Fuzz Testing (Fuzzing)

Mohammadreza Hazhirpasand

Software Composition Group

What I want to share

- Software testing → How it can help
  The drawbacks
- Fuzzing → History of fuzzing
  Fuzzing in software development
  Basics of fuzzing
  Demo: Radamsa – Blab – Spike - Burb suite - zzuf
  AFL + demo
  Symbolic execution and SMT solvers
  Concolic execution
  Hybrid fuzzing
- Wrap up → Fuzzing resources

Software testing

- Why?
- Suppose the following Python program:

The Newton–Raphson method

Python's math module

```
import math

#computes the square root of X
def my_sqrt(x):
    approx = None
    guess = x / 2
    while approx != guess:
        print ("Approx" + str(approx))
        approx = guess
        guess = (approx + x / approx) / 2
    return approx

print my_sqrt(454.0)
print math.sqrt(454.0)
```

Software testing

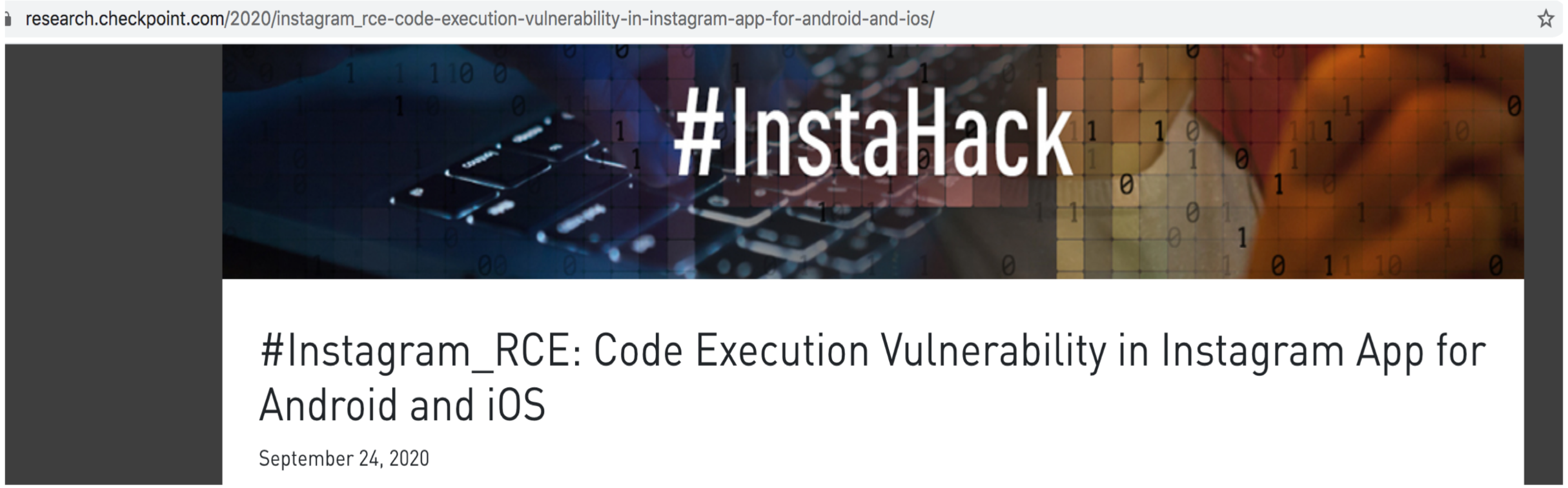
Testing is incomplete because:

1. A finite set of inputs can be checked
2. The correctness of a result is commonly important
3. Test results are used to make business decisions for release dates
4. We cannot be certain that all features of a method are tested
5. When Inputs become complex, it becomes harder to test
6. Time-consuming
7. Adversarial mindset is needed to extensively test the target

The history of fuzzing

- In 1988
- Prof. Barton Miller, University of Wisconsin
- The lightning-induced noise on his network connection caused common UNIX commands to crash
- A class project with the term “fuzzing”

Fuzzing is a way of discovering bugs in software by providing randomized/pattern-based inputs to programs to find test cases that cause a crash.

