Same Coverage, Less Bloat:
Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing

Stefan Nagy
snagy2@vt.edu

Anh Nguyen-Tuong
nguyen@virginia.edu

Jason D. Hiser
hiser@virginia.edu

Jack W. Davidson
jwd@virginia

Matthew Hicks
mdhicks2@vt.edu
Background
Software Fuzz-testing (Fuzzing)

• Today’s leading automated bug-finding approach
• Uncover bugs by bombarding program with inputs
• Coverage-guided search: breed only the winners
  • Measure each input’s code coverage via tracing
  • Keep and mutate only those reaching new code
On average, fewer than 1 in 10,000 inputs reach new code coverage.

For binary-only fuzzing, compounded by upwards of 10x slower speed.
Coverage-guided Tracing (CGT)

Filter-out the **99.9%** of useless inputs at native speed *without* tracing

Overhead approaches 0% = **orders-of-magnitude** faster binary fuzzing
Despite some adoption, CGT’s performance advantages remain **sidelined** by the majority of today’s fuzzers.

**Why?** Most rely on **edge** and **hit count** coverage metrics, yet **CGT only supports binarized basic block** coverage.
The Code Coverage Dilemma

For \textit{critical} edge $A \rightarrow C$:

- **Edge Coverage**
  - Will capture \textit{every} edge irrespective of path taken

- **CGT: Block-level Coverage**
  - If path $A \rightarrow B \rightarrow C$ seen first, \textit{can’t} discern edge $A \rightarrow C$

For \textit{back} edge $C \rightarrow A$:

- **Hit Counts**
  - Will capture \textit{each count} backwards edge is taken

- **CGT: Binarized Counts**
  - \textit{Can’t} discern \textit{any count} edge $C \rightarrow A$ is re-taken
## The Code Coverage Dilemma

<table>
<thead>
<tr>
<th>Name</th>
<th>Covg</th>
<th>Hits</th>
<th>Name</th>
<th>Covg</th>
<th>Hits</th>
<th>Name</th>
<th>Covg</th>
<th>Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>Edge</td>
<td>✓</td>
<td>EnFuzz</td>
<td>Edge</td>
<td>✓</td>
<td>ProFuzzer</td>
<td>Edge</td>
<td>✓</td>
</tr>
<tr>
<td>AFL++</td>
<td>Edge</td>
<td>✓</td>
<td>FairFuzz</td>
<td>Edge</td>
<td>✓</td>
<td>QSYM</td>
<td>Edge</td>
<td>✓</td>
</tr>
<tr>
<td>AFLFast</td>
<td>Edge</td>
<td>✓</td>
<td>honggfuzz</td>
<td>Edge</td>
<td>✕</td>
<td>REDQUEEN</td>
<td>Edge</td>
<td>✓</td>
</tr>
<tr>
<td>AFLSmart</td>
<td>Edge</td>
<td>✓</td>
<td>GRIMORE</td>
<td>Edge</td>
<td>✓</td>
<td>SAVIOR</td>
<td>Edge</td>
<td>✓</td>
</tr>
<tr>
<td>Angora</td>
<td>Edge</td>
<td>✓</td>
<td>lafIntel</td>
<td>Edge</td>
<td>✓</td>
<td>SLF</td>
<td>Edge</td>
<td>✓</td>
</tr>
<tr>
<td>CollAFL</td>
<td>Edge</td>
<td>✓</td>
<td>libFuzzer</td>
<td>Edge</td>
<td>✓</td>
<td>Steelix</td>
<td>Edge</td>
<td>✓</td>
</tr>
<tr>
<td>DigFuzz</td>
<td>Edge</td>
<td>✓</td>
<td>Matryoshka</td>
<td>Edge</td>
<td>✓</td>
<td>Superion</td>
<td>Edge</td>
<td>✓</td>
</tr>
<tr>
<td>Driller</td>
<td>Edge</td>
<td>✓</td>
<td>MOpt</td>
<td>Edge</td>
<td>✓</td>
<td>TIFF</td>
<td>Block</td>
<td>✓</td>
</tr>
<tr>
<td>Eclipser</td>
<td>Edge</td>
<td>✓</td>
<td>NEUZZ</td>
<td>Edge</td>
<td>✓</td>
<td>VUzzzer</td>
<td>Block</td>
<td>✓</td>
</tr>
</tbody>
</table>

Is it possible to uphold the **high speed of CGT** while meeting existing fuzzers’ coverage demands?
Coverage-preserving
Coverage-guided Tracing
Guiding Principle

How can CGT’s lightweight, interrupt-driven coverage support finer-grained edge and hit count coverage?

To extend CGT beyond binarized block coverage, we must find ways to make these finer-grained control-flows self-report their coverage.
Conventional Edge Coverage at Block Level

Resolving critical edges
- Edges whose start, end have 2+ out, in edges (respectively)
- If non-critical path is first, critical edge (A→C) never seen!

