Fuzzing

20210209

Outline

Introduction

Fuzzing : your code is buggier than mine

It's about testing

What is fuzzing ?

At its core, fuzzing is random testing.

A short (selective) pre-history

1981 Random testing is a cost-effective alternative to systematic testing techniques (Duran & Natos)

1983 "The Monkey" (Capps)

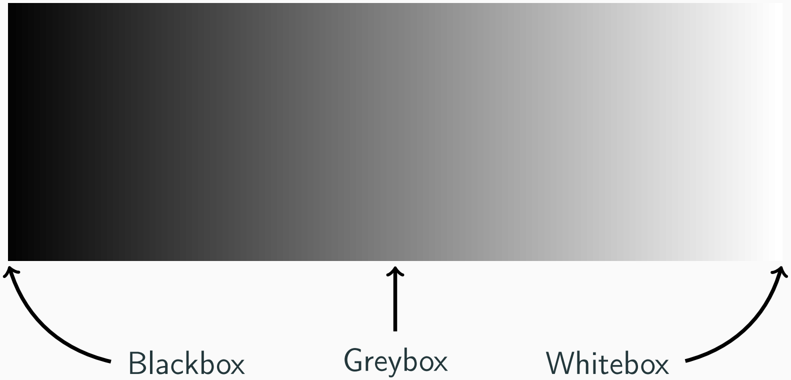
1988 Birth of the term "fuzzing" (Miller)

The initial assignment

> *The goal of this project is to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream. [. . .] First, you will build a fuzz generator. This is a program that will output a random character stream. Second, you will take the fuzz generator and use it to attack as many UNIX utilities as possible, with the goal of trying to break them.*

1/3 of Unix utilities crashed, hung or failed upon fuzzing inputs.

Shades of fuzzing



Blackbox

Key property

A blackbox fuzzer is unaware of the program structure

Because of that, blackbox fuzzers are necessarily limited. They thus can be considered dumb.

Whitebox

Key distinction

A whitebox fuzzer leverages program analysis to reach its targets:

Usual targets are:

1. code coverage;
2. program location.

This is usually synonym with "Dynamic Symbolic Execution".

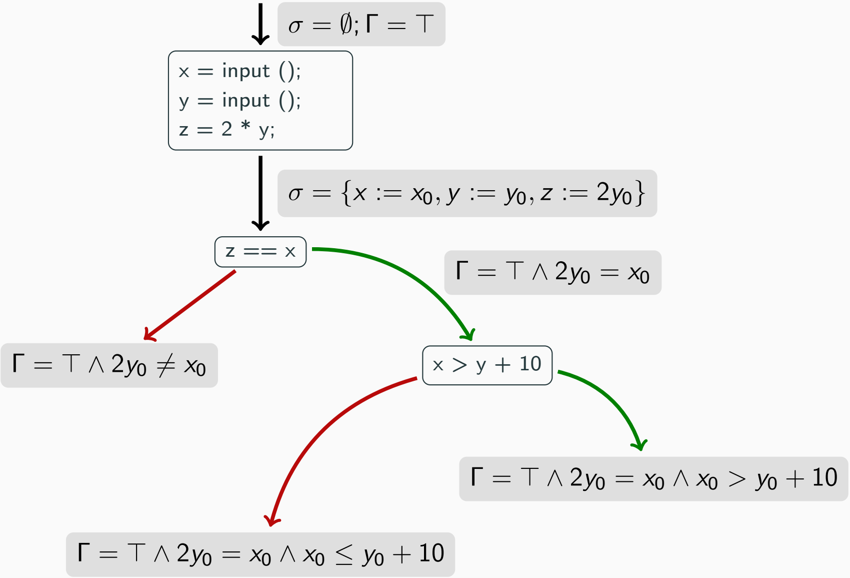
Whitebox fuzzers are smart.

Coverage criteria

```c
void
f(int a, int b, int * x)
{
  if (a > 1 && b == 0)
    *x = *x / a;
  if (a == 2 || *x > 1)
    *x = *x + 1;
}
```

- Instructions (I)
- All decisions (D)
- All simple conditions (C)
- All conditions / decisions (DC)
- All combinations of conditions (MC)
- All paths (P)

Wait ? Whitebox fuzzers ?



