Week 9: Lecture A

Optimization I

Monday, March 11, 2024

Lab 3 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.

Lab 3 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**?

Lab 3 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
```

Lab 3 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!

Lab 3 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!

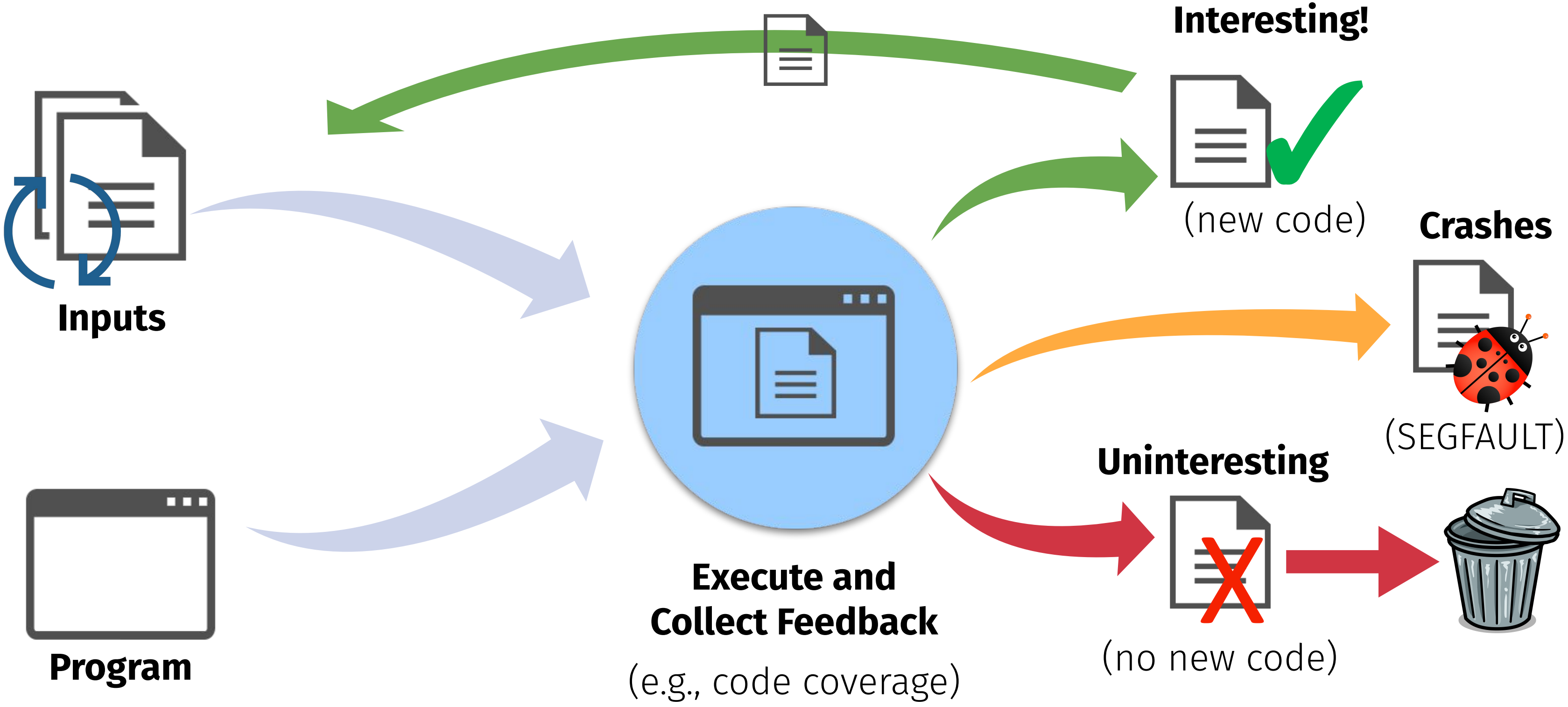
Recap: Project Schedule

- **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

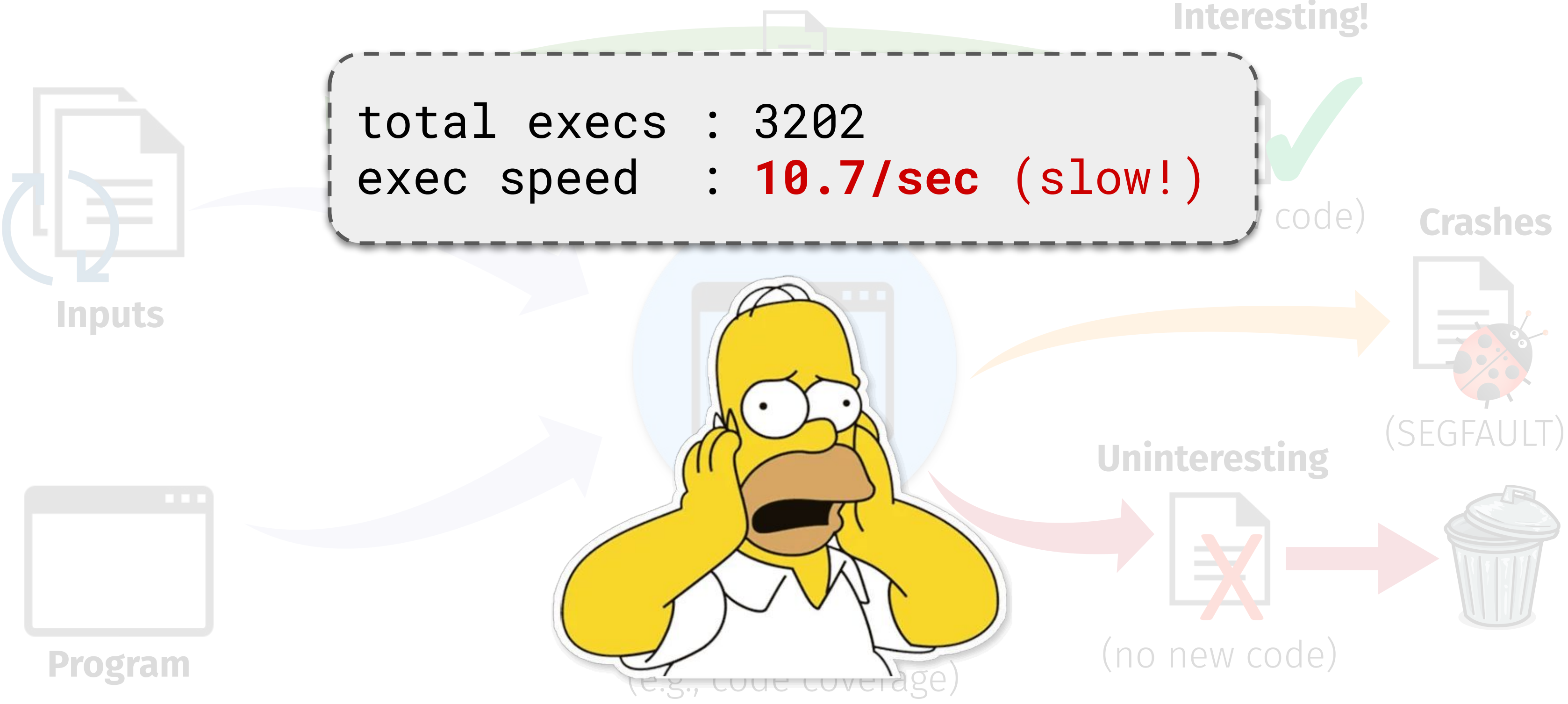
Questions?

Fuzzing Faster

Recap: Coverage-guided Fuzzing



Recap: Coverage-guided Fuzzing