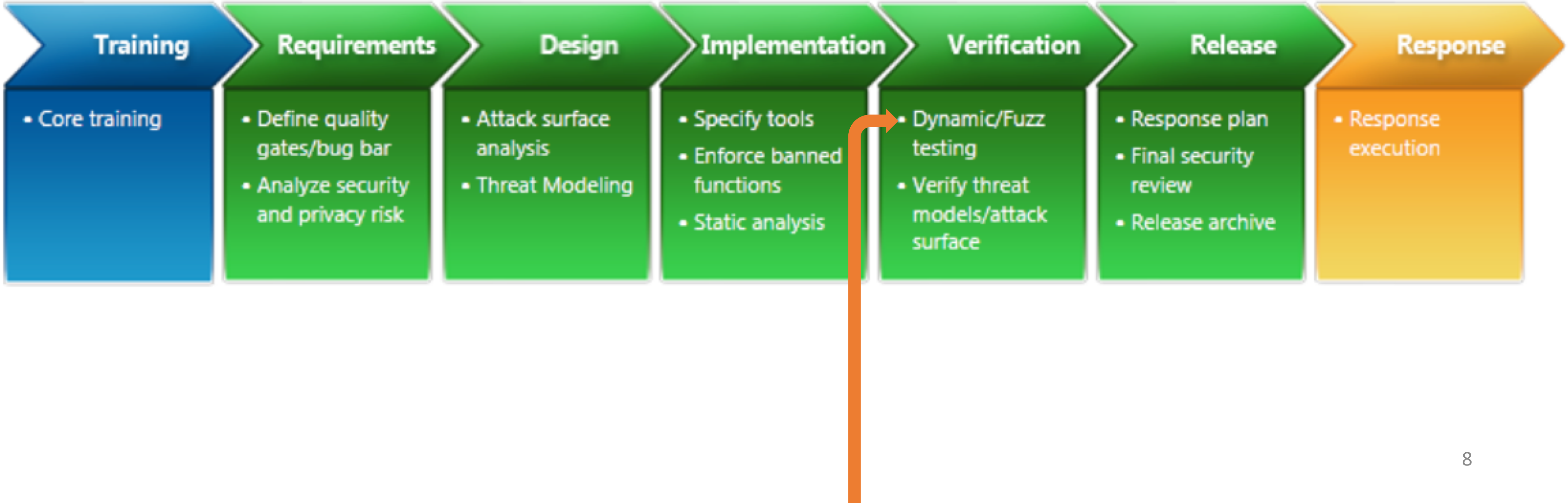


Goal of fuzzing

- To ensure certain bad things never occur (crashes, thrown exceptions)
- Such bad things can lay the cornerstone for security vulnerabilities
- However, sometimes such issues are the security vulnerabilities
- To complement functional testing

When to conduct fuzz testing?

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



Fuzzers are either…

- File-based: mutate or generate inputs and see what happens
- Network-based: act as a man-in-the-middle and mutate inputs exchanged between parties

Smart or dumb?

- A fuzzer that generates completely random input is known as a "dumb" fuzzer
- A fuzzer with knowledge of the input format is known as a "smart" fuzzer

Kinds of fuzzing

- Black box → The tool knows nothing about the target and its input
  Easy to use
  Explore only shallow states
- White box → Generates new inputs by program analysis and constraint solving
  Easy to use (relatively)
  Computationally expensive
- Grey box → Generates new inputs by some knowledge of the program
  Easy to use (relatively)
  Computationally expensive

Fuzzing inputs can be …

- Mutation → A valid input is mutated randomly to produce malformed input
  Dumb fuzzing / Smart fuzzing
- Replay → Replaying the captured messages
- Generation → Generate input from scratch - grammar
  Only mutates randomly a chunk of an input
- Evolutionary → Use feedback from each test case to learn the format of the input
  Code coverage

Terminology: *code coverage*

- In program analysis, code coverage is a standard metric that describes how much of the code is exercised
- However, higher code coverage does not imply more bugs ☹, but it certainly increases the likelihood of finding one ☺
- In scientific papers, researchers attempt to prove the efficiency of their proposed fuzzer by either code coverage or bug coverage