Greybox

Greybox fuzzers favor leveraging instrumentations instead of program analysis.

Principles of fuzzing

Stages

1. Preprocess
2. Scheduling
3. Input Generation
4. Input Evaluation
5. Configuration Updating
6. Continue

General algorithm (Manes et al. 2019)

Input: $\mathbb{C}$, t_{limit}

Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$

$\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$

while $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$ do

  $\texttt{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{elapsed}, t_{limit})$

  $\texttt{tcs} \leftarrow \text{InputGen}(\mathit{conf})$

  // O_{bug} is embedded in a fuzzer

  $\mathbb{B}', \texttt{execinfos} \leftarrow \text{InputEval}(\mathit{conf}, \mathit{tcs}, O_{bug})$

  $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \mathit{conf}, \mathit{execinfos})$

  $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$

return $\mathbb{B}$

Stages (summarized) i

Preprocess

- User supplies set of fuzz configurations as input, gets a potentially-modified set of fuzz configurations.
- May perform a variety of actions
  - insert instrumentation code to Program Under Tests (PUT),
  - measure execution speed of seed files
  - etc.

Stages (summarized) ii

Schedule

- Takes in :
  - current set of fuzz configurations
  - current time, and
  - a timeout
- Selects a fuzz configuration to be used for the current fuzz iteration.

Stages (summarized) iii

Input Generation

- Takes in a fuzz configuration
- Returns a set of concrete test cases.
- Some fuzzers use a seed in for generating test cases, while others use a model or grammar as a parameter.

Stages (summarized) iv

Input Evaluation

- Takes in :
  - a fuzz configuration
  - a set of test cases, and
  - a bug oracle.
- Executes PUT on test cases and checks if executions violate the security policy using the bug oracle.
- Outputs set of bugs found $\mathbb{B}'$ and information about each of the fuzz runs.

Stages (summarized) v

Configuration Update

- Takes in:
  - a set of fuzz configurations,
  - current configuration, and
  - the information of each fuzz runs.
- May update the set of fuzz configurations.
  For example, many grey-box fuzzers reduce the number of fuzz configurations based on fuzz runs.

Stages (summarized) vi

Continue

- Takes a set of fuzz configurations as input and outputs a boolean indicating whether a next fuzz iteration should happen or not.
- This models white-box fuzzers that can terminate when there are no more paths to discover.

Preprocess: Instrumentation

Goal

Gather execution feedback during runs

Types of instrumentation

Static usually at compile time, sometimes rewriting the binary (e.g., `afl-gcc`)

Dynamic more costly but can instrument dlls (e.g., `afl-qemu`)

Preprocess: Seed Selection

Goal

Find minimal set that maximizes a coverage metric

Examples of metrics

AFL	branch coverage with logarithmic counters on each branch
Honggfuzz	# executed instructions, branches, unique basic blocks

Scheduling Problem

Goal

Pick the configuration that is the most likely to lead to the most favorable outcome.

- finding most unique bugs
- maximizing coverage

Exploration vs Exploitation

Scheduling balances:

- time gathering more information on configurations (exploration) and
- time fuzzing configurations believed to lead to favorable outcomes (exploitation).

Scheduling: Blackbox

Blackbox fuzzers can only use fuzz outcomes (time spent, bugs found).

Examples

- Favoring the configuration with a high success rate (#bugs/#runs)
- Prefer faster configurations (collection of information on them is quicker) & fix the time per configurations selection instead the number of runs (avoid spending a lot of time in slow configuration).

Scheduling: Greybox i

AFL

- Maintain a population of configurations with a fitness metric, apply some degree of genetic transformation (mutation, recombination)
- On a control-flow edge, AFL considers fastest and smallest inputs as "fitter"
- Fix number of run per selection.

Scheduling: Greybox ii

AFLFast

- On a control-flow edge, favor the one that has been chose least
- On tie, favor the one that exercises path that has been selected the least
- The fuzzing time follows a power schedule

Input generation: Generation-based

Predefined model

- Network protocols
- EBNF
- System calls (types and number of arguments)

Inferred

- Synthesize the grammar of the parser
- Capture packets and infer network protocols
- Observe I/O behavior and infer state machine
- Machine learning

Input generation: Mutation-based

- Use initial seeds providing a structure of a valid input (file, network packets, . . .)
- Mutate portions of previous inputs to generate new *mostly valid* test cases.

