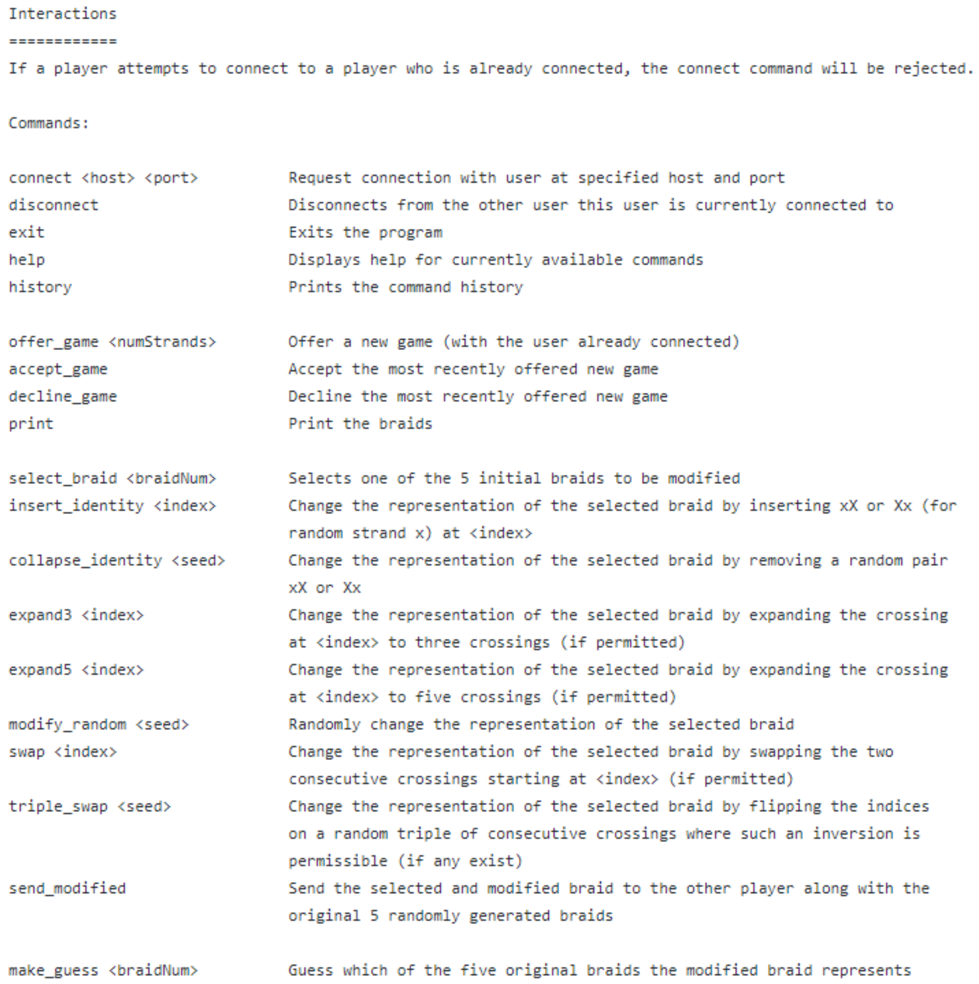


Case Study: BraidIt DARPA Challenge App

- Case study audit of a DARPA STAC challenge application
- Application was *not supposed* to be vulnerable…
    - But we didn’t know that…

Case Study: BraidIt DARPA Challenge App

- BraidIt is a peer-to-peer 2-player game that tests the players' ability to recognize topologically equivalent braids.
- BraidIt is based on the word equivalence problem in the Artin braid group. The application does all the dirty work, so users need not understand the theory and can treat it as a fun guessing game.

