Week 8: Lecture A

Fuzzing Science

Monday, February 26, 2024

Recap: Key Dates

- **Feb. 26** Sign up final project team
- **Feb. 28** Lab 3 due
- **Feb. 28** 5-minute project proposals
- **Mar. 04 & 06** No class (Spring Break)
- **Apr. 17 & 22** Final project presentations

cs.utah.edu/~snagy/courses/cs5963/schedule

Feb. 12 **Harnessing I** (slides) ▶ Readings: **Harnessing Lab released**	Feb. 14 **Harnessing II** (slides) ▶ Readings: **Final Project released** **Triage Lab due by 11:59pm**
Feb. 19 **No Class (President's Day)**	Feb. 21 **Tackling Roadblocks** ▶ Readings:
Feb. 26 **Fuzzing Science** ▶ Readings: **Sign up your final project team by 11:59pm**	Feb. 28 **Proposal Presentations** **Harnessing Lab due by 11:59pm**
Mar. 04 **No Class (Spring Break)**	Mar. 06 **No Class (Spring Break)**

Recap: Lab 3 Overview

- **Assignment:** write your own **AFL-friendly** harness for libArchive
  - Read its documentation in: https://linux.die.net/man/3/libarchive
  - https://github.com/google/oss-fuzz/blob/master/projects/libarchive/libarchive_fuzzer.cc
- **Create a harness that reads data from files**
  - What functions did you try?
  - What worked and what didn't?
- **Deliverable:** a **1–3 page report** detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)
  - Submit your harness code in your report
  - **Free to team up (max 3 students per group)**
  - **Submit one report per group**
- **Linux environments are recommended**
  - Use a VM if you don't have one!

Recap: Lab 3 Tips

- **Read libArchive's documentation and get inspiration from others' code**
  - Understand the libArchive manpages
  - Look at how others (e.g., non-fuzzing projects) use its API
- **Validate your results**
  - Measure code coverage of the libArchive codebase
  - Look for increasing code coverage over time
- **Deadline:** Wednesday, February 28th by 11:59PM
  - Group assignment (**up to 3 members**)
  - Look for teammates in-class and on Piazza
  - See cs.utah.edu/~snagy/courses/cs5963/assignments.html

Recap: Tackling Harnessing Roadblocks

- **No increase in coverage...**
  - AFL's "`new edges on`" counter stays stagnant
  - Are you sure that you instrumented **the library**?
  - If not, you will only get coverage of **the harness**!

Recap: Tackling Harnessing Roadblocks

- **No increase in coverage...**
  - AFL's "`new edges on`" counter stays stagnant
  - Are you sure that you instrumented **the library**?
  - If not, you will only get coverage of **the harness**!
  - Trouble compiling / linking? Can just use **QEMU**!

10) Notes on linking

The feature is supported only on Linux. Supporting BSD may amount to porting the changes made to linux-user/elfload.c and applying them to bsd-user/elfload.c, but I have not looked into this yet.

The instrumentation follows only the .text section of the first ELF binary encountered in the linking process. It does not trace shared libraries. In practice, this means two things:

- Any libraries you want to analyze *must* be linked statically into the executed ELF file (this will usually be the case for closed-source apps).
- Standard C libraries and other stuff that is wasteful to instrument should be linked dynamically - otherwise, AFL++ will have no way to avoid peeking into them.

Setting `AFL_INST_LIBS=1` can be used to circumvent the .text detection logic and instrument every basic block encountered.

Recap: Tackling Harnessing Roadblocks

- **No increase in coverage...**
  - AFL's "new edges on" counter stays stagnant
  - Are you sure that you instrumented **the library**?
  - If not, you will only get coverage of **the harness**!
  - Trouble compiling / linking? Can just use **QEMU**!

- **New coverage, but zero crashes...**
  - Is your harness calling **interesting functionality**?
  - If so, can you verify that it is **calling it correctly**?
  - Are you fuzzing for a **long enough time**?

Recap: Tackling Harnessing Roadblocks

- **No increase in coverage...**
  - AFL's "new edges on" counter stays stagnant
  - Are you sure that you instrumented **the library**?
  - If not, you will only get coverage of **the harness**!
  - Trouble compiling / linking? Can just use **QEMU**!

- **New coverage, but zero crashes...**
  - Is your harness calling **interesting functionality**?
  - If so, can you verify that it is **calling it correctly**?
  - Are you fuzzing for a **long enough time**?
  - You can try **older API versions** with known bugs!

Libarchive downloads

```
sha256sums
libarchive-v3.7.2-amd64.zip.asc
libarchive-v3.7.2-amd64.zip
libarchive-v3.7.1-amd64.zip.zip.asc
libarchive-v3.7.1-amd64.zip.zip
libarchive-v3.7.0-amd64.zip.asc
libarchive-v3.7.0-amd64.zip
libarchive-v3.6.2-amd64.zip.asc
libarchive-v3.6.2-amd64.zip
libarchive-v3.6.1-amd64.zip.asc
libarchive-v3.6.1-amd64.zip
libarchive-v3.6.0-win64.zip.asc
libarchive-v3.6.0-win64.zip
libarchive-v3.5.3-win64.zip.asc
libarchive-v3.5.3-win64.zip
libarchive-v3.5.2-win64.zip.asc
libarchive-v3.5.2-win64.zip
libarchive-v3.5.1-win64.zip.asc
libarchive-v3.5.1-win64.zip
libarchive-v3.5.0-win64.zip.asc
libarchive-v3.5.0-win64.zip
libarchive-v3.4.3-win64.zip.asc
libarchive-v3.4.3-win64.zip
libarchive-3.7.2.zip.asc
libarchive-3.7.2.tar.xz.asc
libarchive-3.7.2.tar.xz
libarchive-3.7.2.tar.gz.asc
```

Recap: Tackling Harnessing Roadblocks

- **No increase in coverage...**
  - AFL's `"new edges on"` counter stays stagnant
  - Are you sure that you instrumented **the library**?
  - If not, you will only get coverage of **the harness**!
  - Trouble compiling / linking? Can just use **QEMU**!
- **New coverage, but zero crashes...**
  - Is your harness calling **interesting functionality**?
  - If so, can you verify that it is **calling it correctly**?
  - Are you fuzzing for a **long enough time**?
  - You can try **older API versions** with known bugs!
- **Lots crashes in very little time...**
  - Are they reproducible with any **available oracles**?
  - Re-run input with `bsdtar` application and check!

