University of Sheffield

COM3529 Software Testing and Analysis

Fuzzing

Dr José Miguel Rojas

BB-DL-JK

Roadmap

Beizer's Maturity Model

Test Automation

Unit Testing

Control/Data-Flow Analysis

Code Coverage

Mutation Testing

Regression Testing

Fuzzing

Search-based Test Generation

Model-Based Testing

BB-DL-JK

Roadmap

Beizer's Maturity Model

Test Automation

Unit Testing

Control/Data-Flow Analysis

Code Coverage

Mutation Testing

Regression Testing

Fuzzing

Search-based Test Generation

Model-Based Testing

BB-DL-JK

Motivation

One dark, stormy night in 1990, Bart Miller (U Wisc.) was logged to work in over dial-up

There was a lot of line noise due to the storm

His shell and editors kept crashing

This gave him an idea…

Motivation

Idea: Feed "**fuzz**" – streams of pure randomness to OS & utility code, e.g., noise from **`/dev/random`**

Watch them break!

This would crash 25-33% of utilities

Reports every few years since then

Some of the bugs are the same ones in common security exploits (specially buffer overruns)

Bart Miller et al., "An Empirical Study of the Reliability of UNIX Utilities"

Fuzzing

Software developers make mistakes

Mistakes -> bugs -> security vulnerabilities

Fuzzing aims to send malformed/random input to a program and hope to crash it or find a security hole

Useful for finding bugs to protect programs – “white hat” work

But also for finding bugs to hack into systems – “black hat”!

Used to find (security) problems in many well-known companies / products, e.g., Apple, Firefox