Bit-flip

Basics

Flip:

- a fixed-number of bits or
- a random number of bits

Mutation ratio

Determines the number of bit flips for a single generated input.

A good mutation ratio can be inferred through program analysis.

Arithmetic mutation

Basics

Perform an arithemtic operation on a sequence of bytes seen as an integer.

Example

AFL selects 4-bytes values i, generates a random integer r and applies $i + r$ or $i - r$.

The range of r is tunable.

Block-based mutation

Block

Arbitrary sequence of bytes

Mutations

1. Insert a new block at a random position
2. Delete a randomly selected block
3. Replace a randomly selected block
4. Permutes the order of block sequences
5. Resize a seed by appending a block
6. Take a random block from another seed to insert/replace

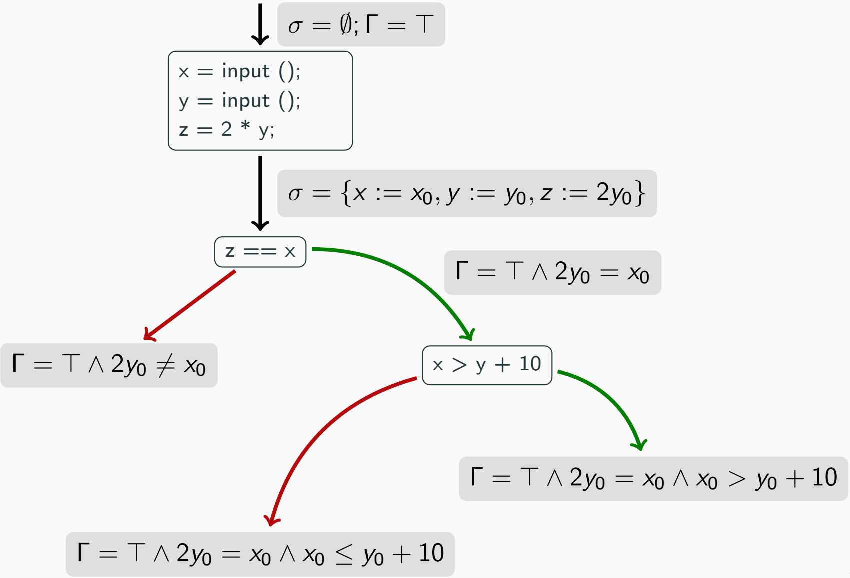
Dictionary-based mutation

Some fuzzers use "magic" values like 0 or -1, or format strings for mutation.

Examples

AFL	Uses 0, 1, -1 for integer mutations
Radamsa	unicode strings
GPF	`%x`, `%s` for string mutations

What about whitebox fuzzing?



Input Evaluation: Bugs

Default policy

A program execution terminated by a fatal signal is a violation

This is enough for memory vulnerabiliies since they usually trigger segmentation faults.

Problem

The default policy will not detect corruptions leading to valid addresses.

To this effect, code sanitizers are needed, i.e., a form of runtime monitor that will trigger a fatal signal on property violation.

Input Evaluation: Triage i

Deduplication

Prune test cases triggering the same bugs as another one.

Implemented by stack backtrace hashing or coverage-based deduplication (AFL).

Prioritization

Rank or group violating test cases according to severity and uniqueness, aka determines a form of exploitability.

`!exploitable`: 4-rank scales (yes, probably, unknown, not likely).

Input Evaluation: Triage ii

Test case minimization

Identify the part of the test case that is necessary to trigger the violation.

Produce a smaller test case (e.g., AFL sets bytes to 0 and shorten the test case).

This is not specific to fuzzers and thus dedicated techniuqes have been developed and can be reused here.

Configuration Updating

Evoluting the seed pool

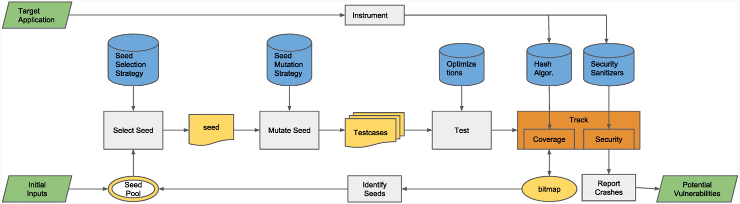
Select the fittest.