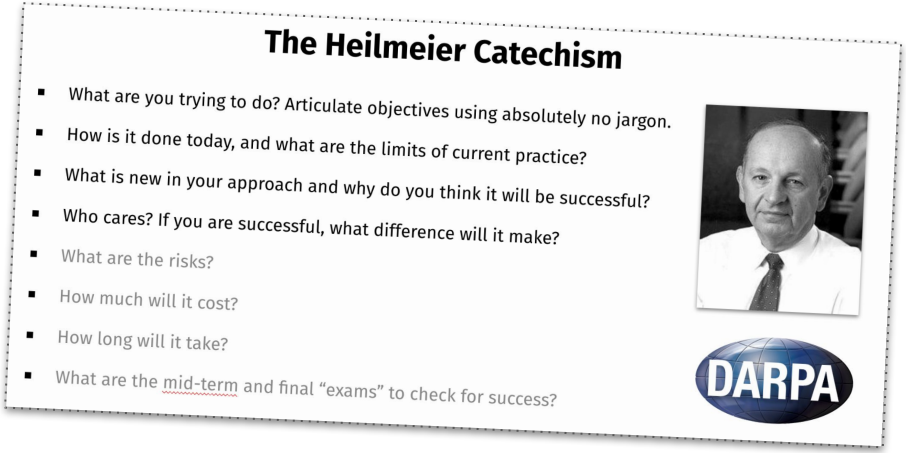
Recap: Tackling Harnessing Roadblocks

- **No increase in coverage...**
  - AFL's "new edges on" counter stays stagnant
  - Are you sure that you instrumented **the library**?
  - If not, you will only get coverage of **the harness**!
  - Trouble compiling / linking? Can just use **QEMU**!
- **New coverage, but zero crashes...**
  - Is your harness calling **interesting functionality**?
  - If so, can you verify that it is **calling it correctly**?
  - Are you fuzzing for a **long enough time**?
  - You can try **older API versions** with known bugs!
- **Lots crashes in very little time...**
  - Are they reproducible with any **available oracles**?
  - Re-run input with `bsdtar` application and check!
  - **Not a silver bullet**—may cover different functions!

Trial-and-error harness refinement!

Recap: Project Schedule

- **Monday, Feb 26th:** team signup due
- **Wednesday, Feb. 28th:** proposal day
  - **Instructions:** a **5-minute** presentation that motivates your project
  - **Goal:** practice the art of “the pitch”
    - Get feedback from your peers
    - Follow **Heilmeier’s Catechism!**
- **Mar. 27th:** in-class project workday
- **Apr. 17th & 22nd:** final presentations
  - 15–20 minute slide deck and discussion
  - What you did, and why, and what results



Recap: Project Team Signup

- **Signup sheet** available on course website (must use **UofU gcloud** account)
  - Fill-in your **project title** and **teammate names** by **11:59PM on Monday, February 26th**

KAHLERT SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

Syllabus Schedule Assignments Piazza Canvas Paper Sign... Project Signup

CS 5963/6963: Applied Software Security Testing

This special topics course will dive into today's state-of-the-art techniques for uncovering hidden security vulnerabilities in software. Introductory fuzzing exercises will provide hands-on experience with industry-popular security tools such as AFL+ and AddressSanitizer, culminating in a final project where **you'll work to hunt down, analyze, and report security bugs in a real-world application or system of your choice**.

This class is open to graduate students and upper-level undergraduates. It is recommended you have a solid grasp over topics like software security, systems programming, and C/C++.

Learning Outcomes: At the end of the course, students will be able to:

- Design, implement, and deploy automated testing techniques to improve vulnerability on large and complex software systems.
- Assess the effectiveness of automated testing techniques and identify why they are well- or ill-suited to specific codebases.
- Distill testing outcomes into actionable remediation information for developers.
- Identify opportunities to adapt automated testing to emerging and/or unconventional classes of software or systems.
- Pinpoint testing obstacles and synthesize strategies to overcome them.
- Appreciate that testing underpins modern software quality assurance by discussing the advantages of proactive and post-deployment software testing efforts.

📌 **Directions:** fill-in your final project **teammate names**, and a **brief title** of your project

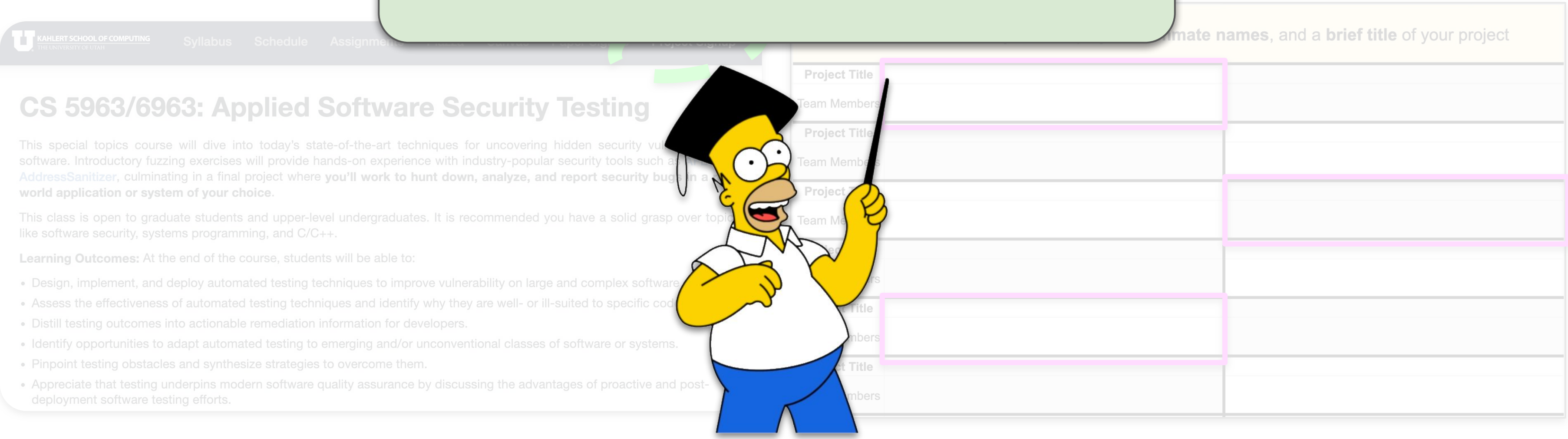
Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		