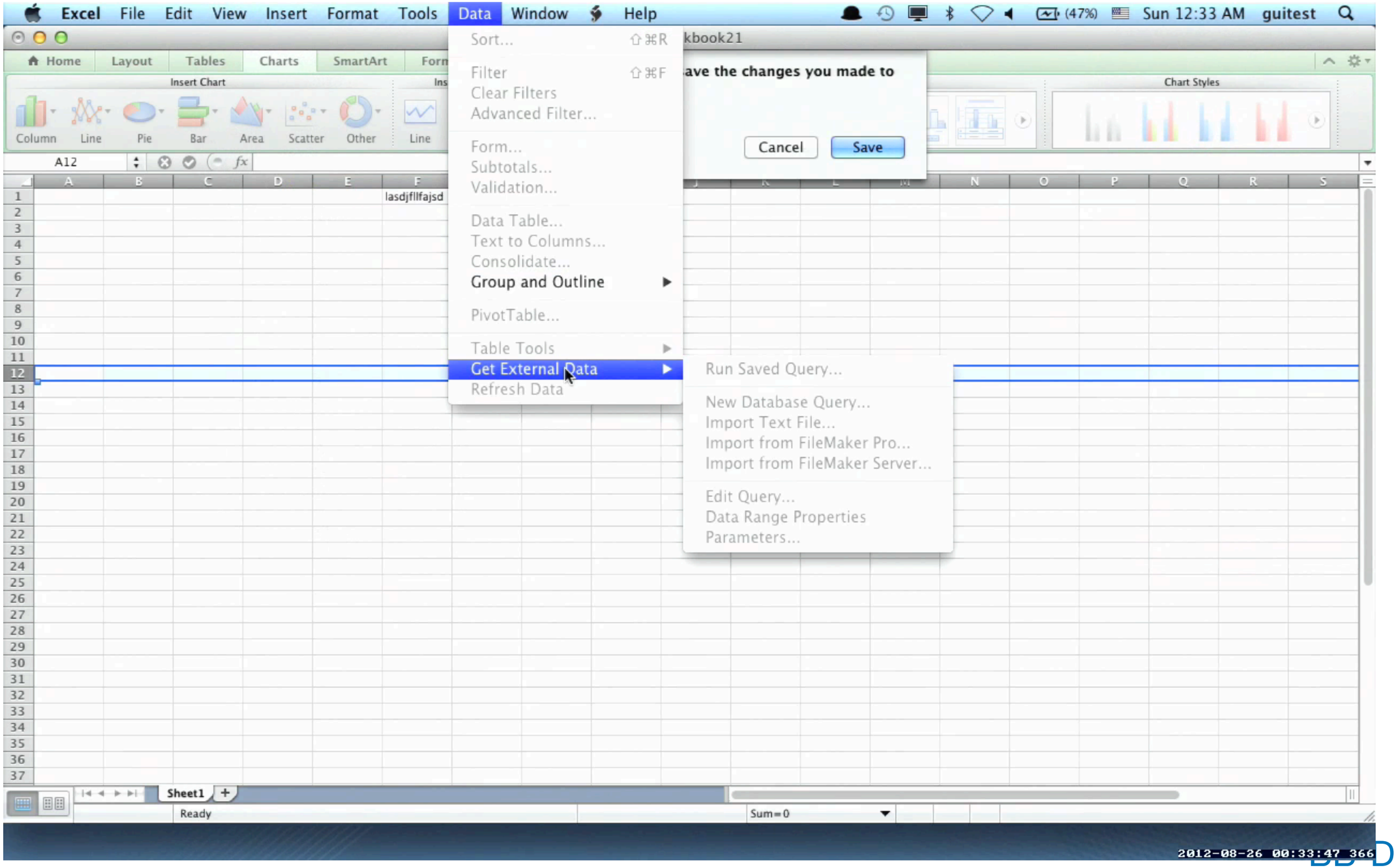
User Testing vs Fuzzing

Run program on multiple **normal** inputs, looking for bad things to happen

Goal: Prevent **normal users** from encountering errors

Run program on multiple **abnormal** inputs, looking for bad things to happen

Goal: Prevent **attackers** from encountering **exploitable** errors


Excel File Edit View Insert Format Tools Data Window Help
Sun 12:33 AM guitest
Sort... ⇧⌘R
Filter ⇧⌘F
Clear Filters
Advanced Filter...
Form...
Subtotals...
Validation...
Data Table...
Text to Columns...
Consolidate...
Group and Outline
PivotTable...
Table Tools
Get External Data
Refresh Data
Run Saved Query...
New Database Query...
Import Text File...
Import from FileMaker Pro...
Import from FileMaker Server...
Edit Query...
Data Range Properties
Parameters...
ave the changes you made to
Cancel
Save
Home
Layout
Tables
Charts
SmartArt
Insert Chart
Column
Line
Pie
Bar
Area
Scatter
Other
Chart Styles
A12
lasdjfllfajsd
Sheet1
Ready
Sum = 0
2012-08-26 00:33:47 366