Example

- Use on node/branch coverage: a newly discovered branch/node is sign of fitness !
- AFL also takes into account the number of times a branch has been taken
- Angora also adds calling context
- Steelix checks input offsets affect the progress in comparison instructions

A concrete example : AFL

The AFL loop



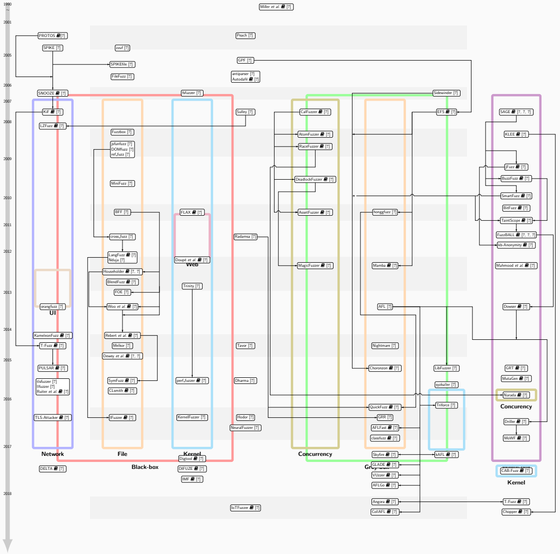
Target Application
Instrument
Seed Selection Strategy
Seed Mutation Strategy
Optimiza tions
Hash Algor.
Security Sanitizers
Select Seed
seed
Mutate Seed
Testcases
Test
Track
Coverage
Security
Initial Inputs
Seed Pool
Identify Seeds
bitmap
Report Crashes
Potential Vulnerabilities

Efficiency

clang / llvm 1 2 3 4 5 6 7 8 ...	nasm 1 2	ctags 1
mutt 1	procmail 1	fontconfig 1
pdksh 1 2	Qt 1 2...	wavpack 1 2 3 4
redis / lua-cmsgpack 1	taglib 1 2 3	privoxy 1 2 3
perl 1 2 3 4 5 6 7...	libxmp	radare2 1 2
SleuthKit 1	fwknop [reported by author]	X.Org 1 2
exifprobe 1	jhead [?]	capnproto 1
Xerces-C 1 2 3	metacam 1	djvulibre 1
exiv 1 2	Linux btrfs 1 2 3 4 6 7 8	Knot DNS 1
curl 1 2 3	wpa_supplicant 1	libde265 [reported by author]
dnsmasq 1	libbpg (1)	lame 1 2 3 4 5 6
libwmf 1	uudecode 1	MuPDF 1 2 3 4
imlib2 1 2 3 4	libraw 1	libbson 1
libsass 1	yara 1 2 3 4	W3C tidy-html5 1
VLC 1 2	FreeBSD syscons 1 2 3	John the Ripper 1 2
screen 1 2 3	tmux 1 2	mosh 1
UPX 1	indent 1	openjpeg 1 2
MMIX 1	OpenMPT 1 2	rxvt 1 2
dhcpcd 1	Mozilla NSS 1	Nettle 1
mbed TLS 1	Linux netlink 1	Linux ext4 1
Linux xfs 1	botan 1	expat 1 2
Adobe Reader 1	libav 1	libical 1

The new generation

Genealogy (Manes et al. 2019)



Driller (2016)

