Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing

Stefan Nagy
snagy2@vt.edu

Anh Nguyen-Tuong
nguyen@virginia.edu

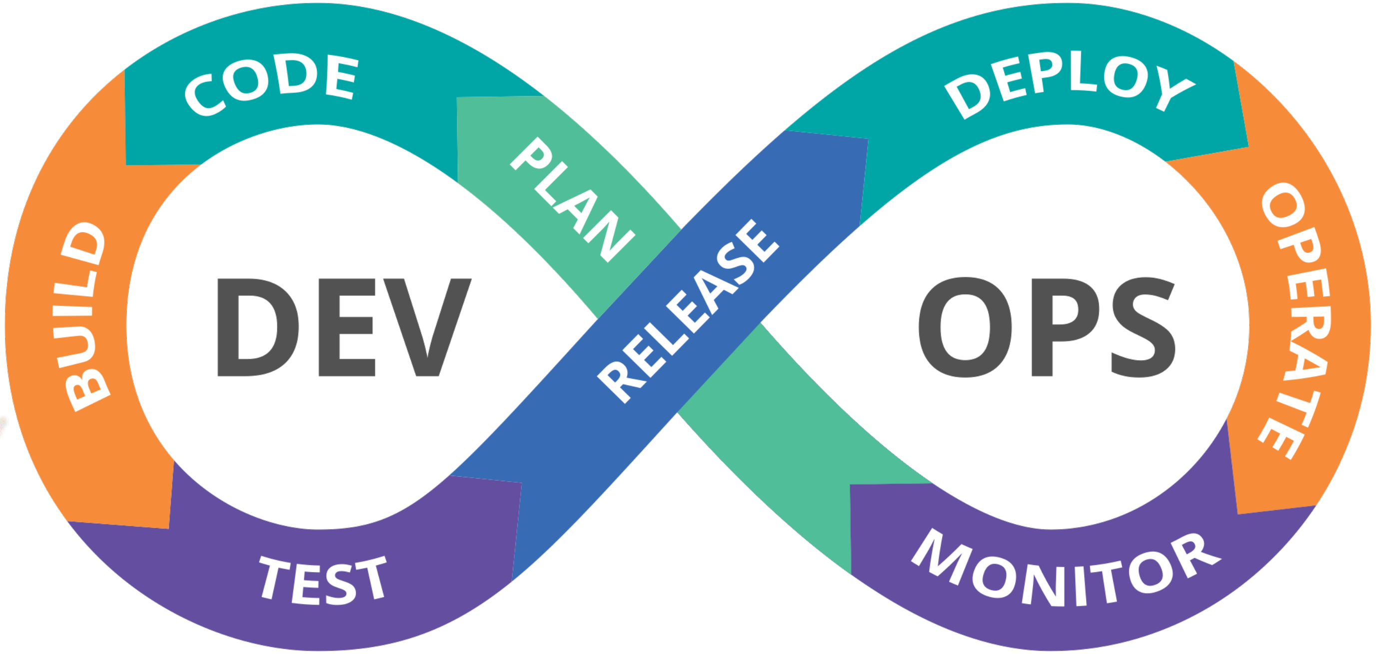
Jason D. Hiser
hiser@virginia.edu

Jack W. Davidson
jwd@virginia

Matthew Hicks
mdhicks2@vt.edu

The Fuzzing Landscape

Software Quality Assurance



Fuzzing (*Fuzz Testing*)