Fuzzing

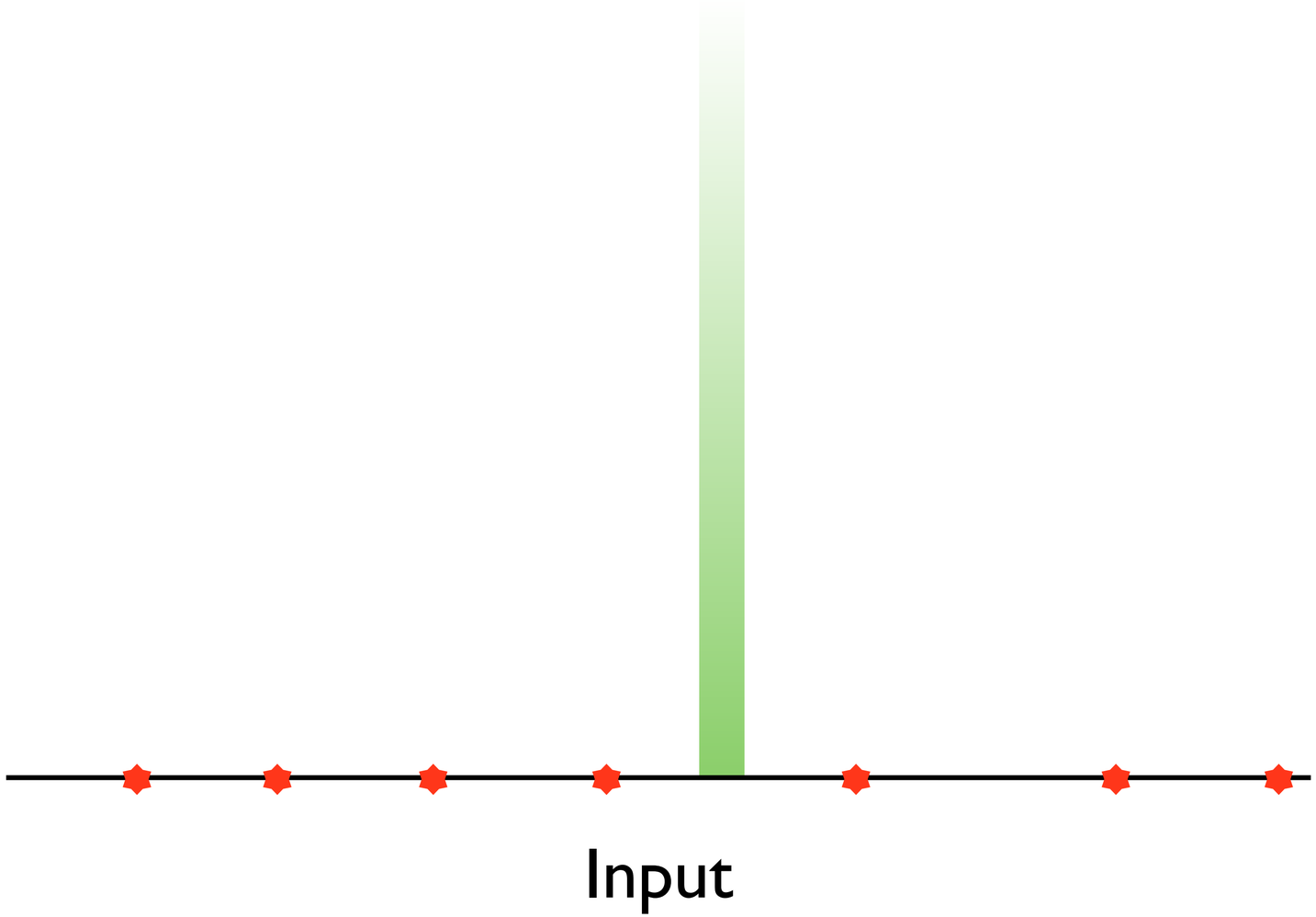
Automatically generate **random** (malformed) test cases

Monitor application for errors

Nature of inputs can be diverse

- Files: .pdf, .png, .wav, .mpg, .json, etc
- Network based: http, SOAP, SNMP

Likelihood of Finding Bugs?



How to Fuzz?

Very long inputs, very short inputs, blank input, …

Min/max values of integers, zero and negative values

Special values, characters or keywords likely to trigger bugs, e.g., nulls, newlines, or end-of-file characters

Format string characters %s %x %n

Semi-colons, slashes and backslashes, quotes, non-printable characters, …

Application-specific keywords, e.g., halt, return, DROP TABLES, …

Template-based Fuzzing

Also known as **Mutation-based**

Programs requiring **complex input** cannot properly be tested by random input

Example: JavaScript interpreter needs valid JavaScript programs as test inputs

Instead of random inputs, **modify (mutate)** existing, valid inputs

Mutations may be random or guided by heuristics

Minimal set up time, but technique is dependant on target programs

BB-DL-JK

Coverage-based Fuzzing

If the fuzzer happens to create a valid, useful input, e.g., covering a new branch, use that as a starting point for further mutations

Also known as: **white-box fuzzing**

Grammar-based Testing

How to generate complex **textual** inputs?

Classical application: Testing compilers

Nowadays: Web applications, interpreters, parsers, …anything that uses XML, JSON, etc.

Microsoft's **SAGE** tool (Godefroid et al, 2008a) is reported to have saved the company millions of dollars' worth of software bugs.