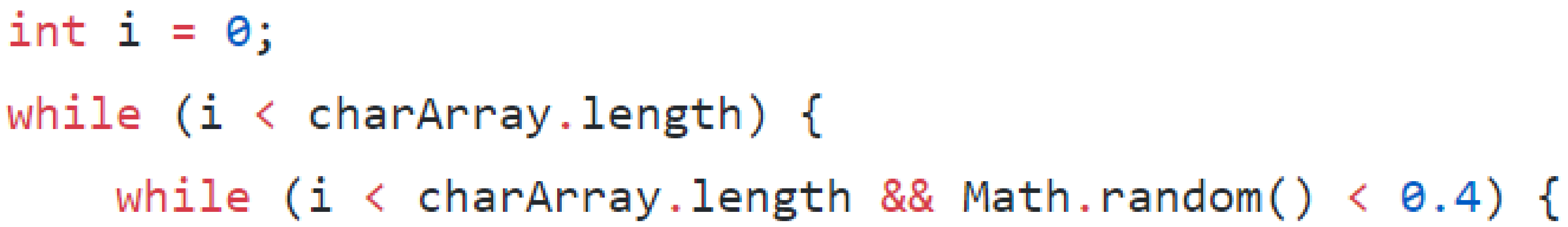


```
Interactions
============
If a player attempts to connect to a player who is already connected, the connect command will be rejected.

Commands:

connect <host> <port>        Request connection with user at specified host and port
disconnect                   Disconnects from the other user this user is currently connected to
exit                         Exits the program
help                         Displays help for currently available commands
history                      Prints the command history

offer_game <numStrands>      Offer a new game (with the user already connected)
accept_game                  Accept the most recently offered new game
decline_game                 Decline the most recently offered new game
print                        Print the braids

select_braid <braidNum>      Selects one of the 5 initial braids to be modified
insert_identity <index>      Change the representation of the selected braid by inserting xX or Xx (for
                             random strand x) at <index>
collapse_identity <seed>     Change the representation of the selected braid by removing a random pair
                             xX or Xx
expand3 <index>              Change the representation of the selected braid by expanding the crossing
                             at <index> to three crossings (if permitted)
expand5 <index>              Change the representation of the selected braid by expanding the crossing
                             at <index> to five crossings (if permitted)
modify_random <seed>         Randomly change the representation of the selected braid
swap <index>                 Change the representation of the selected braid by swapping the two
                             consecutive crossings starting at <index> (if permitted)
triple_swap <seed>           Change the representation of the selected braid by flipping the indices
                             on a random triple of consecutive crossings where such an inversion is
                             permissible (if any exist)
send_modified                Send the selected and modified braid to the other player along with the
                             original 5 randomly generated braids

make_guess <braidNum>        Guess which of the five original braids the modified braid represents
```

Case Study: BraidIt DARPA Challenge App

- Is there an algorithmic complexity vulnerability in space that would cause the challenge program to store files with combined logical sizes that exceed the resource usage limit given the input budget?
- Input Budget: Maximum sum of the PDU sizes of the application requests sent from the attacker to the server: 2 kB (measured via sum of the length fields in tcpdump)
- Resource Usage Limit: Available Logical Size: 25 MB (logical size of output file measured with 'stat’)
- Probability of Success: 99%

Application Was Hardened Against Fuzzing

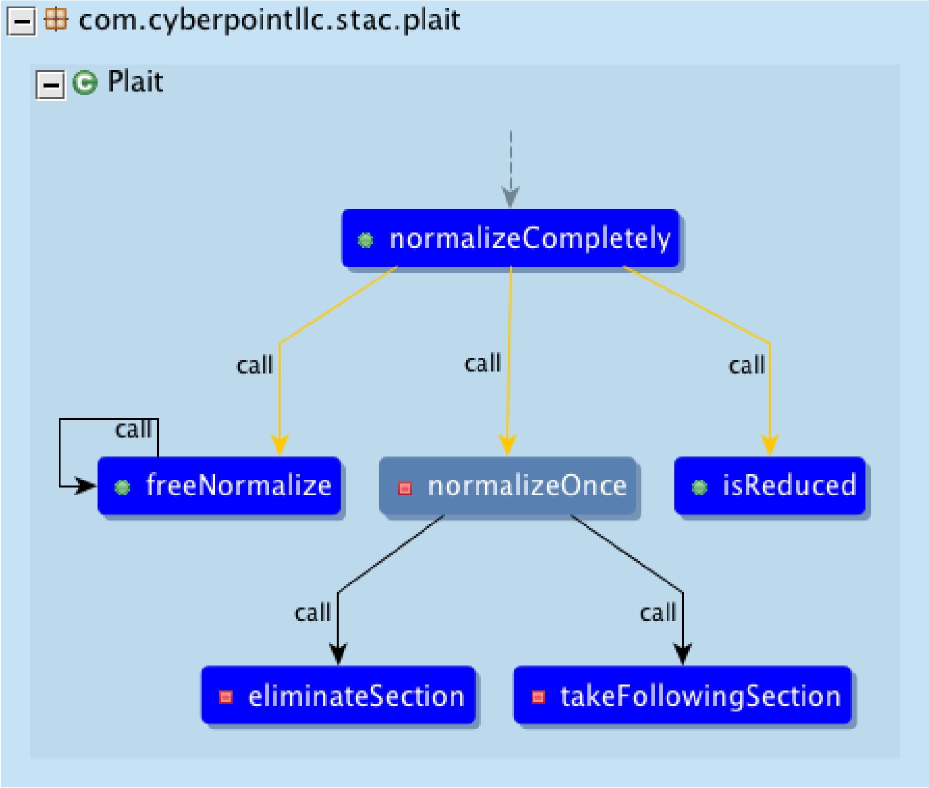


```
int i = 0;
while (i < charArray.length) {
    while (i < charArray.length && Math.random() < 0.4) {
```