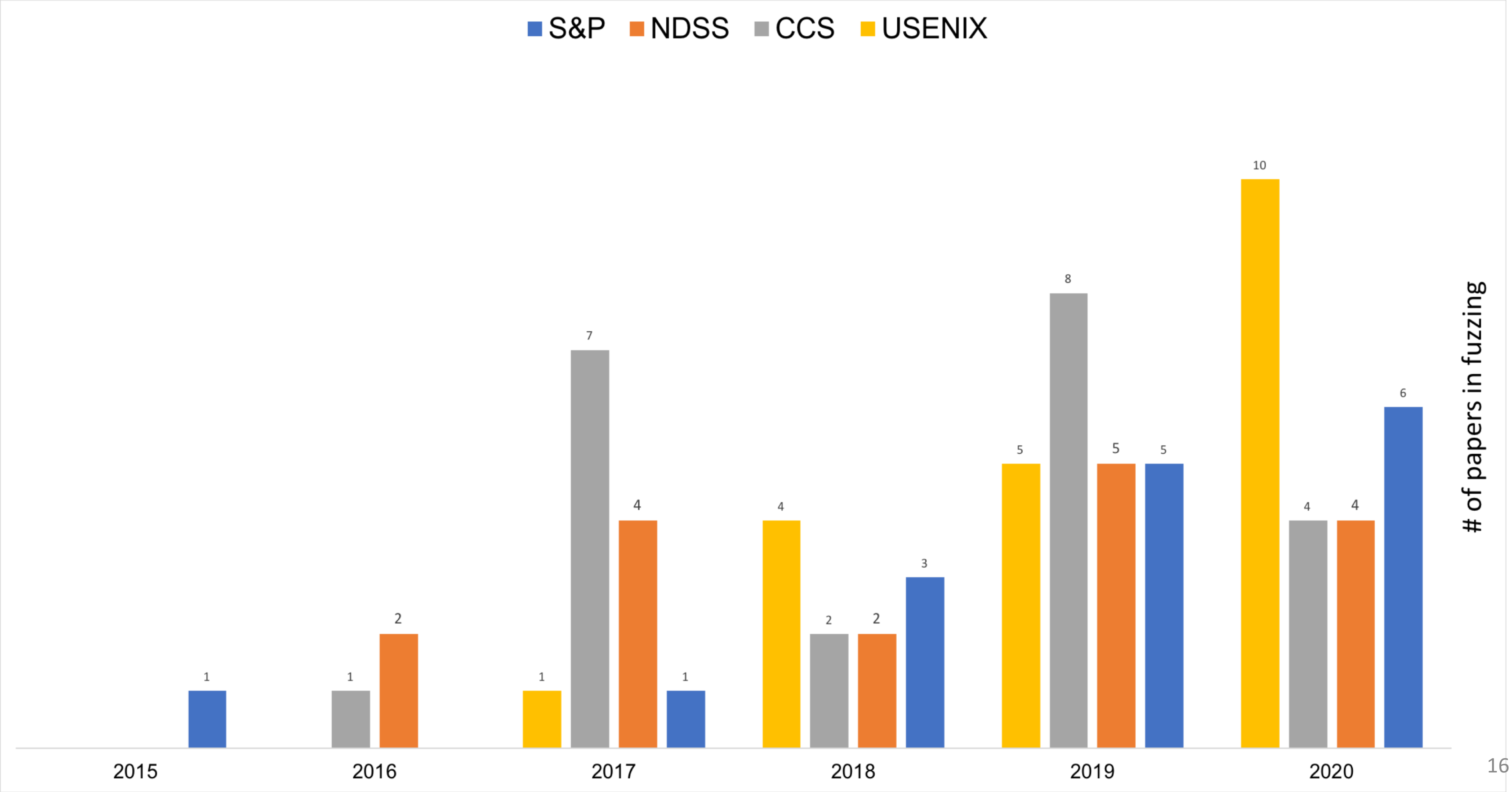
Vulnerable friends!