Context-free Grammars

Finite set of *terminals*

Finite set of *non-terminals*

Finite set of *rules*

Each *rule*: *Non-terminal* → List of *terminals* and *non-terminals*

Starting rule

Example Grammar

```
expr → expr op expr
expr → ( expr )
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑
```

Derivation

Interpret production as rewriting rule

Non-terminal on the left hand side is replaced by string on the right hand side

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

Sequence of replacements = derivation

Two choices at each derivation step:

- Which non-terminal to replace
- Which alternative to use for the replaced non-terminal

Grammar-based Testing

Test generation → rewriting based on grammar

Begin with start symbol

Replace one non-terminal on the right with a rule with the non-terminal on the left

Repeat until only terminals are left

Result is a **test input**

```
expr → expr op expr
expr → ( expr )
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑
```

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

Problems

Context-free grammars → recursion

Recursion → **infinite** number of possible inputs

Which rule to replace with next?

Grammar Annotation

Limit recursion depth

Limit number of occurrences

Limit parse tree depth

```
{count 3} zeros;
zeros ::= ‘0’;
zeros ::= ‘0’ zeros;
```

Covering Grammars

Production coverage

Each production rule must be used to generate at least one (section of) test case

Boundary condition

Annotate each recursive production rule with minimum and maximum number of application, then generate:

Minimum
Minimum + 1
Maximum - 1
Maximum

A Combinatorial Problem

Testing VoIP software…

- Caller, VoIP server, client
- CallerOS: Windows, Mac
- ServerOS: Linux, Sun, Windows
- CalleeOS: Windows, Mac

Same Problem as Grammar

It is possible to use covering arrays to test with grammars

```
Call ::= CallerOS ServerOS CalleeOS;
CallerOS ::= ‘Mac’;
CallerOS ::= ‘Win’;
ServerOS ::= ‘Lin’;
ServerOS ::= ‘Sun’;
ServerOS ::= ‘Win’;
CalleeOS ::= ‘Mac’;
CalleeOS ::= ‘Win’;
```

Tools

LibFuzzer (https://llvm.org/docs/LibFuzzer.html),

AFL (https://github.com/google/AFL)

AFL++ (https://github.com/AFLplusplus)

Spike (https://www.kali.org/tools/spike/)

Peach (https://wiki.mozilla.org/Security/Fuzzing/Peach)

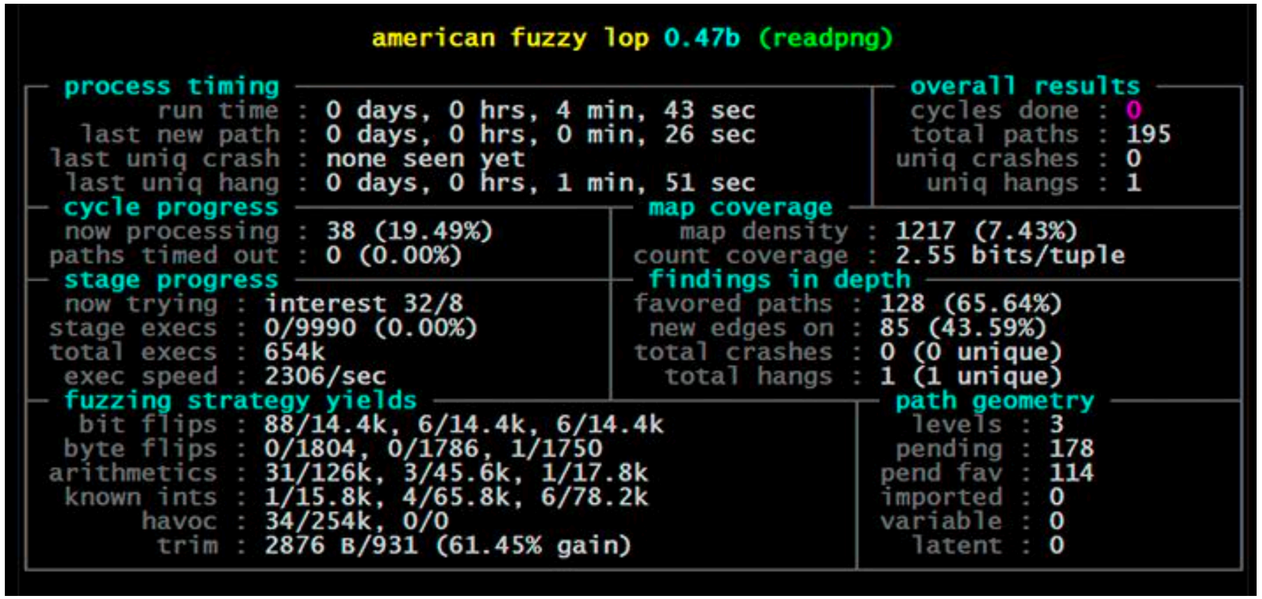
OneFuzz (https://github.com/microsoft/onefuzz)