```
total execs  : 3202
exec speed   : 10.7/sec (slow!)
```

What affects fuzzing speed?

- **Process execution**
  - Performed on every input
- **Runtime instrumentation**
  - Code coverage tracing
- **Information post-processing**
  - Data structure writing/reading
  - Other essential computation

Why is speed so important?

- **Need to find the bugs before attackers do**
  - Time is money; bug-finders limited by time/resource budgets
  - Race to find and fix before monthly “Patch Tuesday”

- **People’s privacy (and lives) at stake**
  - Nation-state attackers have unlimited budgets
  - They’re in it to win it just as much

Complexity adds Overhead

- **Fancy/slow is often less effective than crude/fast**
  - E.g., taint tracking-based fuzzing vs. good ol' AFL
  - **Academically interesting is not always practical**

Applications	Version	AFL	CollAFL- br	Honggfuzz	VUzzer
libbson	1.8.0	1	1	1	0
libsndfile	1.0.28	1	2	2	1
libconfuse	3.2.2	1	2	0	0
libwebm	1.0.0.27	1	1	0	0
libsolv	2.4	0	0	3	2
libcaca	0.99beta19	2	4	1	0
liblas	2.4	1	2	0	0
libslax	20180901	3	5	0	0
libsixl	v1.8.2	2	2	2	2
libxsmm	release-1.10	1	1	2	0
Total	-	21	34	18	6

Table 1: Number of vulnerabilities (accumulated in 5 runs)

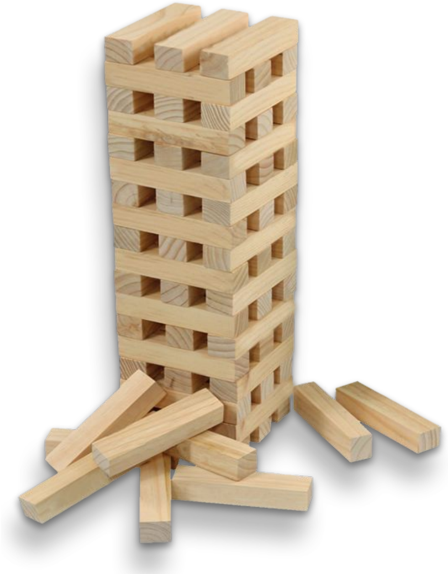
Source: GREYONE: Data Flow Sensitive Fuzzing

Pre-execution Optimization

Test Case Minimization

- **Test cases get larger as fuzzing continues**
  - More program execution = more overhead
  - Need to **cut-out unnecessary execution**

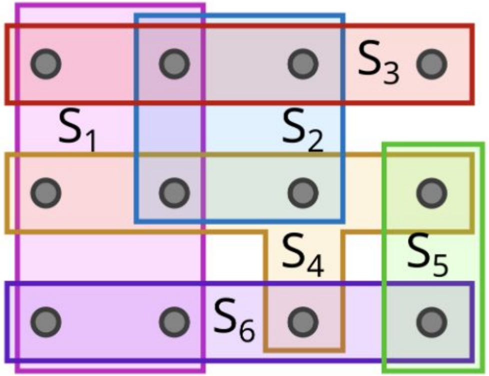
- **Delta debugging:** change, then check
  - Iteratively remove input bytes
  - Check if code coverage changes
    - If coverage changes, undo
    - **Like one big jenga game**



Corpus Minimization

- **Test case corpus grows as fuzzing continues**
  - Lots of test cases reach new edge, hit count coverage
