Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing

Stefan Nagy
snagy2@vt.edu

Matthew Hicks
mdhicks2@vt.edu
Fuzzing
An Overview of Fuzzing

Time-tested technique
AFL, honggFuzz, libFuzzer
CVE’s galore

Popular in the industry
Google, Microsoft

Fuzzing platforms
MSRD, OSS-Fuzz, FuzzBuzz, FuzzIt

Most popular: coverage-guided fuzzing

<table>
<thead>
<tr>
<th>LIJG jpeg</th>
<th>libjpeg-turbo</th>
</tr>
</thead>
<tbody>
<tr>
<td>libtiff</td>
<td>mozjpeg</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>Internet Explorer</td>
</tr>
<tr>
<td>Adobe Flash / PCRE</td>
<td>sqlite</td>
</tr>
<tr>
<td>LibreOffice</td>
<td>poppler</td>
</tr>
<tr>
<td>GnuTLS</td>
<td>GnuPG</td>
</tr>
<tr>
<td>PuTTY</td>
<td>ntpd</td>
</tr>
<tr>
<td>bash (post-Shellshock)</td>
<td>tcpdump</td>
</tr>
<tr>
<td>pdfium</td>
<td>ffmpeg</td>
</tr>
</tbody>
</table>

Source: lcamtuf.coredump.cx/afl
Coverage-guided Fuzzing

- Angora
- Steelix
- Fidgety AFL
- T-Fuzz
- QSYM
- SkyFire
- VUzzer
- Driller
- CollAFL
- AFLFast
- MutaGen
- AFLFast
- CollAFL

Coverage-guided Tracing

- New coverage ($<<N$)
- Trigger bugs
- No new coverage ($N$)

- $<<N$ test cases
- 0.3% overhead
- Orthogonal to tracing, generation
How are coverage-increasing test cases found?
By tracing **every** test case!

- **Dynamic translation**
- **Static callbacks**
- **Static inlining**

Dynamic translation is slower and binary-only ("black-box"). Static callbacks are faster and from source ("white-box").
How do fuzzers spend their time?

AFL – “naïve” fuzzing
Driller – “smart” fuzzing

8 benchmarks, 1hr trials

▼ O1: > 90% time on test case tracing, execution

▼ O2: < 3/10000 test cases increase coverage

<table>
<thead>
<tr>
<th>Fuzzer, tracer</th>
<th>Avg. % time on exec/trace</th>
<th>Avg. rate cvg.-incr. test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL-Clang</td>
<td>91.8</td>
<td>6.20E-5</td>
</tr>
<tr>
<td>AFL-QEMU</td>
<td>97.3</td>
<td>2.57E-4</td>
</tr>
<tr>
<td>Driller-QEMU</td>
<td>95.9</td>
<td>6.53E-5</td>
</tr>
</tbody>
</table>
Likelihood of coverage-increasing test cases?

AFL-QEMU

5x 24hr trials
x 8 benchmarks

▼ O3: rate decreases over time (< 1/10000)
Impact of tracing *every* test case?

▼ Over 90% of time is spent *tracing test cases*...
▼ Over 99.99% of which are *discarded*!

Equivalent to checking *every* straw to find the needle!
Why is tracing every test case expensive?

Storing coverage
- Bitmaps, arrays

Multiple additional instructions per block

Many blocks, edges

Long exec paths, loops

Overhead quickly adds up

<table>
<thead>
<tr>
<th>benchmark</th>
<th># blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsd.tar</td>
<td>31379</td>
</tr>
<tr>
<td>pdftohtml</td>
<td>54596</td>
</tr>
<tr>
<td>readelf</td>
<td>21249</td>
</tr>
<tr>
<td>tcpdump</td>
<td>33743</td>
</tr>
</tbody>
</table>

```
call loc.__afl_maybe_log
mov rax, qword [arg_10h]
mov rcx, qword [arg_8h]
mov rdx, qword [rsp] lea
    rsp, qword rsp + 0x98
```
Coverage-guided Tracing
Guiding Principle

Can we identify coverage-increasing test cases without tracing every test case?
Find New Coverage Without Tracing

Apply and dynamically remove interrupts
Coverage-guided Tracing

Approach: Trace only coverage-increasing test cases
"Filter-out" those that don’t hit an interrupt

▲ Common case (99.99%) don’t hit—thus aren’t traced
▲ Approaches native execution speed (0% overhead)
Incorporating CGT into Fuzzing

Implementation: **UnTracer**

\[ \oplus (N) \text{ of } (N): \text{native speed!} \]
Evaluation
### Performance Evaluation

**Goal: isolate tracing overhead**

1-core VM’s to avoid OS noise

Strip AFL to tracing-only code

8 diverse real-world benchmarks

Compare tracer exec times
  - 5 days’ test cases per benchmark
  - 5x trials per day of test cases