Naive approach: split each with new dummy block
- Covering a dummy (D) implicitly covers its critical edge
- To facilitate CGT, add interrupts on every dummy

Problem: splitting adds 30–40% more basic blocks
- Accumulates more and more overhead over native speed

Splitting each critical edge with new basic blocks will deteriorate CGT’s performance
How do critical edges manifest?

**Observation:** 89% of fuzzer-covered critical edges are **conditional jump target** branches.
Optimizing Common-case Critical Edges

**Observation:** conditional jumps’ targets are self-encoded

- Jump instruction encoding:

  \[
  [\text{opcode}] [\text{PC-relative displacement}]
  \]

- To resolve a jump to a target address:

  \[
  [\text{opcode}] = \text{Intuition: rewrite and force execution to an interrupt!}
  \]

To signal the edge as taken, we can resolve its target to a CGT-style interrupt
Our Solution: Jump Mistargeting

- Modify jump target to resolve in a **CGT-style interrupt**

- Following a crash, **restore** displacement for future test cases

**Outcome**: CGT-style edge coverage **at native speed** (i.e., **zero additional** basic blocks or instructions)
Conventional Hit Count Coverage Tracking

Most fuzzers rely on AFL-style bucketed hit counts:

\[
\begin{align*}
\end{align*}
\]

Advances to higher buckets (e.g., \([3] \rightarrow [4,7]\)) flagged interesting

Problem: implemented within always-on instrumentation
  
  • Increments each edge’s unique counter for each execution

Hit count tracking’s reliance on exhaustive tracing contradicts CGT’s only-when-needed tracing mindset
Why are hit counts important?

A testing property of cycles (e.g., loops)

Unlocking deeper loop iterations
• Common precedent for many critical bugs

Differentiating progress of nested loops
• Maximal consecutive iterations

Observation: Hit counts primarily guide fuzzing toward higher loop exploration progress
Optimizing Loop Hit Count Tracking

**Observation:** loops’ induction variables encode their iterations

```c
for( int i = 0; i < 100; i = i + 1 ){
}
```

- Track jumps to higher buckets via range check on induction variable

```c
for(i=0; i<100; i++){
    if (i > 1)
        15
    2
    3
    7
}
```

Intuition: use *interrupt* to detect crossing buckets!

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 2]</td>
</tr>
</tbody>
</table>
| 2         | [2, 31]|*
| 3         | [3, 127]|*
| 4         | [15, 31]|*
| 5         | [16, 127]|*
| 6         | [32, 127]|*
| 7         | [128, 128+]|*

To signal a loop’s change in a hit count buckets, we can use a range check guarded by CGT-style interrupts.
Our Solution: Bucketed Unrolling

- Inject discrete interval checks (with *interrupts* on all *false* edges)

- If crash, entered a *higher* bucket; then *clear* interrupt and move on

**Outcome:** CGT-style hit counts *without* relying on always-on tracing
Implementation: HeXcite

• **High-Efficiency eXpanded Coverage for Improved Testing of Executables**

• **Binary-only** fuzzer built atop **AFL 2.52b** and **ZAFL** fuzzing rewriter

• Jump mistargeting:
  • Implementation based on *zero-address* mistargeting
  • Critical edge identification performed after control-flow parsing
  • Jumps converted to 32-bit displacements (e.g., all are mistargetable)

• Bucketed unrolling:
  • Implementation based on conventional AFL-style eight ranges
  • Loop identification performed via standard back edge analysis
  • For simplicity, we insert a fake induction variable and incrementor
Evaluation
Evaluation Setup

<table>
<thead>
<tr>
<th>Approach</th>
<th>Tracing Type</th>
<th>Level</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeXcite</td>
<td>coverage-guided</td>
<td>binary</td>
<td>edge + counts</td>
</tr>
<tr>
<td>UnTracer</td>
<td>coverage-guided</td>
<td>binary</td>
<td>block</td>
</tr>
<tr>
<td>QEMU</td>
<td>always-on</td>
<td>binary</td>
<td>edge + counts</td>
</tr>
<tr>
<td>Dyninst</td>
<td>always-on</td>
<td>binary</td>
<td>edge + counts</td>
</tr>
<tr>
<td>RetroWrite</td>
<td>always-on</td>
<td>binary</td>
<td>edge + counts</td>
</tr>
<tr>
<td>Clang</td>
<td>always-on</td>
<td>source</td>
<td>edge + counts</td>
</tr>
</tbody>
</table>

- **Benchmarks**: 8 diverse open-source + 4 closed-source binaries
- **Evaluations**: perform 16x24-hr trials per benchmark on Azure cloud
- **Edge coverage**: collect with LLVM instrumentation and AFL tools
- **Loop coverage**: compute max consecutive iterations capped at 128
- **Performance**: scale throughput relative to worst-performing competitor
- **Bug-finding**: crash triage performed via AddressSanitizer
Does HeXcite improve edge coverage?