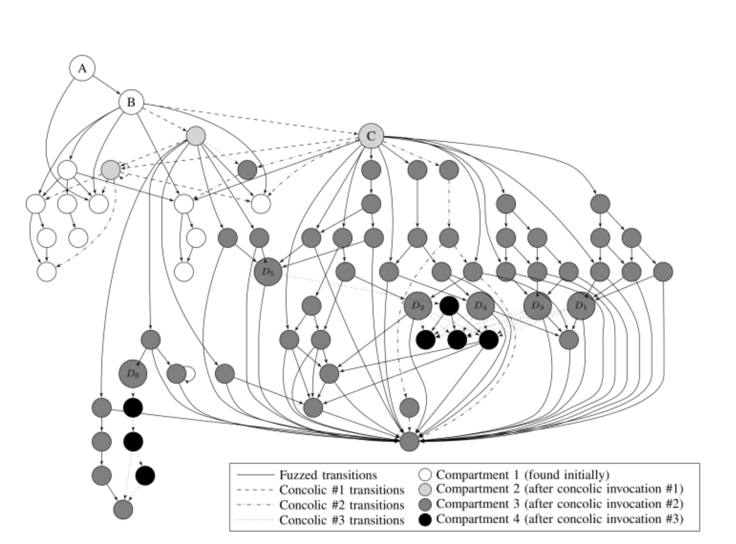
Goal

Combination of SE & greybox fuzzing

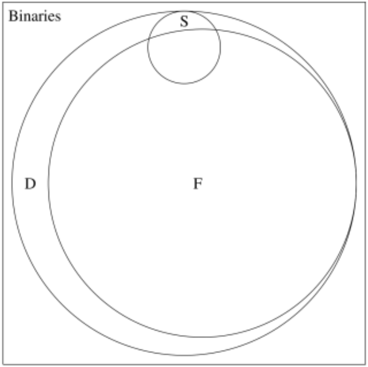
Key insight

When the fuzzer finds "hard conditions", launch SE to solve them, then let the fuzzer continue with this new input.

Driller : Inner workings



Driller : Results



Method	Crashes Found
Fuzzing	68
Fuzzing ∩ Driller	68
Fuzzing ∩ Symbolic	13
Symbolic	16
Symbolic ∩ Driller	16
Driller	77

Vuzzer (2017) i

Proposal

Configurations are added only upon discovering a new non-error handling block (statically determined).

The configuration fitness is the weighted sum of the log of the frequency over exercised blocks.

Key insight

Error-handling blocks lower the chance of vulnerabiliies.

Consequence

VUzzer prefers normal blocks that are rare according to CFG random walks.

Vuzzer (2017) ii

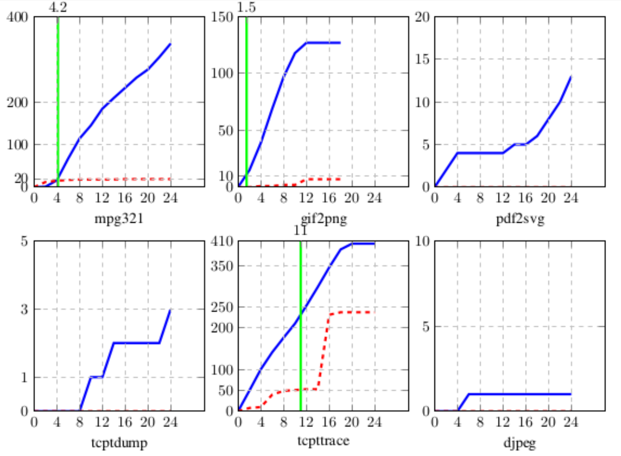
Table 1: Vuzzer vs AFL: #crashes

Software	Vuzzer	AFL
mpg321	337	19
gif2png/libpng	127	7
pdf2svg/libpoppler	13	0
tcdpump/libpcap	3	0
tcptrace/libpcap	403	238
djpeg/libjpeg	1	0

Vuzzer (2017) iii

Table 2: Vuzzer vs AFL: #inputs

Software	Vuzzer	AFL
mpg321	23.6k	883k
gif2png/libpng	43.2k	1.84m
pdf2svg/libpoppler	5k	923k
tcdpump/libpcap	77.8k	2.89m
tcptrace/libpcap	40k	3.29m
djpeg/libjpeg	90k	35.9m


4.2
400
200
100
20
0
0 4 8 12 16 20 24
mpg321
1.5
150
100
50
10
0
0 4 8 12 16 20 24
gif2png
20
15
10
5
0
0 4 8 12 16 20 24
pdf2svg
5
3
1
0
0 4 8 12 16 20 24
tcptdump
11
410
350
250
200
100
50
0
0 4 8 12 16 20 24
tcpttrace
10
5
0
0 4 8 12 16 20 24
djpeg

Vuzzer (2017) v

Table 3: New bugs discovered

Program	Bug type	Fixed ?	Reported
tcpdump	Oob read	✓	✗
mpg321	Oob read	✗	✓
mpg321	Double free	✗	✓
pdf2svg	Null ptr deref	✓	✗
pdf2svg	Abort	✓	✗
pdf2svg	Assert failure	✓	✗
tcptrace	Oob read	✗	✓
gif2png	Oob read	✗	✓

AFLGo (2017) i

Goal

Generating inputs with the objective of reaching a set of program locations.