Recap: Project Team Signup

- **Signup sheet** available on course website (must use **UofU gcloud** account)
  - Fill-in your pr… y, February 26th

Need help finalizing your project idea?

Come chat with me in office hours!



Questions?

Fuzzing Science

Why evaluate fuzzers?

- **Advance science**
  - “I must publish to graduate”
- **Validate your technique**
  - “My fix really does work!”
- **Convince others your fuzzer is best**
  - “I made the best fuzzer for Microswat Superclick!”

How should fuzzers be evaluated?

- Pick a few **benchmarks**
- **Compare** against AFL
- Run a few **trials**
- Compute **average coverage**

How should fuzzers be evaluated?

- Pick a few **benchmarks**
- **Compare** against AFL
- Run a few **trials**
- Compute **average coverage**

WRONG

Fuzzer evaluations must be *scientific*

Evaluating Fuzz Testing

George Klees, Andrew Ruef, Benji Cooper
University of Maryland

Shiyi Wei
University of Texas at Dallas

Michael Hicks
University of Maryland

ABSTRACT

Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We conclude with some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust.

Why do we think fuzzers work? While inspiration for new ideas may be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally. When a researcher develops a new fuzzer algorithm (call it A), they must empirically demonstrate that it provides an advantage over the status quo. To do this, they must choose:

- a compelling *baseline* fuzzer B to compare against;
- a sample of target programs—the *benchmark suite*;
- a *performance metric* to measure when A and B are run on the benchmark suite; ideally, this is the number of (possibly exploitable) bugs identified by crashing inputs;
- a meaningful set of *configuration parameters*, e.g., the *seed file* (or files) to start fuzzing with, and the *timeout* (i.e., the duration) of a fuzzing run.

An evaluation should also account for the fundamentally random

Fuzzer evaluations must be *scientific*

Evaluating Fuzz Testing

George Klees, Andrew Ruef, Benji Cooper
University of Maryland

Shiyi Wei
University of Texas at Dallas

Michael Hicks
University of Maryland

ABSTRACT

Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We conclude with some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust.

Why do we think fuzzers work? While inspiration for new i
may be drawn from mathematical analysis, fuzzers are prima
evaluated experimentally. When a researcher develops a new fu
algorithm (call it A), they must empirically demonstrate tha
provides an advantage over the status quo. To do this, they m
choose:

- a compelling *baseline* fuzzer B to compare against;
- a sample of target programs—the *benchmark suite*;
- a *performance metric* to measure when A and B are ru the benchmark suite; ideally, this is the number of (poss exploitable) bugs identified by crashing inputs;
- a meaningful set of *configuration parameters*, e.g., the *file* (or files) to start fuzzing with, and the *timeout* (i.e., duration) of a fuzzing run.

An evaluation should also account for the fundamentally rand

Hicks Wins NSA's Best Scientific Cybersecurity Paper Award

NSA · cybersecurity · faculty · research · awards

Published October 2, 2019

Michael Hicks, a professor of computer science, helped lead a team of researchers to victory in the National Security Agency's (NSA) 7th Annual Best Scientific Cybersecurity Paper Competition.

Benchmark Selection

Benchmark Selection

- **Size matters**
  - File size
    - Megabytes
  - Complexity
    - Basic blocks
    - Proxy for # of paths

Benchmark Selection

- **Size matters**
  - File size
    - Megabytes
  - Complexity
    - Basic blocks
    - Proxy for # of paths

Does execution mechanism speed always matter?

- Profile average time spent on **target program** vs. **execution mechanism**

Avg. Target Time / input	Avg. Execution Time / input	Prop. spent on Execution
2 ms	1—10 ms	33.3—83.3%
300 ms	1—10 ms	0.0—3.2%

  - **Short-running** test cases = execution speed matters **more**
  - **Long-running** test cases = execution matters **less** (and **coverage tracing** matters more)
- As usual, this phenomena is **target-dependent**

Benchmark Selection

- **Size matters**
  - File size
    - Megabytes
  - Complexity
    - Basic blocks
    - Proxy for # of paths

- **Results tell all**
  - Ideally good on all sizes

Does execution mechanism speed always matter?

- Profile average time spent on **target program** vs. **execution mechanism**

Avg. Target Time / input	Avg. Execution Time / input	Prop. spent on Execution
2 ms	1—10 ms	33.3—83.3%
300 ms	1—10 ms	0.0—3.2%

  - **Short-running** test cases = execution speed matters **more**
  - **Long-running** test cases = execution matters **less** (and **coverage tracing** matters more)
- As usual, this phenomena is **target-dependent**

Benchmark Selection

- **Maximize variety**
  - Program type
    - Image parser
    - Document reader
    - Audio file converter
  - Program input format
    - JPEG, GIF, EXIF
    - PDF, DOC, XML
    - MP3, WAV, OGG
  - Parent library / application
    - ImageMagick
    - Binutils
    - RARLab

Benchmark Selection

- **Cardinal sins of benchmark selection**
  - Fuzzing programs of a single type, format
    - E.g., PDF parsers
  - Fuzzing programs from a single package
    - E.g., Binutils, Coreutils
    - Happens far too often

- **Results should be generalizable**
  - If not, then explain why
  - If not justified, then **reject**

- GNU Binutil Command Examples
  - readelf
  - strings
  - nm
  - ar
  - objdump
  - strip
  - objcopy
  - addr2line
  - size

Other Benchmark Selection Sins

- Developing a **binary-only** approach
  - **But only evaluating open-source programs**
  - Finding closed-source benchmarks is hard!