  - Many test cases have overlapping code coverage
  - **Fuzzer will struggle to pick the "best" one**

- **Corpus minimization:** condense your corpus
  - I.e., smallest set that covers all edges seen so far
  - **AFL:** also minimize file size and execution time



Source: https://securityboulevard.com/2021/10/generating-a-tiny-corpus-with-greedy-set-cover-minimization/

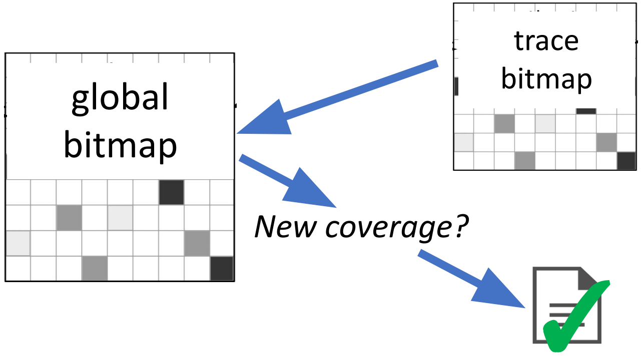
Trade-offs

- Complicated for **highly-structured inputs**
  - E.g., JPEG images versus ELF executables
  - Byte-level changes won't work on the latter
  - Grammar-level mutations require more machinery
- Complicated by **code coverage granularity**
  - E.g., edges versus hit counts
  - Finer-grained info is harder to condense
  - **Still an unsolved research problem**

Post-execution Optimization

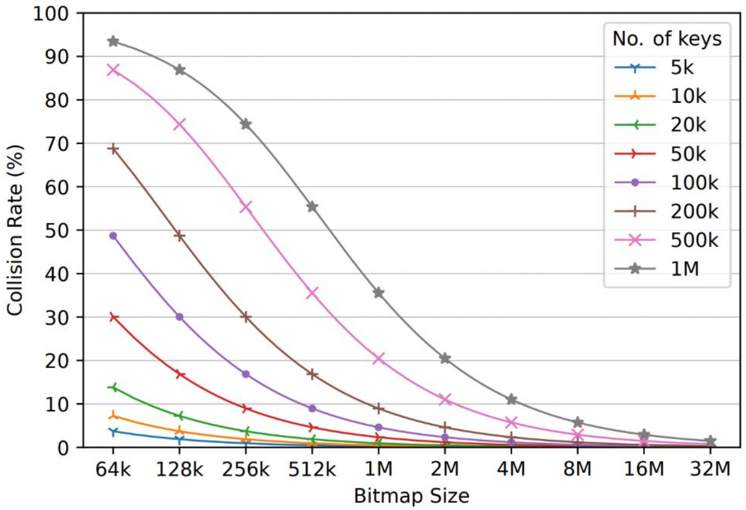
Storing Information

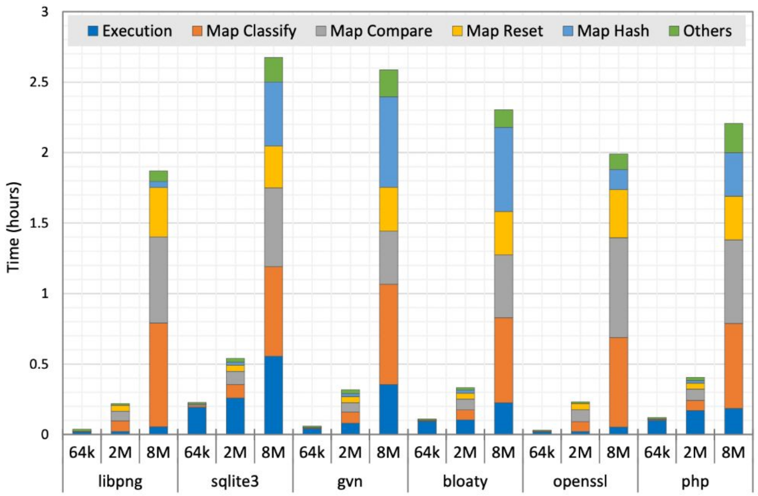
- **Must store information in data structures**
  - E.g., bitmaps for code coverage traces
  - E.g., ASTs for dynamically-learned grammars
- **Data structure design affects fuzzing speed**
  - Memory footprint
  - Cost of reads/writes



Trade-offs

- **Best case:** small enough to fit in L2 cache
  - But, smaller size sacrifices information storage





Source: BigMap: Future-proofing Fuzzers with Efficient Large Maps

Intra-execution Optimization

Recap: AFL's Edge Coverage

- Edge coverage via hashed basic block tuples

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

Recap: AFL's Edge Coverage

- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

Recap: AFL's Edge Coverage

- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

  - **Edge hash:** current basic block ID is **XOR'd** to previous basic block's

Recap: AFL's Edge Coverage

- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