Key idea : Directed Greybox Fuzzing

Applications

- Patch testing
- Crash reproduction
- Static analysis report verification
- Information flow detection

AFLGo (2017) ii

Measure

- Distance between a function & a set of target functions = harmonic mean

  Harmonic mean can distinguish between a node that is close to one target and further from another and one that is equidistant from both (average mean may be equal).

Scheduling

Key insight :: simulated annealing

Use more energy to fuzz seeds closer to the targets.

Enter exploitation after given exploration time has elapsed.

FairFuzz (2018) i

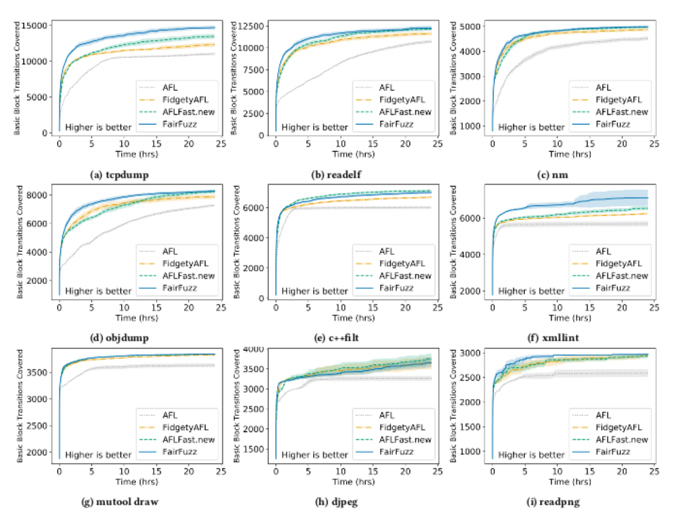
Goal

- Achieve better branch coverage for AFL
- No extra instrumentation

Key steps

1. Identify *rare* branches
2. New mutation technique to increase probability of hitting rare branches.

FairFuzz (2018) ii



Angora (2018)

Angora (2018) i

Goal

Increase branch coverage by solving path constraints *without symbolic execution*

Key ingredients

- Context-sensitive branch coverage
- Byte-level taint tracking
- Gradient descent-based search
- Type & shape inference

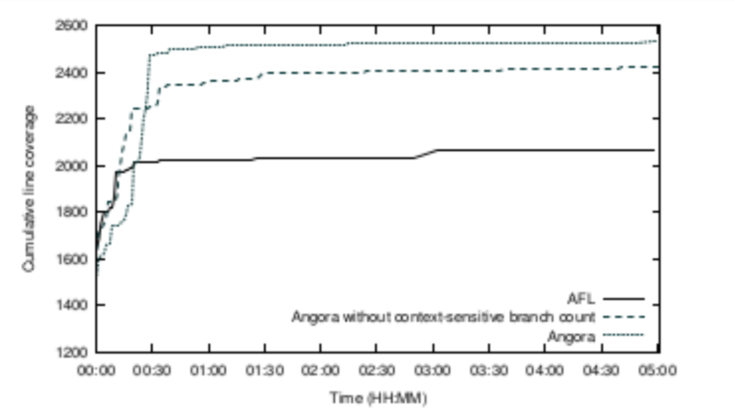
Angora (2018) ii

Table 4: Results

Program	Listed bugs	Angora	AFL	Vuzzer	Steelix
uniq	28	29	9	27	7
base64	44	48	0	17	43
md5sum	57	57	1	✖	28
who	2136	1541	1	50	194

Omitted from table : SES, FUZZER

Angora (2018) iii



Angora (2018) iv

Table 5: Angora vs AFL line coverage

Program	AFL	Angora
file-5.32	2070	2534
jhead-3.00	347	789
xmlwf(expat)-2.2.5	1980	2025
djpeg(ijg)-v9b	5401	5509
readpng(libpng)-1.6.34	1592	1799
nm-2.29	6372	7721
objdump-2.29	3448	6216
size-2.29	2839	4832