Identification of an Interesting Loop

- *freeNormalize* and *normalizeOnce* each write to log file
- *normalizeCompletely* is an instance method that depends on existing program state
- The termination of *normalizeCompletely* depends on the result of *isReduced*
- Involves loop nesting and recursion

```
public void normalizeCompletely() {
    this.freeNormalize();
    while (!this.isReduced()) {
        this.normalizeOnce();
        this.freeNormalize();
    }
}
```



Identification of an Interesting Loop

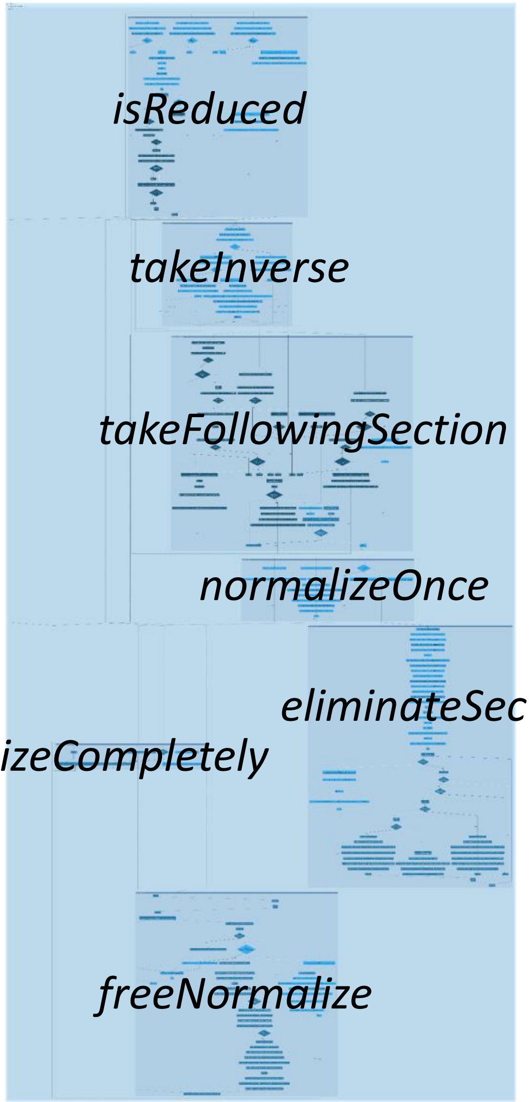
- *isReduced* reads a global variable called *intersections*
- *freeNormalize* and *normalizeOnce* update *intersections*
- *intersections* is a string variable that is initially attacker controlled

Hypothesis: *Can there be a string that has the property of being irreducible and therefore cause an infinite loop that writes to the file system?*