  - **Edge hash:** current basic block ID is **XOR'd** to previous basic block's
    - Edge-specific hit counter incremented by one for each exercising

Recap: AFL's Edge Coverage

- Edge coverage via hashed basic block tuples
  - Each basic block assigned a random ID at compile-time

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

  - **Edge hash:** current basic block ID is **XOR'd** to previous basic block's
    - Edge-specific hit counter incremented by one for each exercising
  - **Right shift** current block to preserve edge **directionality** (because XOR is commutative)
    - Enables A→B to be seen as distinct from B→A; also A→A from B→B

Instrumenters: How Instrumentation is Added

Compiler	LLVM Compiler Infrastructure, GCC
Binary (static)	Dyninst
Binary (dynamic)	Pin, FRIDA, QEMU

Instrumenters: How Instrumentation is Added

- **Open-source: compiler instrumentation**
  - Bake-in instrumentation code at compile-time
  - **Efficient** and **correct**
- **Closed-source: dynamic binary translation**
  - Instrument program as it is executing
  - Generally **correct** but **inefficient**
- **Closed-source: static binary rewriting**
  - Instrument program before it executes
  - Generally **incorrect** but **efficient**

Instrumenters: How Instrumentation is Added

- **Open-source: compiler instrumentation**
  - Bake-in instrumentation code at compile-time
  - **Efficient** and **correct**
- **Closed-source: dynamic binary translation**
  - Instrument program as it is executing
  - Generally **correct** but **inefficient**
- **Closed-source: static binary rewriting**
  - Instrument program before it executes
  - Generally **incorrect** but **efficient**

Key pillars of fuzzing instrumentation speed:

Use **faster** instrumentation

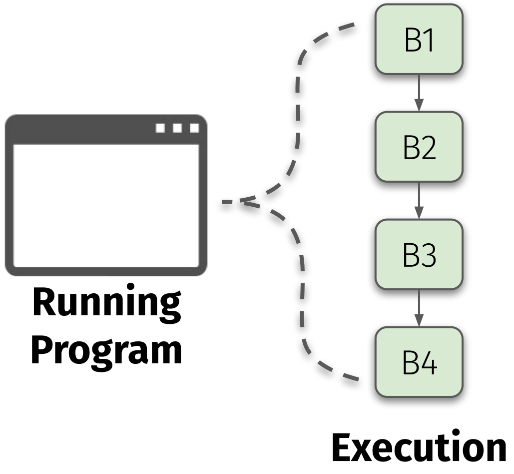
Use **less** instrumentation

Faster Instrumentation

Binary Instrumentation