- Protocol → TCP, DNS, FTP, …
- File format → MP3, JPEG, PNG, …
- User input → Names, addresses, file names,
- Programming lang... → JavaScript, PHP

A fuzzer’s skeleton

- Test case generation → Completely blank or long strings, null character, max and min values for integers
- Reproducibility → Record test cases and associated information
- Crash detections → Attach a debugger, process disappears, timeouts

Fuzzing in conferences



Fuzzing in competition

- CGC – Cyber Grand Challenge – created by DARPA

Give some examples please! ☺

Radamsa

- Radamsa is a mutation-based, black box fuzzer
- Radamsa performs dumb mutation on inputs

Blab

- Blab generates inputs according to a grammar

- The grammar can be specified as regexps or CFGs

zzuf

- Zzuf is a simple, lightweight, and deterministic tool
- Bug reproducibility is easy
- It intercepts file operations and modifies random bits in the program's input

SPIKE

- SPIKE is a fuzzer creation kit
- SPIKE provides an C language API for fuzzing network protocols

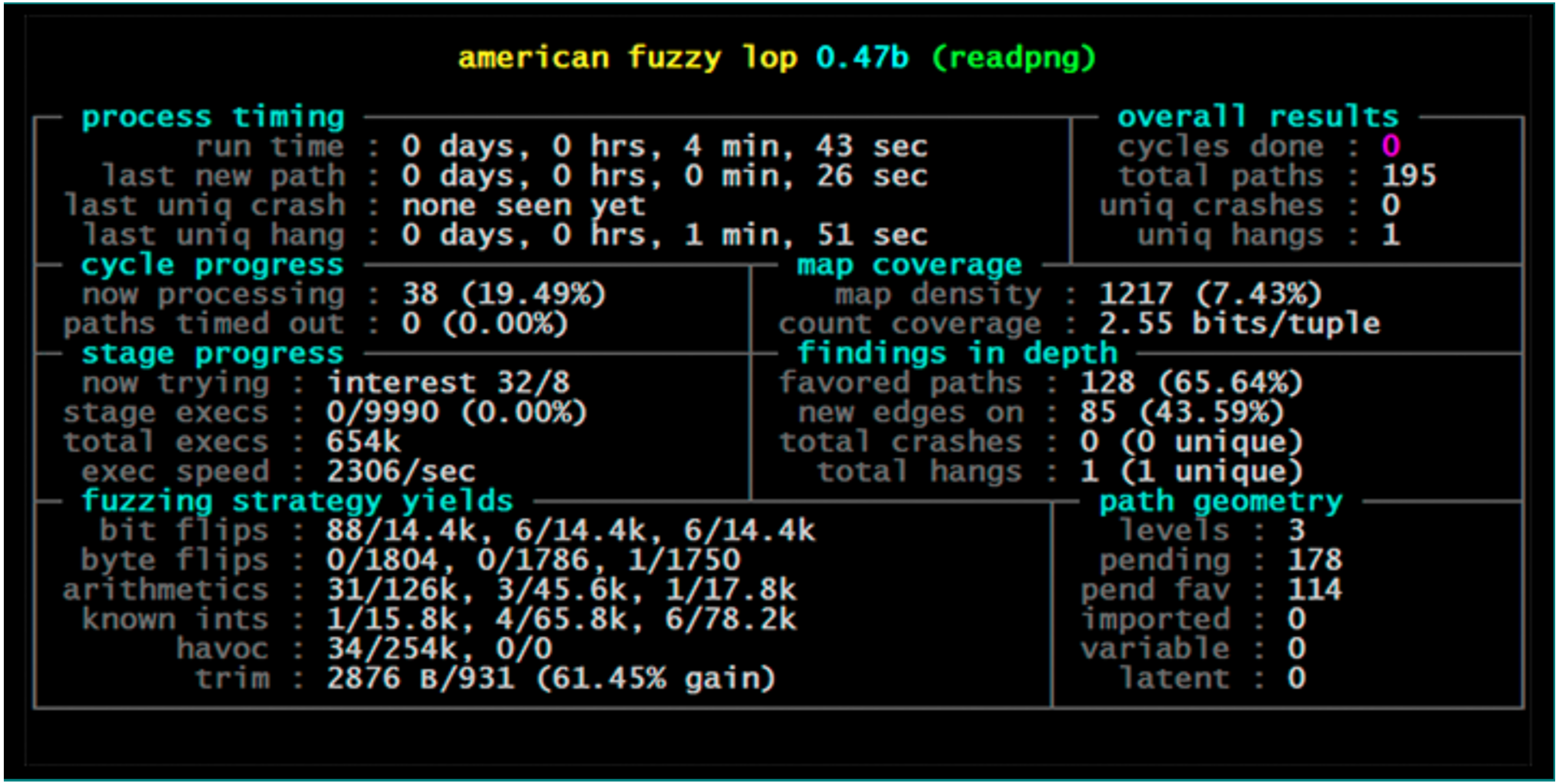
Burp intruder

- Burp orchestrates hand-crafted attacks against web applications

- Users can benefit from other features of burp suites, e.g., proxy, spider

BURPSUITE

AFL – American Fuzzy Lop



AFL – American Fuzzy Lop

- Michal Zalewski, 2013
- First practical high performance guided fuzzer
- Compile-time instrumentation and genetic algorithms
- A tuple of <ID of current code location, ID last code location>
- Many bugs!

AFL – American Fuzzy Lop

More than 20 forks of AFL:

1. AFL++
2. WinAFL
3. AFLsmart
4. AFLGo
5. FairFuzz
6. AFLnet
7. …

https://github.com/Microsvuln/Awesome-AFL

AFL – American Fuzzy Lop

DEMO

There is always a problem…

- The indomitable spirit of mutation-based fuzzers is questionable as …

How can mutation-based fuzzers solve such constraints? ☹

```
int check(uint64_t magic) {
  if ((magic ^ 0x9cfbd61bad9abad9) + (magic * 0xa68977238907ef1e))  == 939)
  {
  return 1;
  }
return 0;
}
```

Symbolic execution: history

- In 1976, Symbolic execution and program testing
- As a means of program verification to prove the program's correctness
- From the formal verification to vulnerability analysis of the program
- 2005-present: practical symbolic execution (using SMT solvers)

Terminology: *SMT solvers*

- SMT or Satisfiability Modulo Theories
- An SMT formula is a Boolean combination of formulas over first-order theories
- Example of SMT theories include arrays, integer and real arithmetic, strings, ...
- Outcome → SAT(+ model) → if F is satisfiable
  unsat → if F is unsatisfiable

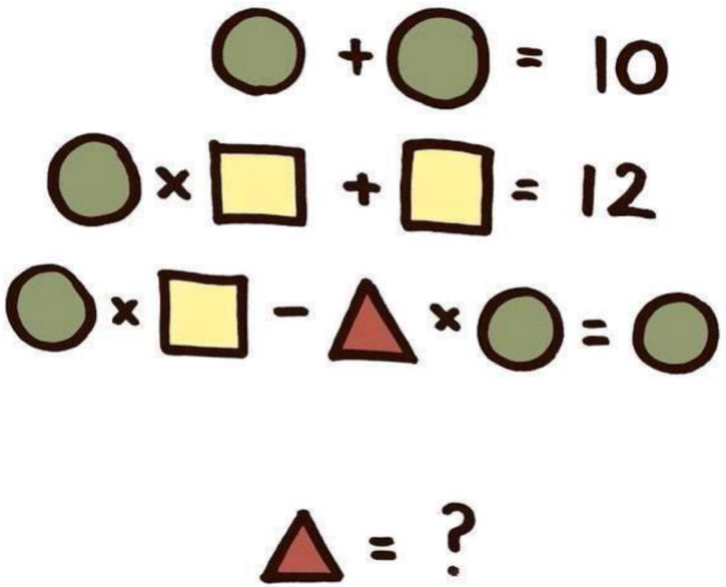
Terminology: *SMT solvers*

- Z3 is a high-performance theorem prover, developed at Microsoft Research

  https://github.com/Z3Prover/z3

```
#!/usr/bin/python
from z3 import *
circle , square , triangle = Ints('Enter three inputs')
s = Solver()
s.add(circle+circle==10)
s.add(circle*square+square==12)
s.add(circle*square -triangle*circle==circle)
print s.check()
print s.model()


=> sat
=> [triangle = 1, square = 2, circle = 5]
```