Other Benchmark Selection Sins

- Developing a **binary-only** approach
  - **But only evaluating open-source programs**
  - Finding closed-source benchmarks is hard!

Application	OS	Binary	Size	Blocks	P*C	Sym	Opt
B1FreeArchiver	L	b1	4.1M	150,138	D	✔	✔
B1FreeArchiver	L	b1manager	19.3M	290,628	D	✔	✔
BinaryNinja	L	binaryninja	34.4M	998,630	D	✔	✔
BurnInTest	L	bit_cmd_line	2.6M	73,229	D	✗	✔
BurnInTest	L	bit_gui	3.4M	107,897	D	✗	✔
Coherent PDF	L	smpdf	3.9M	61,204	D	✔	✔
IDA Free	L	ida64	4.5M	173,551	I	✗	✔
IDA Pro	L	idat64	1.8M	82,869	I	✗	✔
LzTurbo	L	lzturbo	314K	13,361	D	✗	✔
NConvert	L	nconvert	2.6M	111,652	D	✗	✔
NVIDIA CUDA	L	nvdisasm	19M	46,190	D	✗	✔
Object2VR	L	object2vr	8.1M	239,089	D	✔	✔
PNGOUT	L	pngout	89K	4,017	D	✗	✔
RARLab	L	rar	566K	25,287	D	✗	✔
RARLab	L	unrar	311K	13,384	D	✗	✔
RealVNC	L	VNC-Viewer	7.9M	338,581	D	✗	✔
VivaDesigner	L	VivaDesigner	28.9M	1,097,993	D	✗	✔
VueScan	L	vuescan	15.4M	396,555	D	✗	✔
Everything	W	Everything	2.2M	115,980	D	✔	✗
Imagine	W	Imagine64	15K	99	D	✗	✗
NirSoft	W	AppNetworkCounter	122K	4,091	D	✗	✗
OcenAudio	W	ocenaudio	6.1M	178,339	D	✗	✗
USBDView	W	USBDeview	185K	7,367	D	✗	✗

Other Benchmark Selection Sins

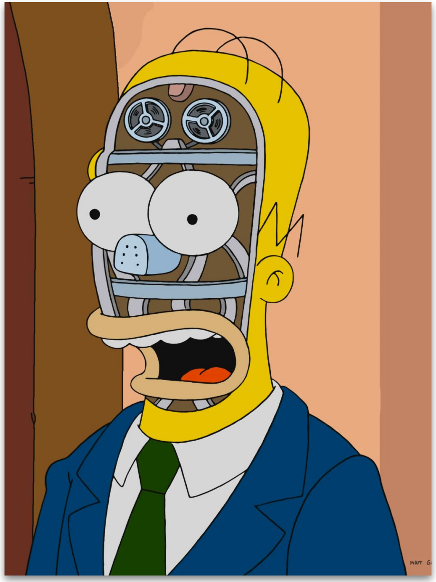
- Developing a **binary-only** approach
  - **But only evaluating open-source programs**
  - Finding closed-source benchmarks is hard!

Application	OS	Binary	Size	Blocks	P*C	Sym	Opt
B1FreeArchiver	L	b1	4.1M	150,138	D	✔	✔
B1FreeArchiver	L	b1manager	19.3M	290,628	D	✔	✔
BinaryNinja	L	binaryninja	34.4M	998,630	D	✔	✔
BurnInTest	L	bit_cmd_line	2.6M	73,229	D	✗	✔
BurnInTest	L	bit_gui	3.4M	107,897	D	✗	✔
Coherent PDF	L	smpdf	3.9M	61,204	D	✔	✔
IDA Free	L	ida64	4.5M	173,551	I	✗	✔
IDA Pro	L	idat64	1.8M	82,869	I	✗	✔
LzTurbo	L	lzturbo	314K	13,361	D	✗	✔
NConvert	L	nconvert	2.6M	111,652	D	✗	✔
NVIDIA CUDA	L	nvdisasm	19M	46,190	D	✗	✔
Object2VR	L	object2vr	8.1M	239,089	D	✔	✔
PNGOUT	L	pngout	89K	4,017	D	✗	✔
RARLab	L	rar	566K	25,287	D	✗	✔
RARLab	L	unrar	311K	13,384	D	✗	✔
RealVNC	L	VNC-Viewer	7.9M	338,581	D	✗	✔
VivaDesigner	L	VivaDesigner	28.9M	1,097,993	D	✗	✔
VueScan	L	vuescan	15.4M	396,555	D	✗	✔
Everything	W	Everything	2.2M	115,980	D	✔	✗
Imagine	W	Imagine64	15K	99	D	✗	✗
NirSoft	W	AppNetworkCounter	122K	4,091	D	✗	✗
OcenAudio	W	ocenaudio	6.1M	178,339	D	✗	✗
USBDView	W	USBDeview	185K	7,367	D	✗	✗

Binary	ZAFL vs. AFL-Dyninst rel. crash	rel. total	rel. queue	ZAFL vs. AFL-QEMU rel. crash	rel. total	rel. queue
`idat64`	1.000	0.789	2.332	✗	1.657	1.192
`nconvert`	3.538	0.708	48.140	1.095	1.910	1.303
`nvdisasm`	1.111	0.757	1.484	1.111	0.578	1.252
`pngout`	1.476	5.842	1.380	1.476	3.419	1.023
`unrar`	✗	0.838	6.112	2.000	1.284	1.249
Mean Rel. Increase	**+55%**	**+16%**	**+326%**	**+38%**	**+52%**	**+20%**
Mean MWU Score	**0.036**	**0.041**	**0.009**	**0.082**	**0.021**	**0.045**