Complex String Operations

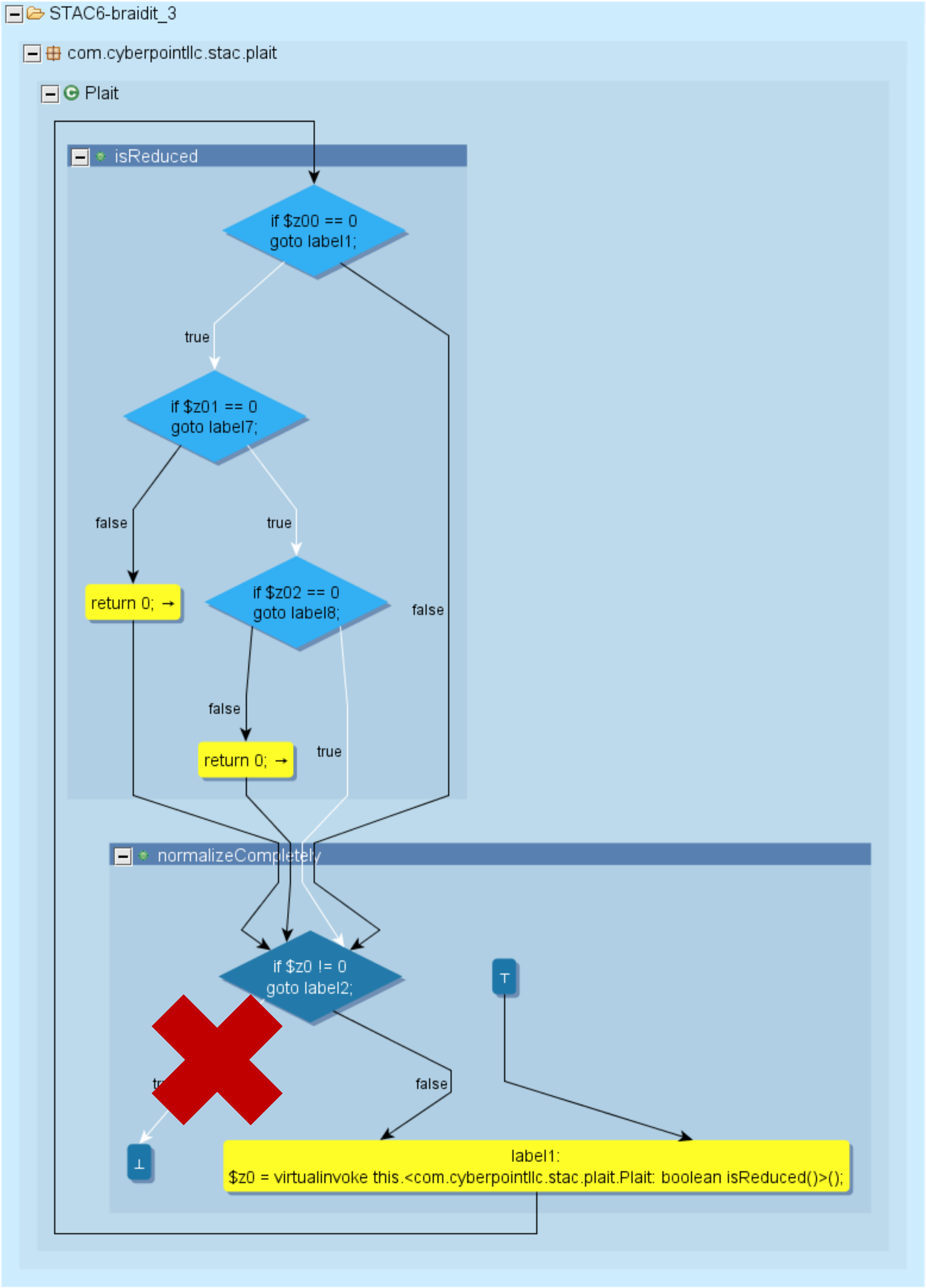
- 9 unique string operations in 62 locations
    - 20 of which are within loops
- 8 unique character level operations in 60 locations
    - 27 locations are within loops