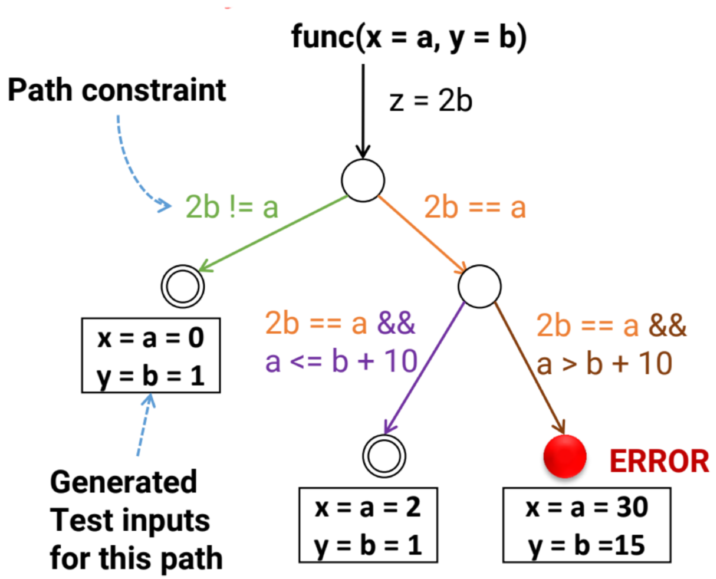
Symbolic Execution engines

- KLEE: a dynamic symbolic execution engine built on top of the LLVM compiler – OSDI 2008
- SAGE: Scalable, Automated, Guided Execution – NDSS 2008
- More: jCUTE (Java), Kleenet (sensor networks), Angr, S2E, many others…

Symbolic Execution - example

- Traditional fuzzers fail to exercise all possible behaviors
- Execute the program with symbolic valued
- Generate new inputs at each branch to cover all parts of code

```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
            if (x > y + 10)
                ERROR
    }
}
int main(){
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;
}
```



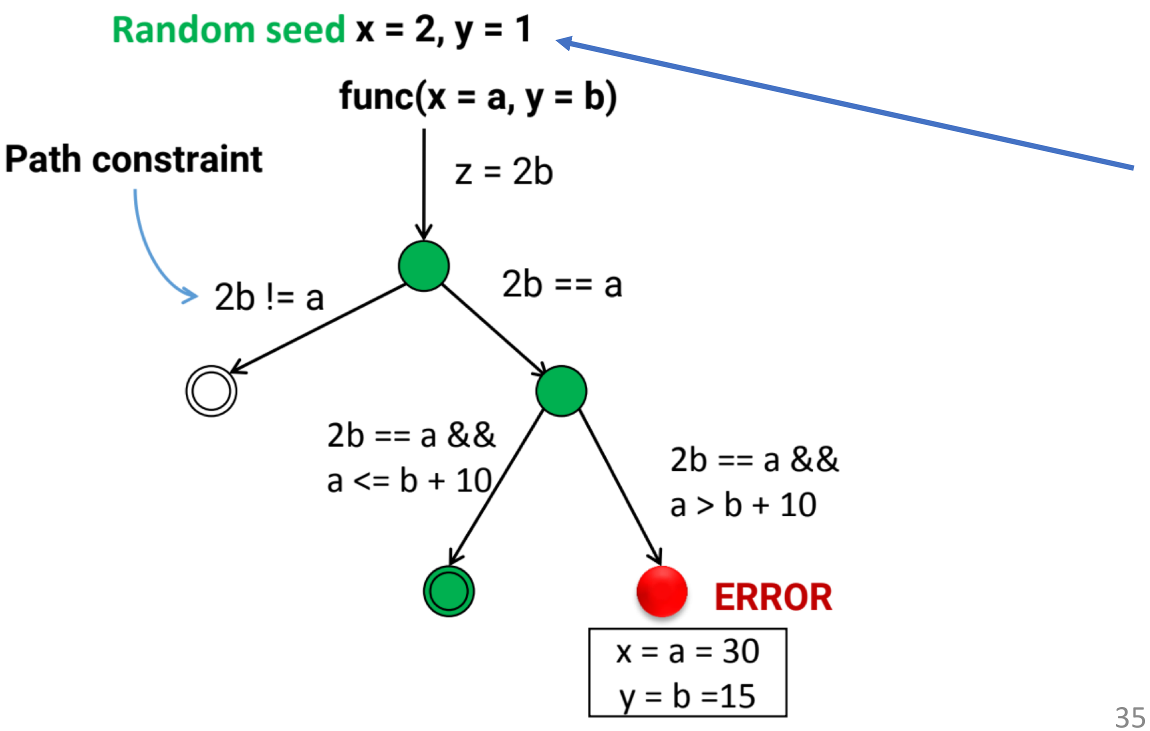
Symbolic Execution - limitations

- **Path explosion**: symbolically executing all feasible program paths does not scale to large programs
- **Loops and recursions**: infinite execution tree
- **SMT solver limitations**: dealing with complex path constraints