Other Benchmark Selection Sins

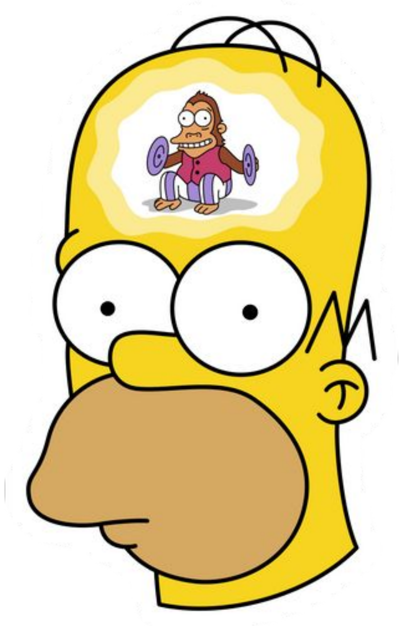
- Relying on **synthetic benchmark corpora**
  - E.g., SPEC2000, LAVA-M
  - **Often limited in their semantics**
    - LAVA-M: only magic-byte bugs
  - Many reviewers hate this
    - I am more forgiving
    - **Best served as a “preliminary” data point**



Competitor Selection

Choosing worthy competitors…

- **Many different fuzzers today**
  - Random fuzzing
  - Grammar fuzzing
  - Token-level fuzzing
  - Rare branch targeting
  - Invariant-guided fuzzing
  - Sub-instruction profiling
  - …
  - **Which should you choose?**



Choose the state of the art!

- Pick the best **conventional** fuzzers
  - E.g., AFL, AFL++, libFuzzer
- Include the **latest and greatest** fuzzers
  - Are you building a better grammar fuzzer?
    - Compare to other grammar fuzzers!
    - E.g., Gramatron, Nautilus
  - Are you building a fast binary instrumenter?
    - Compare other binary instrumenters!
    - E.g., ZAFL, AFL-QEMU, AFL-Dyninst
  - **Up to you to stay up to date on the literature**

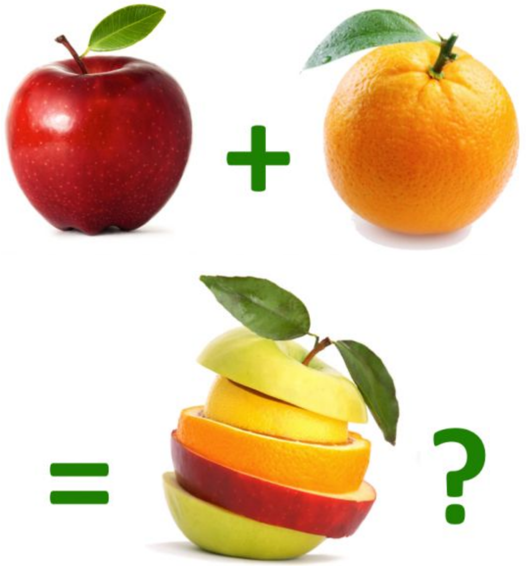
Implementation differences matter!

- **Build your fuzzer off a common platform**
  - AFL is today's most popular platform
    - Most fuzzers derived from AFL
  - Every change matters
    - E.g., speed, queue strategy, mutation
- **Leave core fuzzer design as a control**

Ablation Studies

- **Did you implement a ton of new features?**
  - Lots of levers to pull, knobs to twist
  - E.g., coverage granularity, execution timeout

- **Compare results with & without each**
  - Ablation studies make for better science
    - Is an idea the sum of its parts?
    - Or is one feature most critical?
  - **Better yet:** publish **one** key idea at a time

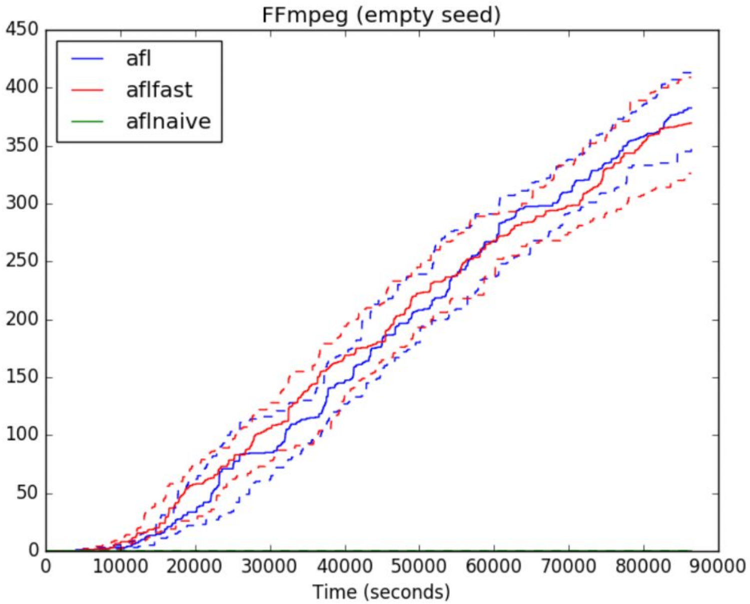


Cardinal sins of competitor selection...