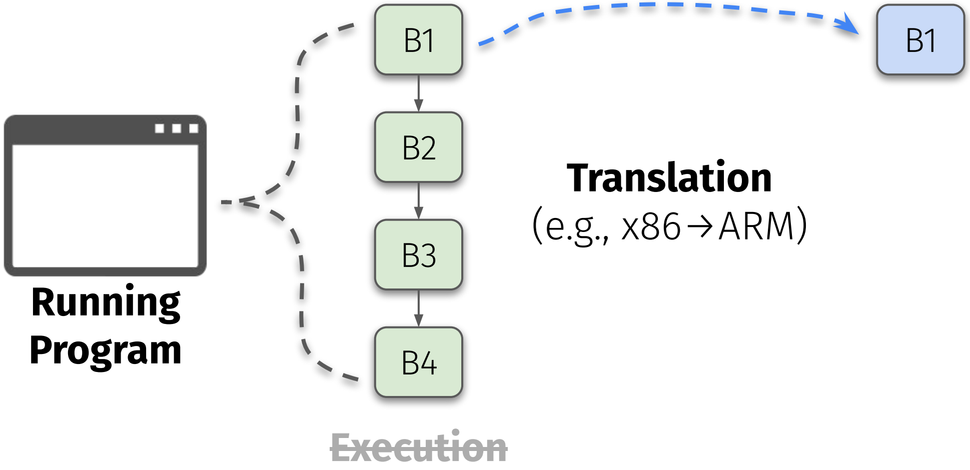
- **Dynamic binary translation**
  - **Idea:** translate basic blocks to host ISA

QEMU



Binary Instrumentation

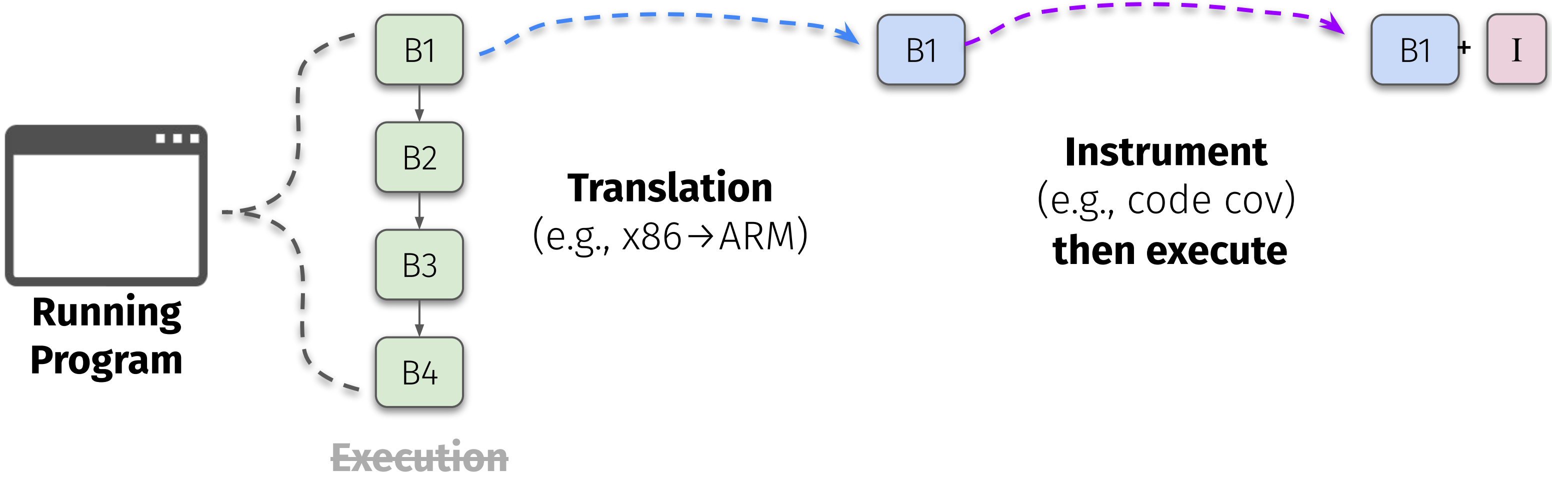
- **Dynamic binary translation**
  - **Idea:** translate basic blocks to host ISA



Binary Instrumentation

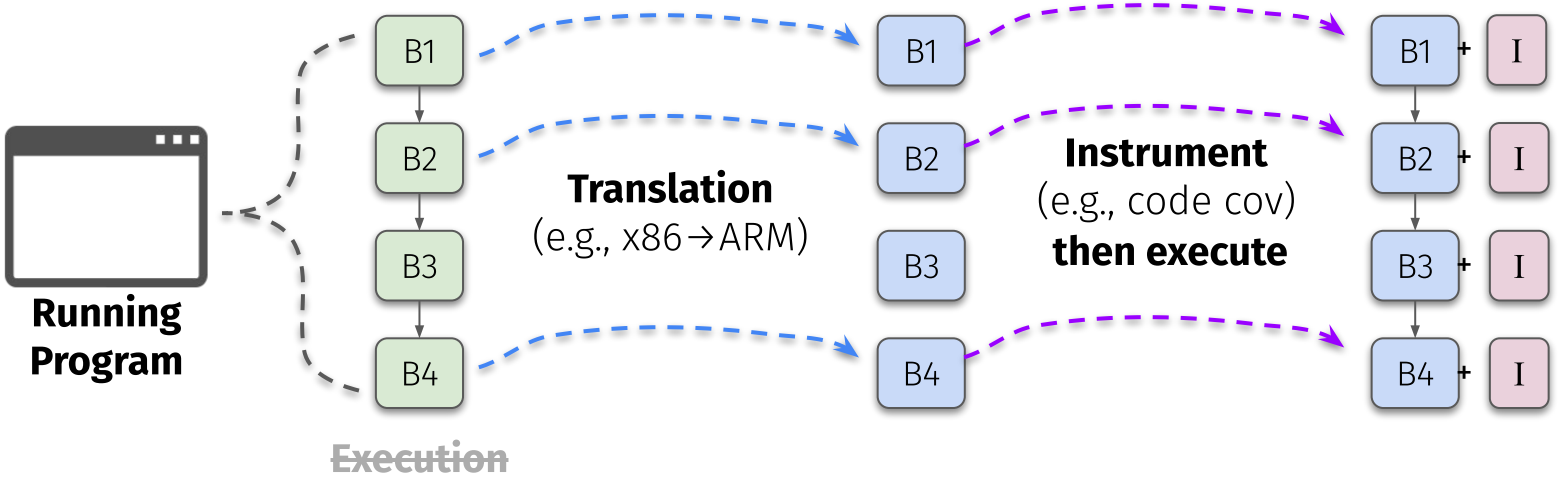
- **Dynamic binary translation**
  - **Idea:** translate basic blocks to host ISA

QEMU



Binary Instrumentation

- **Dynamic binary translation**
  - **Idea:** translate basic blocks to host ISA



Binary Instrumentation

- **Dynamic binary translation**
  - **Idea:** translate basic blocks to host ISA
  - Primary expense comes from **translation**
    - Performed on **every** piece of code
    - **Re-translate** already seen code

Binary Instrumentation

- **Dynamic binary translation**
  - **Idea:** translate basic blocks to host ISA
  - Primary expense comes from **translation**
    - Performed on **every** piece of code
    - **Re-translate** already seen code
  - **Solution:** make already-seen code **cached**
    - Avoid re-translating as much as possible
  - **Problem:** still really slow even with caching!
    - Upwards of **600% slower** than compilers!

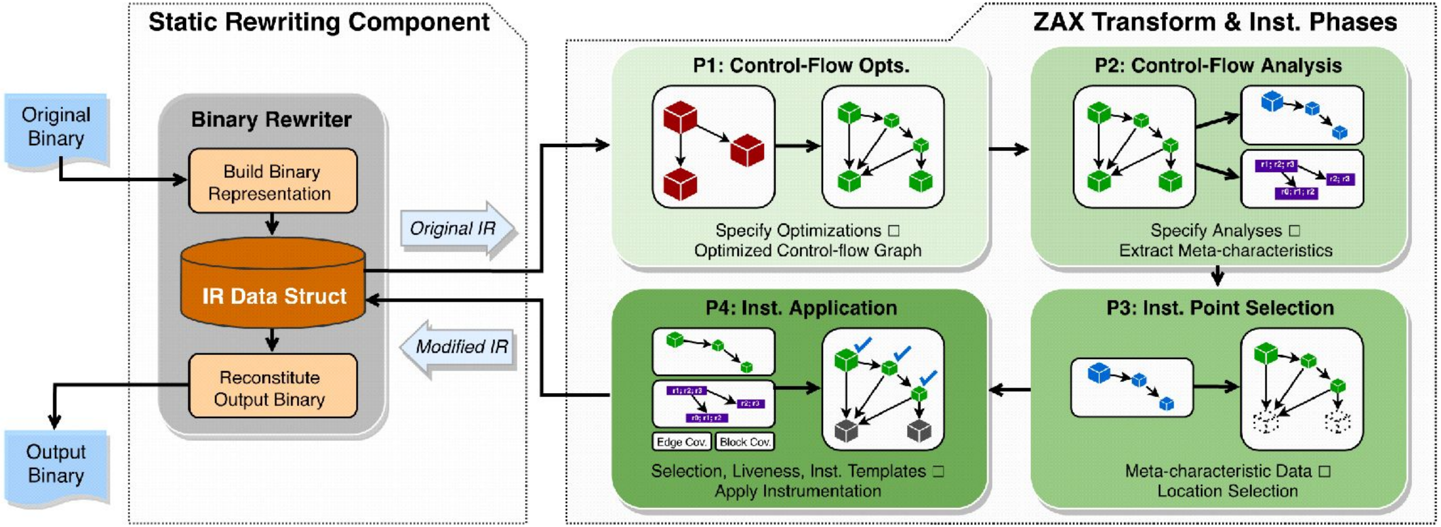
QEMU

DynamoRIO

FRIDA

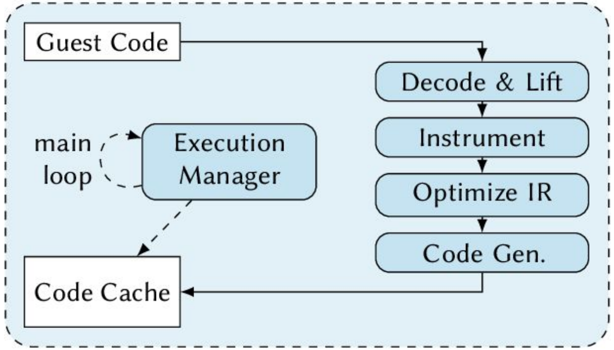
Faster Binary Instrumentation

- **Our solution (ZAFL):** design **static rewriters** to match compilers
  - Achieves **compiler-level speeds** for closed-source targets



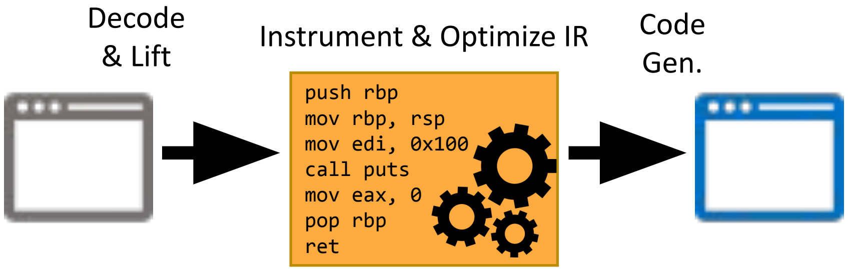
ZAFL's Design Decisions

Dynamic Binary Translation



- Analyze / instrument **during** runtime
- Repeatedly pay translation cost

Static Binary Rewriting



- Perform all tasks **prior to** runtime
- Analogous to compiler (e.g., LLVM IR)

ZAFL's Design Decisions

Trampolined Invocation

Original