Inter-procedural Control Flow Graph



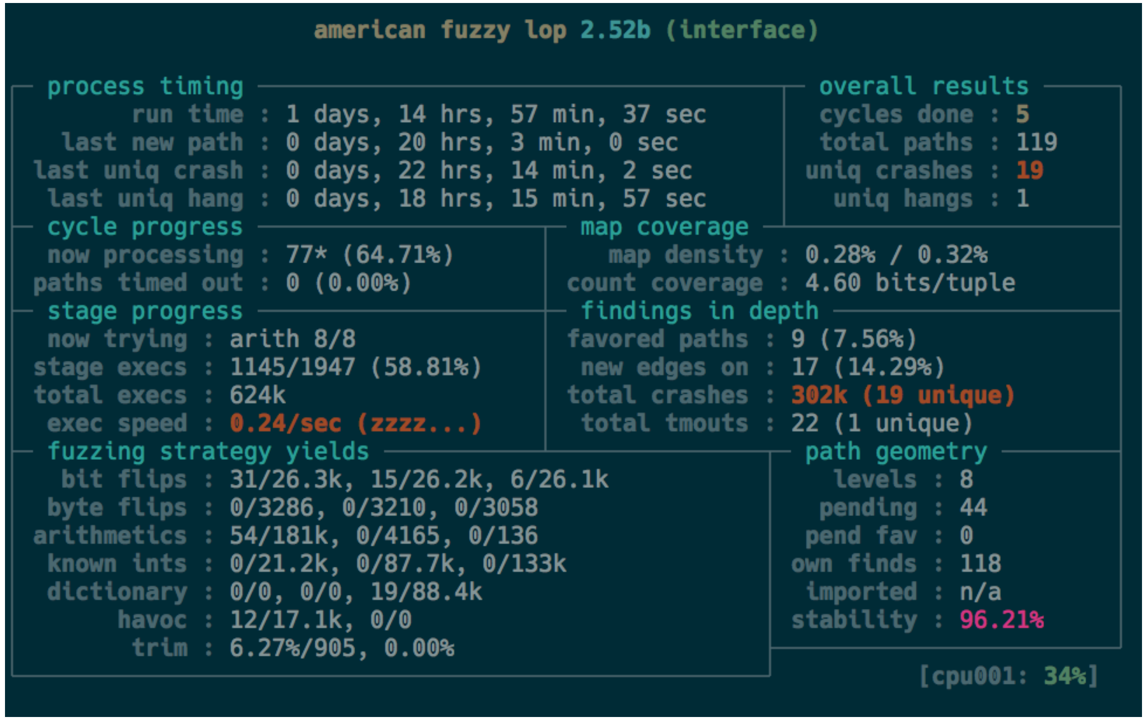
Relevant Paths

- What do we care about?
  - The loop should not exit
  - isReduced() should always return false
  - If an input gets reduced then abort immediately


STAC6-braidit_3
com.cyberpointllc.stac.plait
Plait
isReduced
if $z00 == 0 goto label1;
true
if $z01 == 0 goto label7;
false
true
return 0; →
if $z02 == 0 goto label8;
false
false
return 0; →
true
normalizeCompletely
if $z0 != 0 goto label2;
⊤
tr
false
⊥
label1:
$z0 = virtualinvoke this.<com.cyberpointllc.stac.plait.Plait: boolean isReduced()>();

Targeted Dynamic Analysis of *normalizeCompletely*

- Initial experiment: 20 hours of directly fuzzing *normalizeCompletely*…
    - H. Invariant: *isReduced* is always false
    - H. Invariant: *intersections* is always a non-empty string
- Input: “ËĎġčçęêďªã”
    - What does this mean?



```
                    american fuzzy lop 2.52b (interface)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 1 days, 14 hrs, 57 min, 37 sec      │  cycles done : 5     │
│   last new path : 0 days, 20 hrs, 3 min, 0 sec        │  total paths : 119   │
│ last uniq crash : 0 days, 22 hrs, 14 min, 2 sec       │ uniq crashes : 19    │
│  last uniq hang : 0 days, 18 hrs, 15 min, 57 sec      │   uniq hangs : 1     │
├─ cycle progress ─────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 77* (64.71%)        │    map density : 0.28% / 0.32%        │
│ paths timed out : 0 (0.00%)           │ count coverage : 4.60 bits/tuple      │
├─ stage progress ─────────────────────┼─ findings in depth ───────────────────┤
│  now trying : arith 8/8               │ favored paths : 9 (7.56%)             │
│ stage execs : 1145/1947 (58.81%)      │  new edges on : 17 (14.29%)           │
│ total execs : 624k                    │ total crashes : 302k (19 unique)      │
│  exec speed : 0.24/sec (zzzz...)      │  total tmouts : 22 (1 unique)         │
├─ fuzzing strategy yields ─────────────┴───────────┬─ path geometry ──────────┤
│   bit flips : 31/26.3k, 15/26.2k, 6/26.1k          │    levels : 8             │
│  byte flips : 0/3286, 0/3210, 0/3058               │   pending : 44            │
│ arithmetics : 54/181k, 0/4165, 0/136               │  pend fav : 0             │
│  known ints : 0/21.2k, 0/87.7k, 0/133k             │ own finds : 118           │
│  dictionary : 0/0, 0/0, 19/88.4k                   │  imported : n/a           │
│       havoc : 12/17.1k, 0/0                        │ stability : 96.21%        │
│        trim : 6.27%/905, 0.00%                     ├───────────────────────────┘
└────────────────────────────────────────────────────┘          [cpu001: 34%]
```