Concolic Execution Engines – ~~Symbolic execution~~

- **Concolic** = **Conc**rete + Symb**olic** *(dynamic symbolic execution)*
- A Program is executed with concrete (random inputs) and symbolic inputs

```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
            if (x > y + 10)
                ERROR
    }
}
int main(){
    int x = input();
    int y = input();
    func(x, y);
    return 0;
}
```

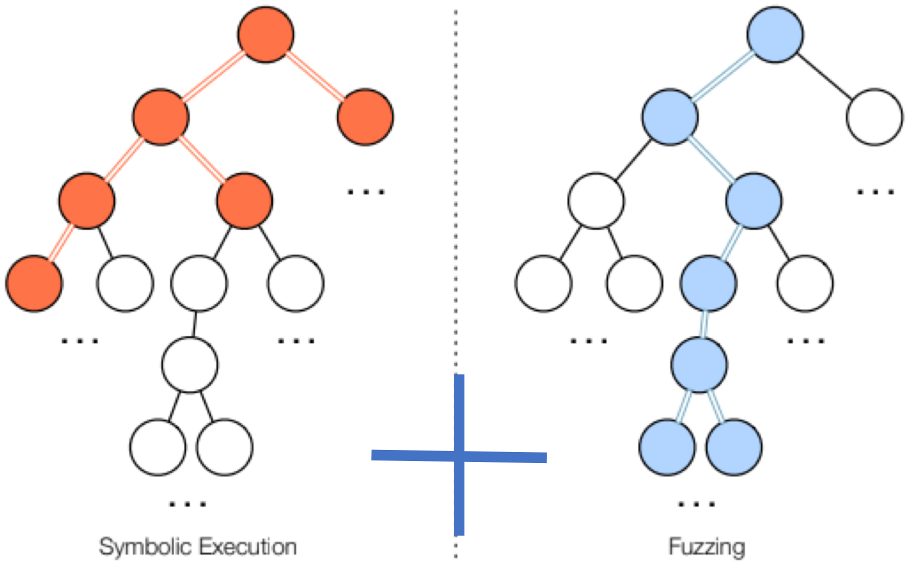


Concolic Execution engines

- QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing - USENIX 2018
- Symbolic execution with SymCC: Don't interpret, compile! - USENIX 2020
- Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing - CCS 2019
- Eclipser : Grey-box Concolic Testing on Binary Code - ICSE 2019
- Driller: Augmenting Fuzzing Through Selective Symbolic Execution- NDSS 2016
- SAVIOR: Towards Bug-Driven Hybrid Testing - S&P 2019

Traditional fuzzing vs. symbolic execution

- The drawback of symbolic execution is its impracticality for real-world cases
- Traditional fuzzing is way faster and explores deeper parts of the code
- However, traditional fuzzing has limited code coverage in breadth



Hybrid fuzz testing

- To combine the two aforementioned approaches to achieve better results
- Hybrid fuzz testing is commonly composed of

basic block profiling + symbolic execution + input generation + guided random fuzzing

Code coverage | To increase breadth of covered code | Generate random inputs | Monitor program’s state

Fuzzing resources

- The Fuzzing Book -- https://www.fuzzingbook.org
- Fuzzing: Brute Force Vulnerability Discovery
- Fuzzing for Software Security Testing and Quality Assurance
- https://github.com/Microsvuln/Awesome-AFL

Now you should know

- What is fuzzing and why?
- What is code coverage?
- What is a (black box) || (white box) || (grey box) fuzzer?
- What is hybrid fuzzing?
- How can symbolic execution help fuzzers?