```
push rbp
mov rbp, rsp
mov edi, 0x100
call puts
mov eax, 0
mov edi, 7
call payload
pop rbp
ret
```

Trace

Return

Instrumentation

```
payload:
  mov ecx, _prev
  xor ecx, edi
  shr edi
  mov _prev, edi
  ret
```

- Transfer to / from "payload" function
- Repeatedly pay **flow redirection** cost

Inlined Invocation

Original

```
push rbp
mov rbp, rsp
mov edi, 0x100
call puts
mov eax, 0
pop rbp
ret
```

Instrumented

```
push rbp
mov rbp, rsp
mov edi, 0x100
call puts
mov eax, 0
mov edi, 7
mov ecx, _prev
xor ecx, edi
shr edi
mov _prev, edi
pop rbp
ret
```

Trace

- Weave new instructions **with original**
- Preferred mechanism of compilers

ZAFL's Design Decisions

Liveness Unaware

Original

```
push rbp
mov rbp, rsp
mov edi, 0x100
call puts
mov eax, 0
pop rbp
ret
```

Instrumented

```
push rbp
mov rbp, rsp
mov edi, 0x100
call puts
mov eax, 0
push edi        } Save Regs
push ecx        }
mov edi, 7
mov ecx, _prev
xor ecx, edi
shr edi
mov _prev, edi
pop ecx         } Restore Regs
pop edi         }
pop rbp
ret
```

- Transfer to / from "payload" function
- Repeatedly pay **flow redirection** cost

Liveness Aware

Original

```
push rbp
mov rbp, rsp
mov edi, 0x100
call puts
mov eax, 0
pop rbp
ret
```

Instrumented

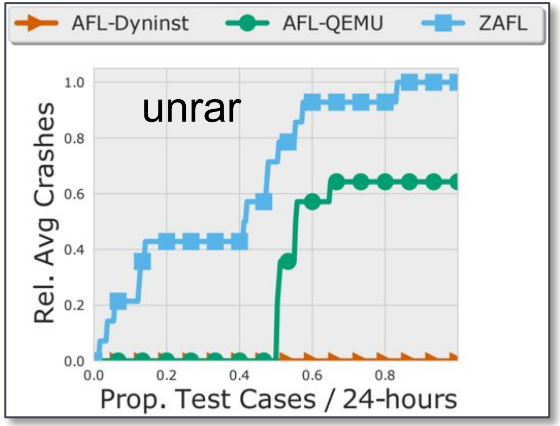
```
push rbp
mov rbp, rsp
mov edi, 0x100
call puts
mov eax, 0
mov edi, 7        }
mov ecx, _prev    }
xor ecx, edi      } Trace
shr edi           }
mov _prev, edi    }
pop rbp
ret
```

- Track liveness to **prioritize dead regs**
- Critical to compiler code optimization

ZAFL's Performance

- **Our solution (ZAFL):** design **static rewriters** to match compilers
  - Achieves **compiler-level speeds** for closed-source targets
    - Finds vulnerabilities faster than other binary tracers

Vulnerability Type	Executable	Dyninst	QEMU	ZAFL
Heap Overflow	nconvert	Can't find	**18.3** hrs	**12.7** hrs
Heap Overflow	unrar	Can't find	**12.3** hrs	**9.04** hrs
Use-After-Free	pngout	**12.6** hrs	**6.26** hrs	**1.93** hrs
Use-After-Free	pngout	**9.35** hrs	**4.67** hrs	**1.44** hrs
Heap Overflow	IDA Pro	**23.7** hrs	Can't find	**2.30** hrs
ZAFL's Mean Relative Decrease		**-660%**	**-113%**	



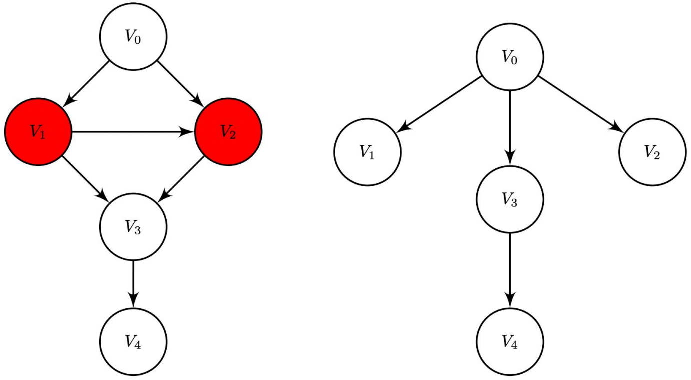
Hardware-assisted Tracing

- **Collect coverage via fast CPU mechanisms**
  - E.g., Intel Processor Trace, ARM Coresight
  - An emerging feature used in binary fuzzing
- **Trade-offs:**
  - Attains speeds similar to compiler instrumentation
  - Only usable (and effective) on specific hardware
    - ARM Coresight is way slower than Intel PT
  - Cannot instrument programs to do other things
    - E.g., hooking and logging CMP instructions

Less Instrumentation

Instrumentation Culling

- Save overhead by **instrumenting less of the program**
  - **Crude approach:** instrument code at **random**
  - **Smart approach:** instrument leaf nodes of **dominator tree**
    - *A dominates B iff every path to B first intersects A*
    - Cuts down about 30–50% of basic blocks



(a) CFG (b) dominator tree of (a)

Instrumentation Optimization

- **Downgrade** from edge to block-based instrumentation
  - Save a few instructions (i.e., from computing edge hashes)
  - Saved for basic blocks with **single predecessors**