- Choosing **old, obsolete fuzzers**
  - Contribution sold as better than it is
  - Automatic reject!
- Omitting **relevant state-of-the-art**
  - Usually a major revision
  - Reevaluate with what reviewers want
  - Reviewers need to know what to suggest
- **Throwing five things at the wall**
  - Many of these papers get accepted as-is
  - Bad science; we need ablation studies!
  - Paper must be carefully read and dissected

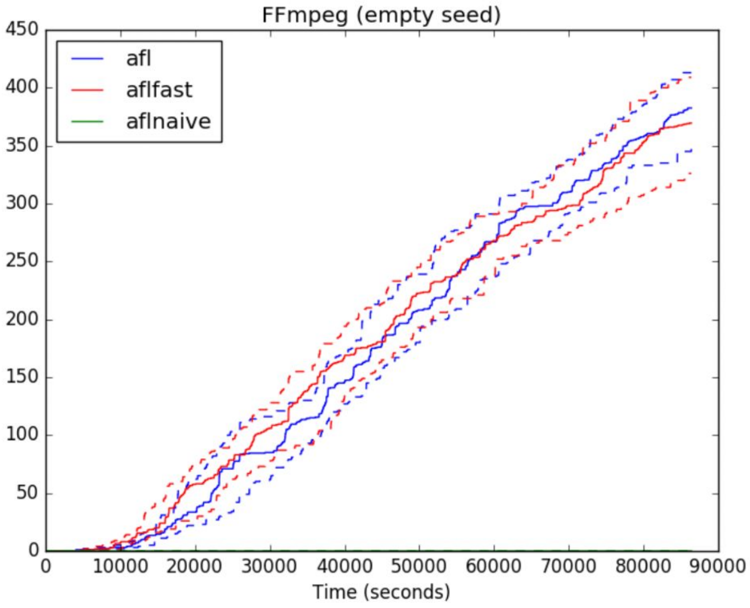
Experiment Setup

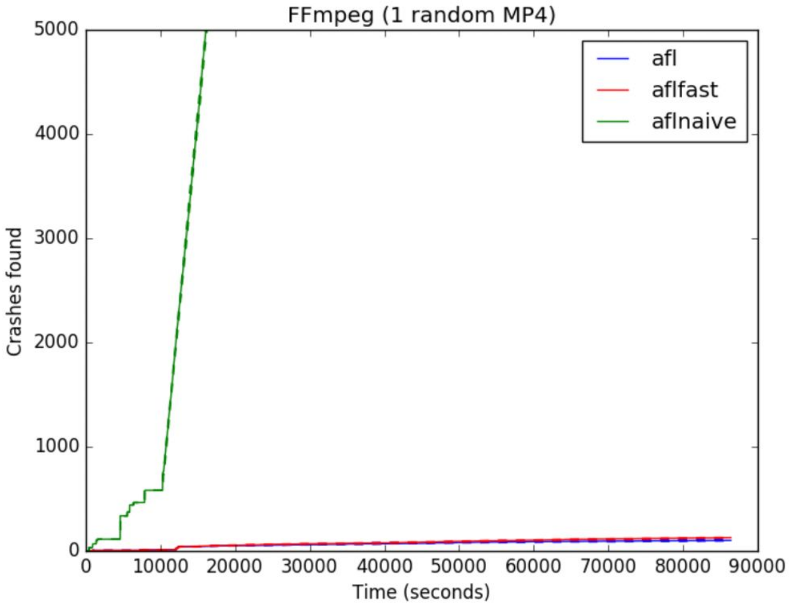
Seed Selection Matters



Source: Evaluating Fuzz Testing

Seed Selection Matters

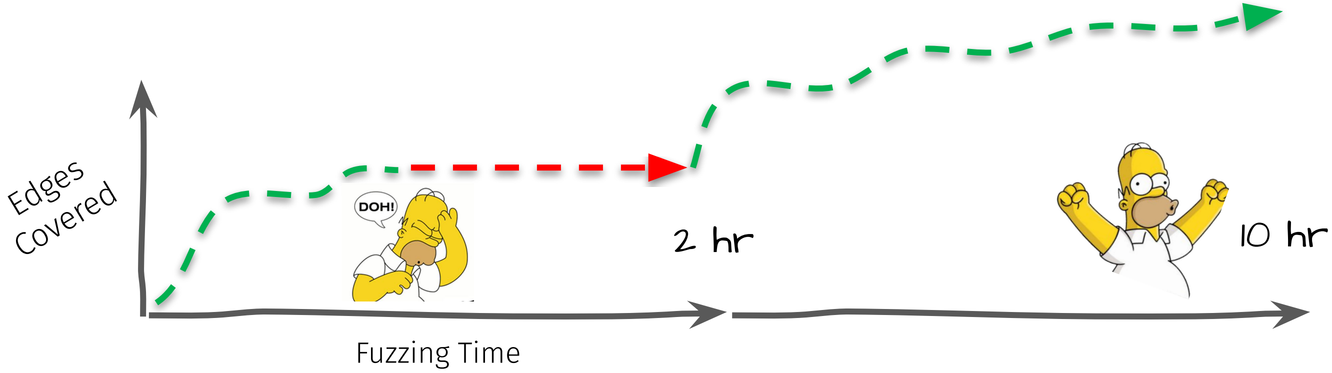




Source: Evaluating Fuzz Testing

Trial Duration

- **Early plateaus** can be misleading
  - Look for **sustained** plateaus
- Likewise, **high coverage early on** can be misleading
  - Want to see **sustained growth** over time



Trial Duration

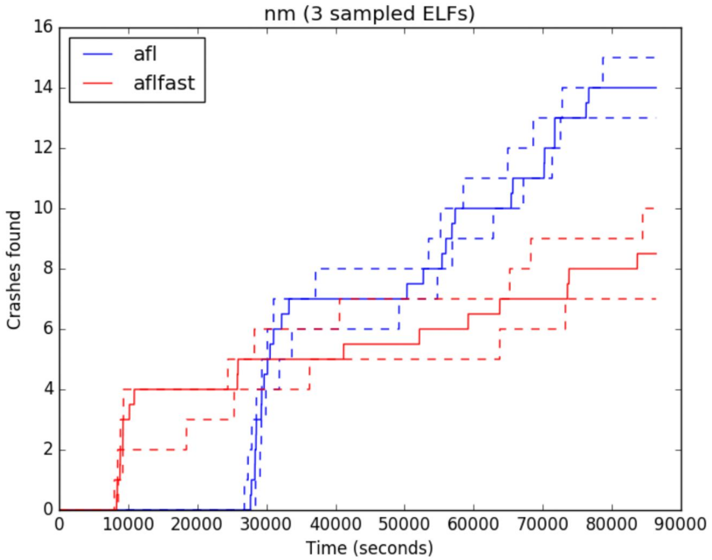


Figure 4: *nm* with three sampled seeds. At 6 hours: AFLFast is superior to AFL with $p < 10^{-13}$. At 24 hours: AFL is superior to AFLFast with $p = 0.000105$.