<table>
<thead>
<tr>
<th>Fuzzing Tracer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL-Dyninst</td>
<td>[BB] Static rewriting</td>
</tr>
<tr>
<td>AFL-QEMU</td>
<td>[BB] Dynamic translation</td>
</tr>
<tr>
<td>AFL-Clang</td>
<td>[WB] Assembly rewriting</td>
</tr>
<tr>
<td>UnTracer (Dyninst)</td>
<td>[BB] Coverage-guided Tracing (static rewriting)</td>
</tr>
</tbody>
</table>

[BB] = black-box (binary-only)
[WB] = white-box (from source)
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Benchmark type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsdTar (libarchive)</td>
<td>archiving</td>
</tr>
<tr>
<td>cert-basic (libksba)</td>
<td>cryptography</td>
</tr>
<tr>
<td>cjson (cjson)</td>
<td>web development</td>
</tr>
<tr>
<td>djjpeg (libjpeg)</td>
<td>image processing</td>
</tr>
<tr>
<td>pdfToHtml (poppler)</td>
<td>document processing</td>
</tr>
<tr>
<td>readelf (binutils)</td>
<td>development</td>
</tr>
<tr>
<td>sfconvert (audiofile)</td>
<td>audio processing</td>
</tr>
<tr>
<td>tcpdump (tcpdump)</td>
<td>networking</td>
</tr>
</tbody>
</table>
Can CGT beat tracing all with *Black-box*?

**AVG. relative overhead:**

- ▼ AFL-Dyninst: 518%
- ▼ AFL-QEMU: 618%
- ▲ UnTracer: 0.3%
Can CGT beat tracing all with *White-box*?

AVG. relative overhead:

- AFL-Dyninst: 518%
- AFL-QEMU: 618%
- UnTracer: 0.3%
- AFL-Clang: 36%
Can CGT boost *hybrid fuzzing* throughput?

**Goal:** measure impact on total test case throughput

QSYM (concolic exec + fuzzing)

8 benchmarks, 5x 24-hr trials

QSYM-UnTracer throughput:

- ▲ **616%**  >>  QSYM-QEMU
- ▲ **79%**  >>  QSYM-Clang
Conclusions: Why Coverage-guided Tracing?

▼ Fuzzers find coverage-increasing test cases by tracing *all of them*
▼ Costs *over 90% of time* yet *over 99.99%* are inevitably discarded

These resources could be better used to find bugs!

CGT restricts tracing to the few *guaranteed* to increase coverage

▲ Performance: Cuts tracing overhead from *36-618%* to *0.3%*
Boosts test case throughput by *79-616%*

▲ Compatibility: “Filter-out” approach allows plugging-in any tracer

▲ Orthogonality: Can combine with other fuzzing improvements
(e.g., better test case generation, faster tracing)
Thank you!

Our open-sourced software:

• **UnTracer-AFL**  |  UnTracer integrated with AFL
• **afl-fid**       |  AFL suite for fixed input datasets
• **FoRTÉ-FuzzBench** |  Our 8 real-world benchmarks

All repos are available here! [https://github.com/FoRTÉ-Research](https://github.com/FoRTÉ-Research)
Expanding Coverage Metrics

Current work: edge coverage, hit counts

Static critical edge handling doable

Hit counts need more complex transforms

Covered Blocks:
A, B, C
A, D, C

Implicit Edges:
A-B, B-C
A-D, D-C
CGT versus Hardware-Assisted Tracing

Can approximate Intel-PT overhead:

- AFL-Clang = 36% OH
- AFL-Clang $\approx$ 10-100% OH rel. to AFL-Clang-fast
- AFL-Clang-fast $\approx$ 18-32% OH
- Intel-PT $\approx$ 7% OH rel. to AFL-Clang-fast
- Intel-PT $\approx$ 19-35% OH

Trace decoding adds way more
Fully Black-box (binary-only) Implementation

Oracle forkserver uses assembly-time instrumentation

Theoretically doable via binary rewriting
  • Dyninst’s performance infeasible

Binary hooking an alternative
  e.g., via LD_PRELOAD
Appendix -- CGT step-by-step

**Intuition:** restrict tracing to coverage-increasing test cases

1. Statically overwrite start of each block with an interrupt
   - The “Interest Oracle”
2. Get a new test case and run it on the oracle
3. If an interrupt is triggered:
   - Trace the test case’s code coverage
   - Unmodify (reset) all newly-covered blocks
4. Return to step 2
As more blocks unmodified over time, binary starts to mirror the original

Thus, most testcases are run at native execution speed!
Appendix -- Implementation: UnTracer

- Built atop AFL
- Dyninst for CFG/tracing
- File I/O for mod/unmod