```
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

↓

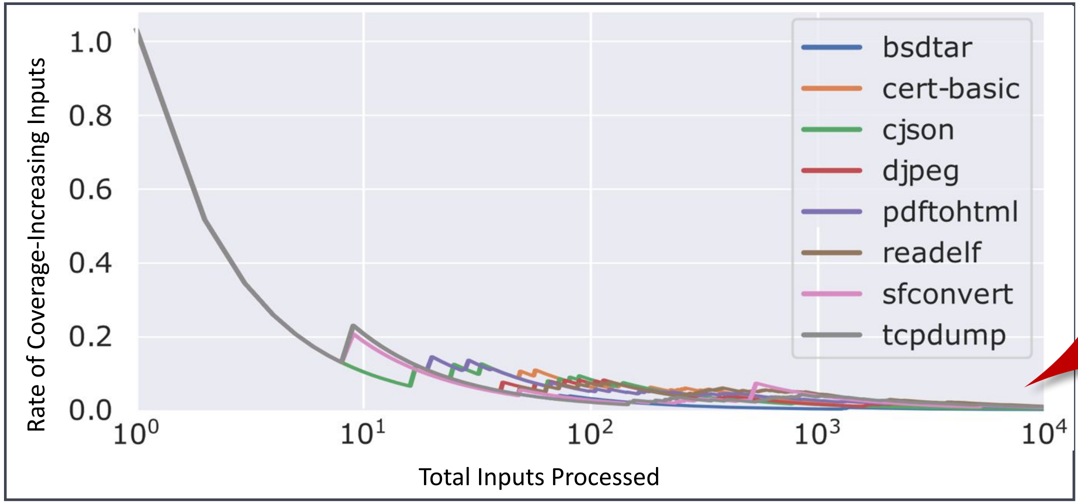
```
Shared_mem [PreDeterminedIID]++;
```

Why trace every single test case?

- Equivalent to checking **each** straw to find **one** needle
  - Cost adds up from instrumentation’s **instruction footprint**
    - 3–5 additional instructions per basic block
    - More instructions from post-processing coverage

Why trace every single test case?

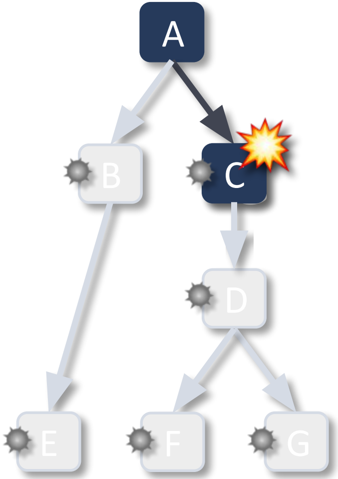
- **Less than 1%** of all inputs reach new code coverage
  - The other 99.9% are **discarded** right after tracing
  - **Wasted resources!**



Less than *one* per 10,000

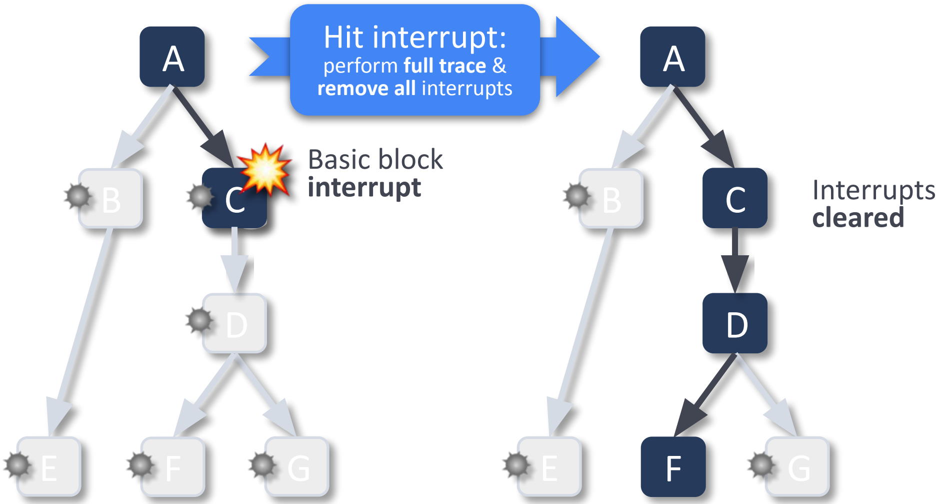
Coverage-guided Tracing

- **Idea:** restrict tracing to **only when new coverage is *guaranteed***
  - Guaranteed how? By using **interrupts**!