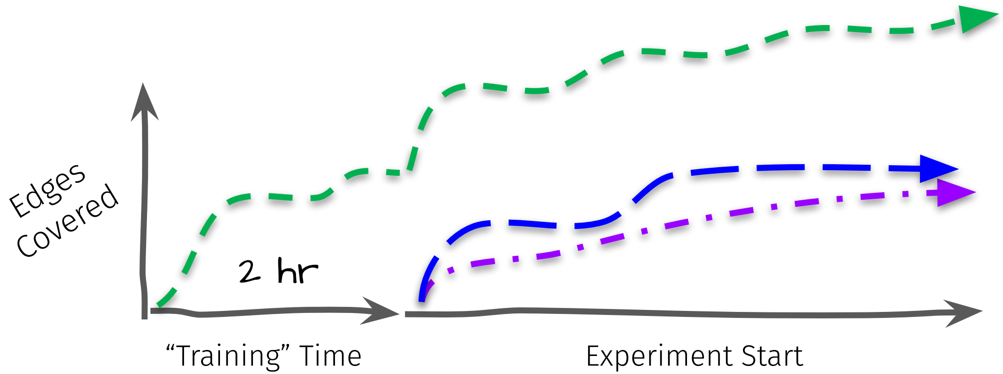
Source: Evaluating Fuzz Testing

Recommended Setup

- **Seeds of varying contents**
  - E.g., empty, well-formed, etc.
- **Trial length of 24+ hours**
  - The bare minimum
  - Longer is better
- **At least 5 trials per benchmark**
  - One trial is not representative

Ensuring Fairness

- **Maintain same setup across all fuzzers**
  - Same seeds, number of trials, duration, etc.
  - If a trial fails, **re-run until all 5 trials** completed
- **Begin fuzzers at same starting time**



Experiment Procedure

Results Processing

- **What metrics do we value most?**
  - **Code coverage**
    - Easy to measure
  - **Bugs and vulnerabilities** found
    - Hard to measure
  - **Zero-day vulnerabilities** found
    - A long time to produce
    - Bad reviewers ask for this

Common Vulnerabilities and Exposures

- **Project-specific metrics**
  - Results that prove a point or back up a claim
  - E.g., queue size, time spent on execution, etc.

Bugs and Vulnerabilities

- **Finding brand-new bugs is challenging**
  - Many common fuzzing targets are well-fuzzed
  - Looks bad to pick random, unknown programs
- **Synthetic bug benchmark corpora**
  - E.g., Magma, LAVA-M
    - Various caveats (e.g., realism)
- **Known bugs in older program versions**
  - E.g., fuzzing TCPDump 4.9.1

Identifier	Category	Binary
CVE-2011-4517	heap overflow	jasper
GitHub issue #58-1	stack overflow	mjs
GitHub issue #58-2	stack overflow	mjs
GitHub issue #58-3	stack overflow	mjs
GitHub issue #58-4	stack overflow	mjs
GitHub issue #136	stack overflow	mjs
Bugzilla #3392519	null pointer deref	nasm
CVE-2018-8881	heap overflow	nasm
CVE-2017-17814	use-after-free	nasm
CVE-2017-10686	use-after-free	nasm
Bugzilla #3392423	illegal address	nasm
CVE-2008-5824	heap overflow	sfconvert

Bug-finding Metrics

- **Number of bugs found**
  - Proxy for general bug-finding ability
  - Don't just report AFL's "unique crashes"—you must deduplicate them!
- **Time-to-exposure on known bugs**
  - Helpful—especially if your focus is on accelerating fuzzing speed

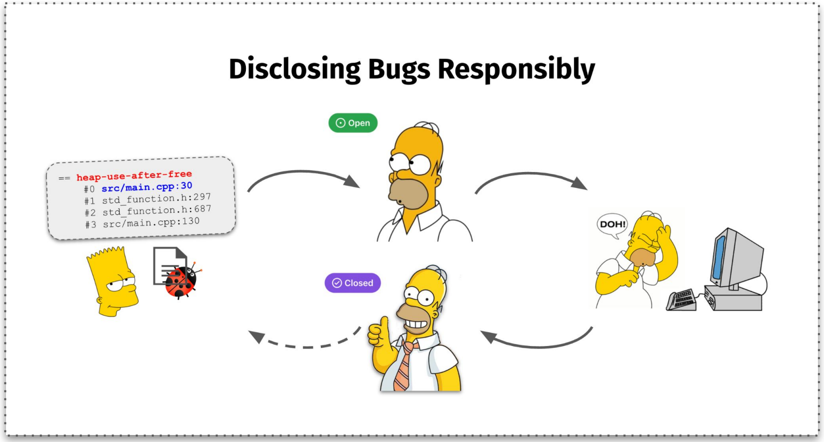
Error Type	Location	AFL-Dyninst	AFL-QEMU	ZAFL
heap overflow	`nconvert`	✗	18.3 hrs	12.7 hrs
stack overflow	`unrar`	✗	12.3 hrs	9.04 hrs
heap overflow	`pngout`	12.6 hrs	6.26 hrs	1.93 hrs
use-after-free	`pngout`	9.35 hrs	4.67 hrs	1.44 hrs
heap overread	`libida64.so`	23.7 hrs	✗	2.30 hrs
ZAFL Mean Rel. Decrease		**-660%**	**-113%**	

Table 7: Mean time-to-discovery of closed-source binary bugs found for AFL-Dyninst, AFL-QEMU, and ZAFL over 5×24-hour fuzzing trials. ✗ = bug is not reached in any trials for that instrumenter configuration.