Angora (2018) v

Table 6: Angora vs AFL branch coverage

Program	AFL	Angora
file-5.32	1462	1899
jhead-3.00	218	789
xmlwf(expat)-2.2.5	2905	3158
djpeg(ijg)-v9b	1677	1782
readpng(libpng)-1.6.34	872	1007
nm-2.29	4105	4693
objdump-2.29	2071	3393
size-2.29	1792	2727

Angora (2018) vi

Table 7: Angora vs AFL unique crashes

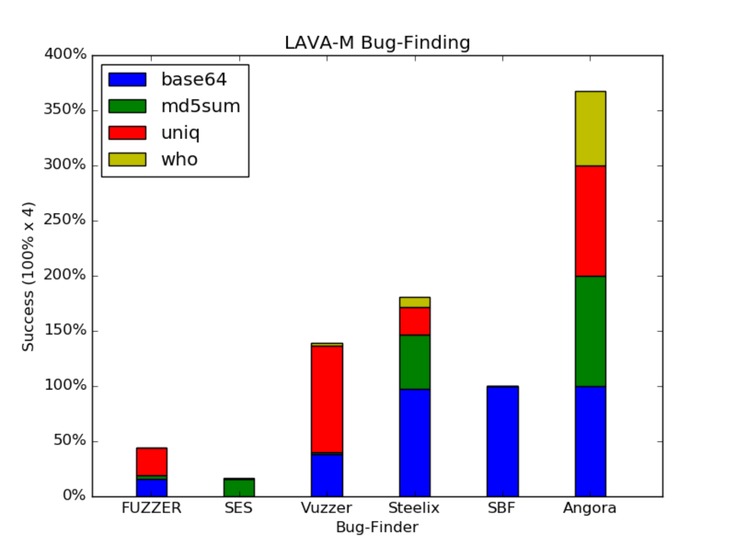
Program	AFL	Angora
file-5.32	0	6
jhead-3.00	19	52
xmlwf(expat)-2.2.5	0	0
djpeg(ijg)-v9b	0	0
readpng(libpng)-1.6.34	0	0
nm-2.29	12	29
objdump-2.29	4	48
size-2.29	6	48

Achievements summary

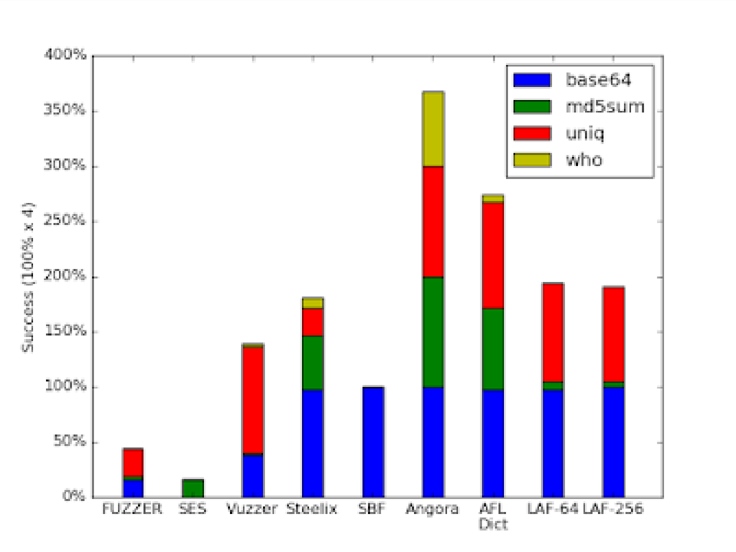
Ideas have been explored

- Better branch coverage (deeper, broader)
- Directed targeting
- Lightweight dynamic constraint solving
- Combination with other analyses:
    - Symbolic execution
    - Static analyses