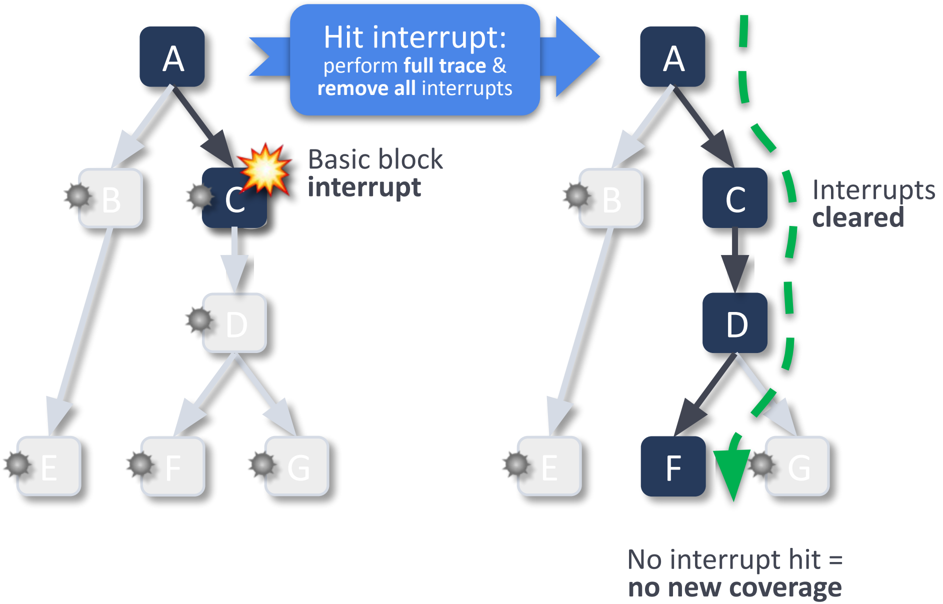
Coverage-guided Tracing

- **Idea:** restrict tracing to **only when new coverage is *guaranteed***
  - Guaranteed how? By using **interrupts**!



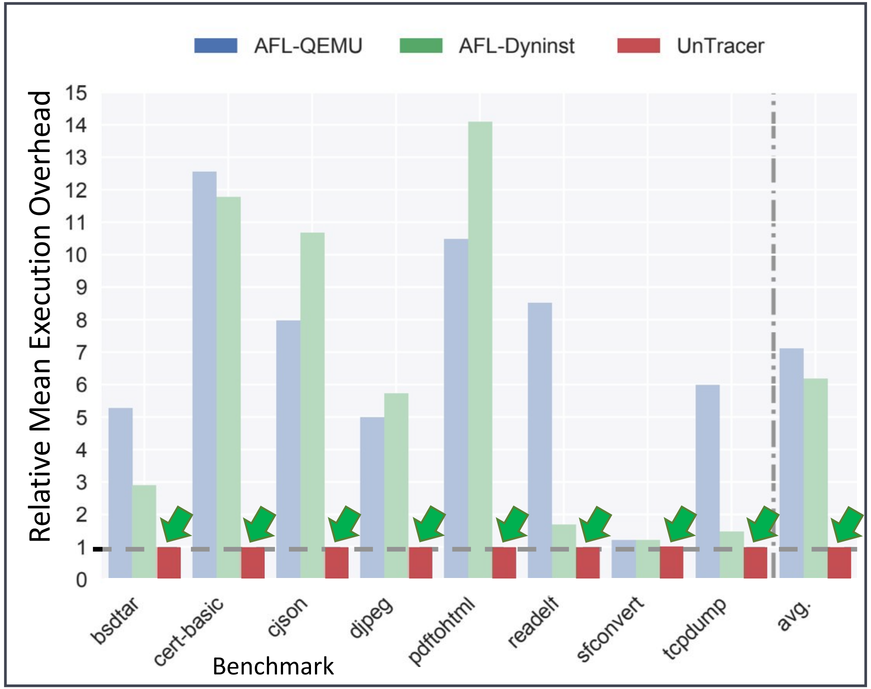
Coverage-guided Tracing

- **Idea:** restrict tracing to **only when new coverage is *guaranteed***
  - Guaranteed how? By using **interrupts**!



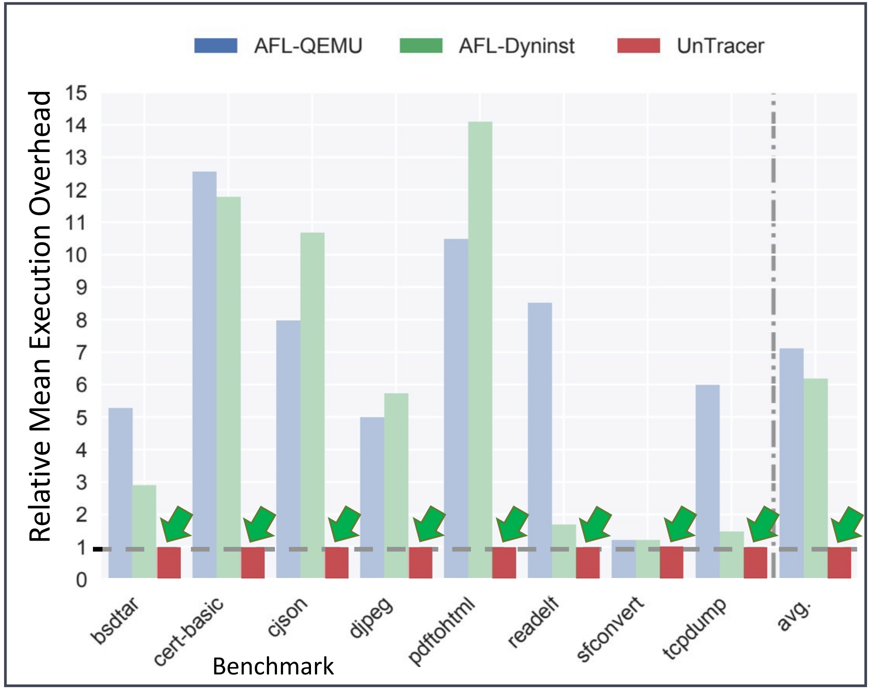
Coverage-guided Tracing

- **Implementation: UnTracer**
  - Averages just **0.3%** overhead
  - Coverage-guided fuzzing at the speed of **black-box** fuzzing
  - **Caveats?**



Coverage-guided Tracing

- **Implementation: UnTracer**
  - Averages just 0.3% overhead
  - Coverage-guided fuzzing at the speed of black-box fuzzing
  - **Caveats?**
    - Only basic block coverage
    - No edges or hit counts!



Questions?