Zero-day Vulnerabilities

- **Requires you to triage and report bugs**
  - You must fuzz the program's latest version
  - Follow responsible disclosure practices
  - Let **developer** request a CVE identifier
  - **See "Bugs & Triage II" lecture from class**

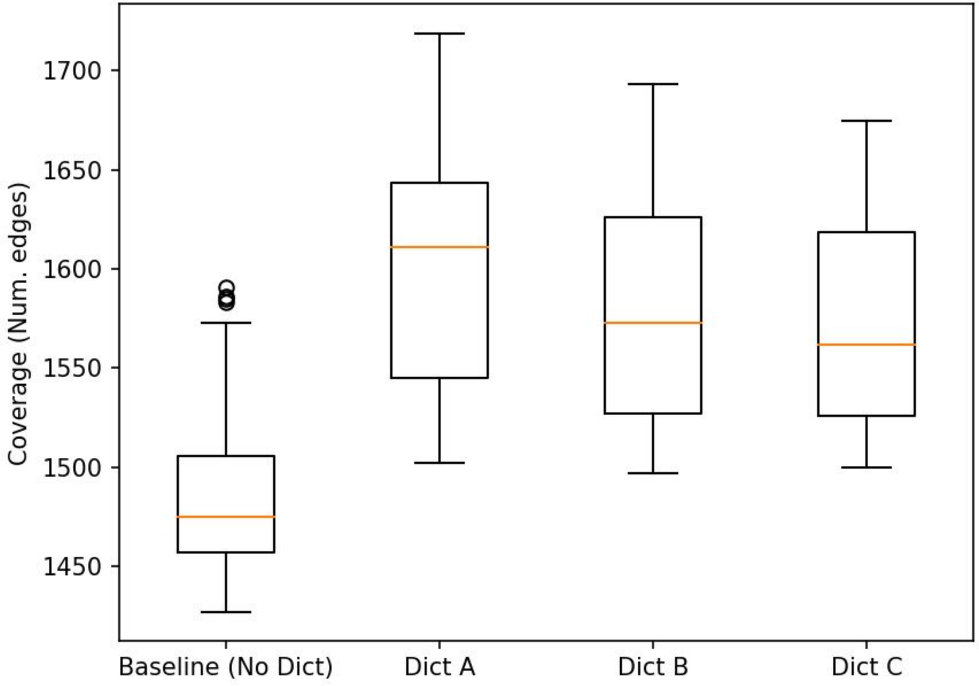
- **"You didn't find new bugs... REJECT!"**
  - A terrible trend in academic fuzzing
  - Happening less (from what I can tell)



Summary Statistics

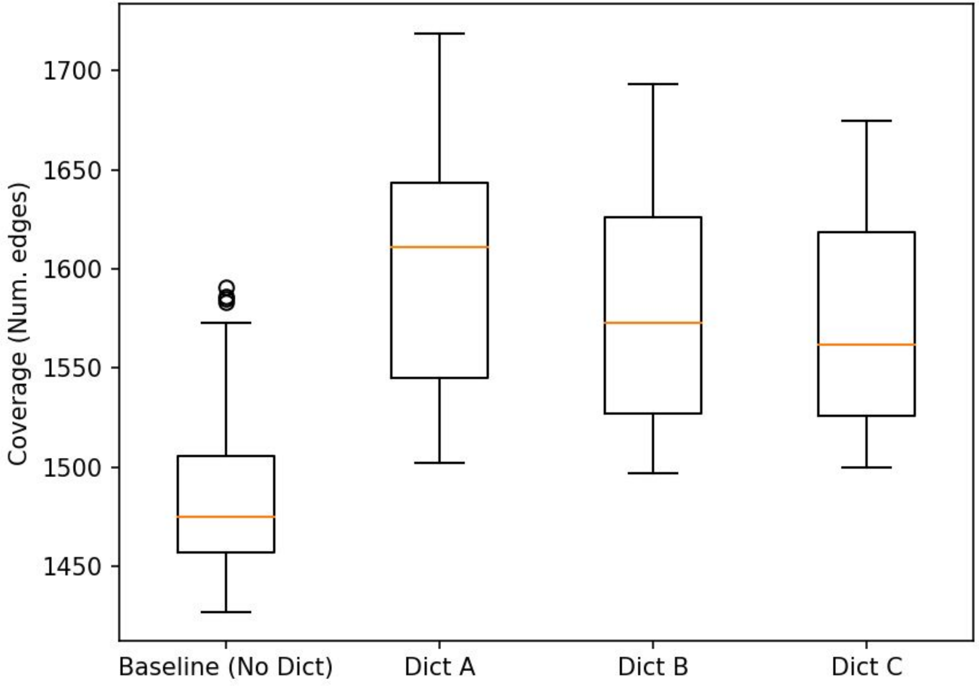
- **Are your results statistically significant?**
  - Arithmetic mean doesn't tell the story
    - Too coarse-grained of a comparison
- **The Mann-Whitney U test**
  - ***p*-value above 0.05** = not statistically significant
    - Your 2x improvement doesn't matter
  - ***p*-value less than 0.05** = statistically significant
    - Great job!
  - **The gold standard of fuzzing evaluations today**
  - Other: Vargha and Delaney's A-12 test
    - "Magnitude" of an improvement

Statistical Significance



	Base	A	B	C
Base				
A	2.58e-26		0.0022	6.96e-5
B	5.72e-23			0.194
C	5.61e-22			

Statistical Significance



	Base	A	B	C
Base				
A	2.58e-26		0.0022	6.96e-5
B	5.72e-23			0.194
C	5.61e-22			

Questions?