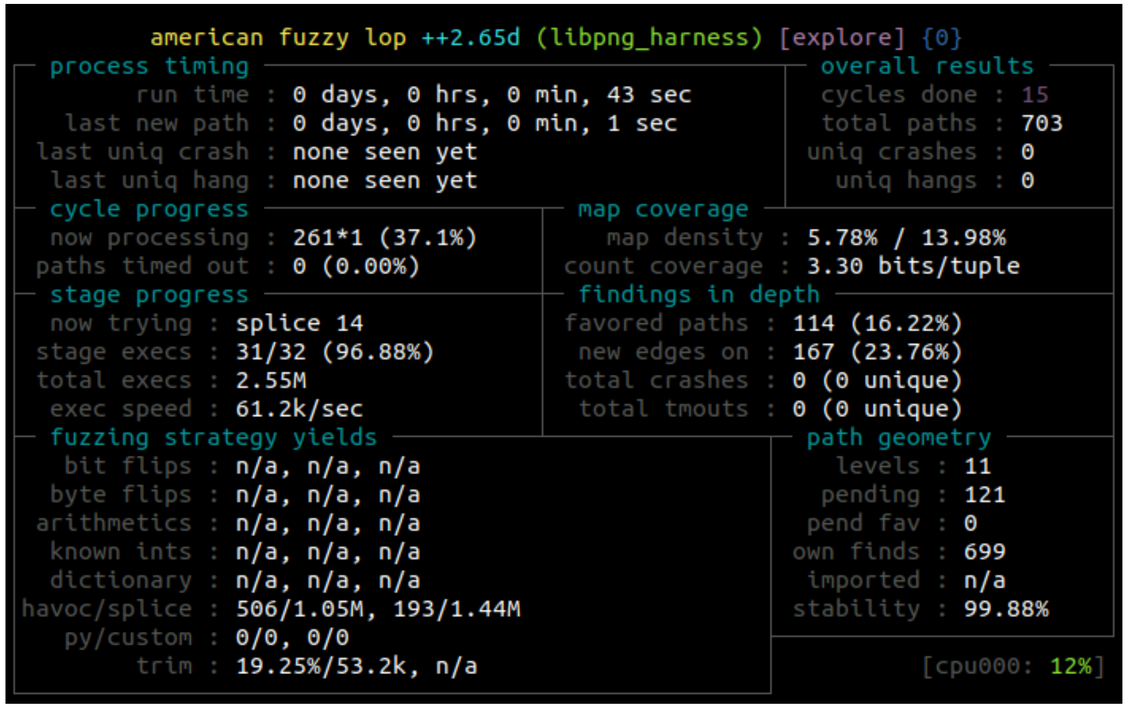
HonggFuzz (https://github.com/google/honggfuzz)

AFL (American Fuzzy Lop)++

Feedback-based fuzzer

Mutate file – Does it add coverage? Keep it; else delete





BB-DL-JK

Summary

Simple testing techniques such as **fuzzing** can be very effective

Relatively **easy to implement**

Can be costly – **when to stop?** – and hard to assess w.r.t. coverage

Several tools available, e.g., AFL

Excellent resource: The Fuzzing Book – https://www.fuzzingbook.org/