Refined Experiment: Constrained Fuzzing

- Plait Constructor does some complex validation on *intersections*, which end with the following checks
  - Checks that each character is alphabetic
  - Checks that each character’s lowercase character is greater than 122 + *numStrands* + 2
  - *numStrands* is attacker controlled input between 8 and 27
- Experiment: Iterate over strings of the alphabet described by constructor
  - 20 minutes to find smallest malicious inputs +13 more…
- Minimal Input: “ªªª”
  - What does this mean?

Refined Experiment: Homomorphic Invariants

H. Invariant: *isReduced* is always false

H. Invariant: *intersections* is always a non-empty string

H. Invariant: *intersections* contains a common subsequence of a single character ‘ª’

Refined Hypothesis: *A property of the character ‘ª’ can be used to create an irreducible string that causes an infinite loop that writes to the file system.*

Reasoning with Homomorphic Invariants

- Debug with the minimal input “ªªª” and pay attention character level operations
  - *freeNormalize* method removes a pair of case insensitive matching characters where one character is the first character in the string (leaving a single character ‘ª’ remaining)
  - *isReduced* method can return false if the string contains an uppercase character of a lowercase character
  - Uppercase(ª) == Lowercase(ª)
  - A fine scheme for ASCII, but Java Strings support Unicode UTF-16 standard…
    - There are 395 UTF-16 characters that alphabetic and lowercase is their uppercase
    - Any of the 395 could be used to craft an exploit (<u>we have identified a family of exploits!</u>)

Thank you!

- Questions?
- Slides: ben-holland.com

References

- [1] https://en.wikipedia.org/wiki/List_of_programming_languages
- [2] https://github.blog/2018-11-08-100m-repos/
- [3] https://archiveprogram.github.com/
- [4] https://www.developer-tech.com/news/2017/jul/05/linux-kernel-412-developers-added-795-lines-code-hour/
- [5] https://www.linuxfoundation.org/blog/2017/10/2017-linux-kernel-report-highlights-developers-roles-accelerating-pace-change/
- [6] https://cybersecurityventures.com/application-security-report-2017/
- [7] https://www.visualcapitalist.com/millions-lines-of-code/

References

- [8] Li, Frank, and Vern Paxson. "A large-scale empirical study of security patches." *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.
- [9] Bilge, Leyla, and Tudor Dumitraş. "Before we knew it: an empirical study of zero-day attacks in the real world." *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012.
- [10] http://pages.kennasecurity.com/rs/958-PRK-049/images/Kenna-NonTargetedAttacksReport.pdf
- [11] https://www.rand.org/content/dam/rand/pubs/research_reports/RR1700/RR1751/RAND_RR1751.pdf
- [12] https://www.fireeye.com/blog/threat-research/2015/04/angler_ek_exploiting.html

References

- [13] http://info.nopsec.com/rs/736-UGK-525/images/NopSec_StateofVulnRisk_WhitePaper_2015.pdf
- [14] Yin, Zuoning, et al. "How do fixes become bugs?."Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011
- [15] https://security.google-blog.com/2019/05/queue-hardening-enhancements.html
- [16] https://www.sciencedirect.com/topics/engineering/defect-density