Standard benchmarks: an emerging concern

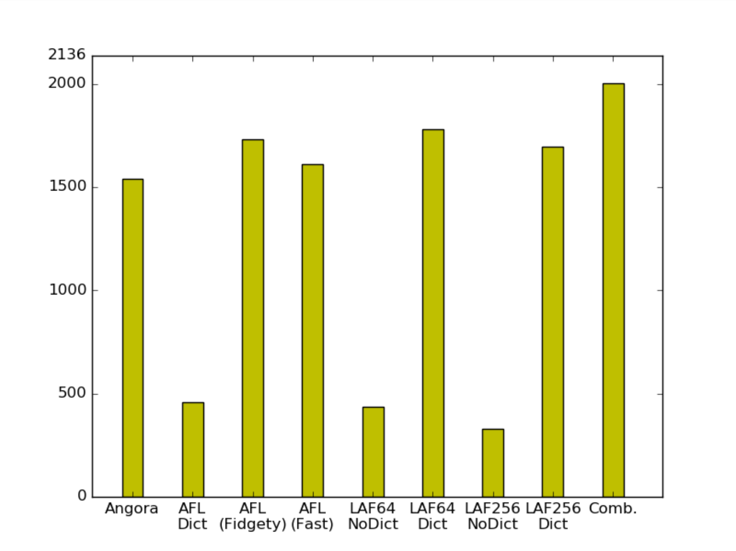


Baseline



Success (100% x 4)
400%
350%
300%
250%
200%
150%
100%
50%
0%
FUZZER
SES
Vuzzer
Steelix
SBF
Angora
AFL Dict
LAF-64
LAF-256
base64
md5sum
uniq
who

Who's who


2136
2000
1500
1000
500
0
Angora
AFL Dict
AFL (Fidgety)
AFL (Fast)
LAF64 NoDict
LAF64 Dict
LAF256 NoDict
LAF256 Dict
Comb.

More goodness to come . . .

Fuzzing is a very active research area

Check `https://wcventure.github.io/FuzzingPaper/`

New developments in 2019-2020

- Hawkeye (CCS '18)
- DigFuzz (NDSS '19)
- MemFuzz (ICSE '19)
- Eclipser (ICSE '19)
- Matryoshka (CCS '19)
- SAVIOR (S&P '20)
- . . .

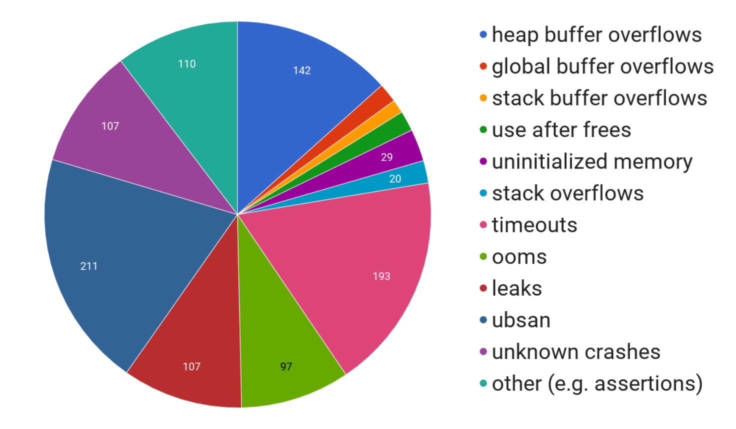
FAAS

Recent years have seen initiatives from "MAANG" members

Google OSS-Fuzz

Microsoft Project Springfield

Types of bugs (OSS-Fuzz 2016)



Anti-fuzzing (Antifuzz, 2019) i

Attacking fuzzer's assumptions

Fuzzers depends on some implicit support:

- Coverage feedback
- Crash detection
- Speedy loop
- (sometimes) Solvable constraints

Coverage feedback countermeasure

Add fake function calls depending on input. This add many new paths, all of them "interesting"

Anti-fuzzing (Antifuzz, 2019) ii

Crash detection countermeasure

Use common anti-debugging technique.

Fake crash catching

Execution speed countermeasure

Check validity of inputs : this induces a slowdown that is enough to delay fuzzers.

Thwart constraint solving

Instead of checking conditionals against a value `v`, check against `sha256(v)`.

Evaluating anti-fuzzing

Fuzzers do not find bugs in LAVA-M anymore

Code coverage is reduced by $\geq 90\%$

Performance overhead is negligible

Questions ?

https://rbonichon.github.io/teaching/2021/asi36/