6.2% more edges than block-only UnTracer

23.1%, 18.1%, and 6.3% more edges than binary-level QEMU, Dyninst, and RetroWrite

1.1% more edges than source-level AFL-Clang
Does HeXcite improve loop exploration?

130% more iterations than block-only UnTracer

36% more iterations than source-level AFL-Clang
Is HeXcite as fast as block-only CGT?

10% higher best-case than block-only UnTracer

11.4x, 24.1x, and 3.6x the fuzzing throughput of binary-level QEMU, Dyninst, and RetroWrite

2.8x the throughput of source-level AFL-Clang
Can HeXcite improve binary bug-finding?

12% more bugs than block-only UnTracer

521%, 749%, and 56% more bugs than binary-level QEMU, Dyninst, and RetroWrite

46% more bugs than source-level AFL-Clang
Does HeXcite accelerate bug-finding?

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Category</th>
<th>Binary</th>
<th>Coverage-guided Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2011-4517</td>
<td>heap overflow</td>
<td>jasper</td>
<td>13.1 hrs, 18.2 hrs</td>
</tr>
<tr>
<td>GitHub issue #58-1</td>
<td>stack overflow</td>
<td>mjs</td>
<td>13.3 hrs, 19.0 hrs</td>
</tr>
<tr>
<td>GitHub issue #58-2</td>
<td>stack overflow</td>
<td>mjs</td>
<td>13.6 hrs, 16.4 hrs</td>
</tr>
<tr>
<td>GitHub issue #58-3</td>
<td>stack overflow</td>
<td>mjs</td>
<td>5.88 hrs, 6.80 hrs</td>
</tr>
<tr>
<td>GitHub issue #58-4</td>
<td>stack overflow</td>
<td>mjs</td>
<td>8.60 hrs, 10.7 hrs</td>
</tr>
<tr>
<td>GitHub issue #136</td>
<td>stack overflow</td>
<td>mjs</td>
<td>1.30 hrs, 7.50 hrs</td>
</tr>
<tr>
<td>Bugzilla #3392519</td>
<td>null pointer deref</td>
<td>nasm</td>
<td>12.1 hrs, 13.5 hrs</td>
</tr>
<tr>
<td>CVE-2018-8881</td>
<td>heap overflow</td>
<td>nasm</td>
<td>5.06 hrs, 14.6 hrs</td>
</tr>
<tr>
<td>CVE-2017-17814</td>
<td>use-after-free</td>
<td>nasm</td>
<td>3.54 hrs, 6.31 hrs</td>
</tr>
<tr>
<td>CVE-2017-10686</td>
<td>use-after-free</td>
<td>nasm</td>
<td>3.84 hrs, 5.40 hrs</td>
</tr>
<tr>
<td>Bugzilla #3392423</td>
<td>illegal address</td>
<td>nasm</td>
<td>8.17 hrs, 14.2 hrs</td>
</tr>
<tr>
<td>CVE-2008-5824</td>
<td>heap overflow</td>
<td>sfconvert</td>
<td>13.1 hrs, 14.8 hrs</td>
</tr>
<tr>
<td>CVE-2017-13002</td>
<td>stack over-read</td>
<td>tcpdump</td>
<td>8.34 hrs, 12.5 hrs</td>
</tr>
<tr>
<td>CVE-2017-5923</td>
<td>heap over-read</td>
<td>yara</td>
<td>3.24 hrs, 5.67 hrs</td>
</tr>
<tr>
<td>CVE-2020-29384</td>
<td>integer overflow</td>
<td>pngout</td>
<td>5.40 min, 34.5 min</td>
</tr>
<tr>
<td>CVE-2007-0855</td>
<td>stack overflow</td>
<td>unrar</td>
<td>10.7 hrs, 17.6 hrs</td>
</tr>
</tbody>
</table>

52.4% exposure speedup over block-only UnTracer
Conclusion: Why Coverage-preserving CGT?

• Maximizing fuzzing performance is critical for effective bug-finding.
• Yet, the coverage shortcomings of **Coverage-guided Tracing**—fuzzing’s **fastest** tracing strategy—restrict fuzzers to far slower, **always-on tracing**.

Making CGT’s **orders-of-magnitude** faster tracing available to **all fuzzers** demands extending it to the finer-grained coverage metrics used today: **edges** and **hit counts**.

By forcing finer-grained control-flow to **self-report** its coverage, we expand CGT to **binary-level edge** and **hit count coverage** at virtually **no performance loss**.

• Fuzzing speed: 2.8—24.1x higher than binary- and source-level tracing
• Code coverage: 6.2% more edges and 130% deeper loops than **block-only CGT**
• Bug-finding: 12—749% more bugs than block-only CGT and always-on tracing
Thank you!

Find HeXcite and our evaluation benchmarks at:

https://github.com/FoRTE-Research/hexcite

Happy \textit{(binary)} fuzzing!

Attribution for images throughout this presentation: www.vecteezy.com