Automated, high-volume testing

1. Generate **lots of testcases**
2. Find, save, and mutate the ***few interesting*** testcases
3. Repeat!

Carpet-bombing testing approach

Fuzzing in the Real World

Coverage-guided *Grey-box* Fuzzing

- Today's *de-facto* bug-finding approach

GitLab

LLVM
COMPILER INFRASTRUCTURE

AFL++

Google: We've open-sourced ClusterFuzz tool that found 16,000 bugs in Chrome

New fuzzing tool finds 26 USB bugs in Linux, Windows, macOS, and FreeBSD

VIRGINIA TECH
UNIVERSITY of VIRGINIA

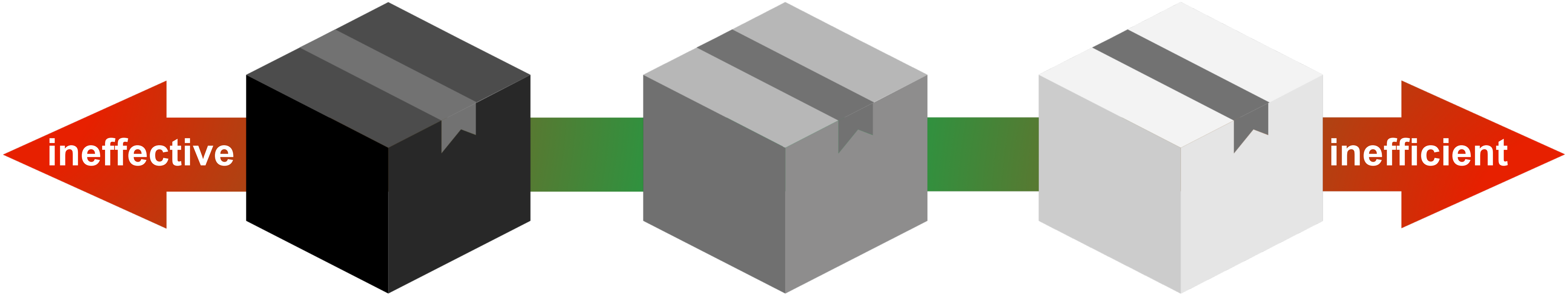
Grey-box Fuzzing

No internals
(basic I/O only)

Some internals
(e.g., code coverage)

Fast *and* effective

All internals
(developer-level)



Key requirement: **ability to instrument the target**

Target is **open-source**? Just ***compile-it-in***

When is instrumentation difficult?

When target is **binary-*only***

The Fuzzing Instrumentation Gap

Source-available Fuzzing

semantically **rich**

GCC

LLVM COMPILER INFRASTRUCTURE

Low (**18–32%**) overhead

Enhanced via **code xform**

Binary-only Fuzzing

semantically **opaque**

QEMU

Pin

DynamoRIO

Dyninst

intel

Up to **10,000%** slower

Outweighed by overhead

Can **compilers'** **capabilities** ***and*** **speed** be extended to **binary-only** fuzzing?

Compiler-quality Binary Fuzzing Instrumentation

Guiding Principle

What **instrumenter properties** must be achieved for ***compiler-quality*** speed **and** transformation?

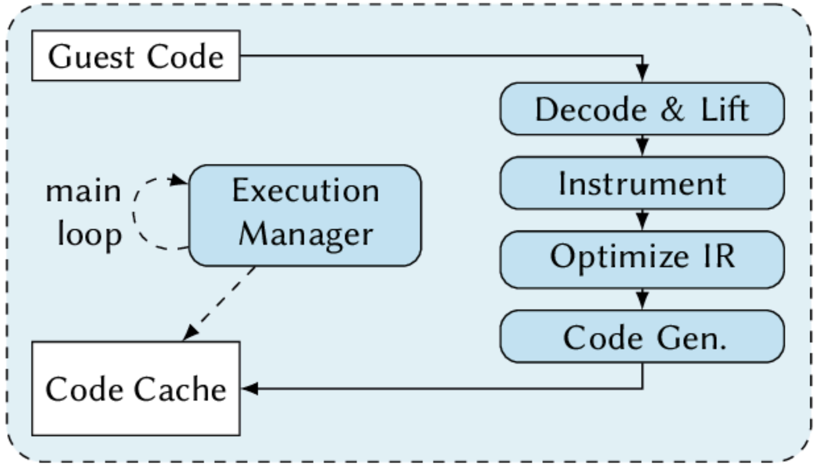
Key considerations:

- Code Insertion
- Code Invocation
- Register Usage
- Scalability

To attain **compiler-quality** instrumentation, we must ***match*** how compilers handle these key considerations

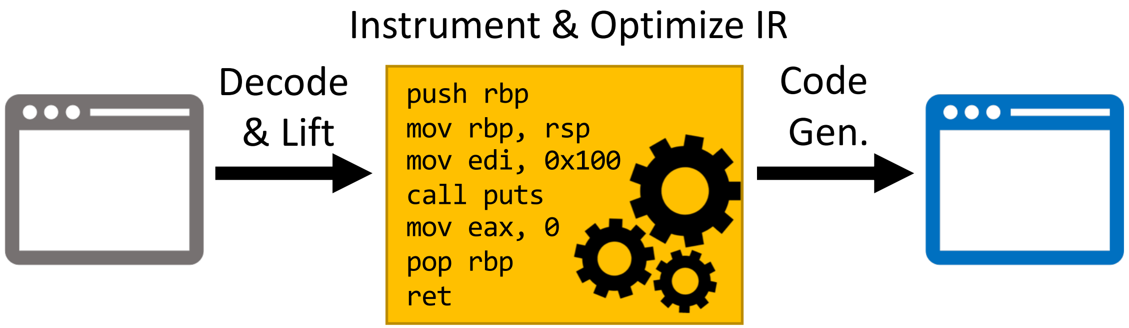
Consideration 1: Code Insertion

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)

Should insert code via **static rewriting**

Consideration 2: Code Invocation

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 **CF 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 most compilers

Should invoke code via **inlining**

DynamoRIO Pin QEMU ~~Dyninst~~ ~~intel~~

Consideration 3: Register Usage

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
```

- Reset **all regs** around instrumentation
- Cost of **saving and restoring** adds up

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 compilers' code optimization

Should carefully track **register liveness**

Consideration 4: Scalability

Common Platforms

- Linux x86-64
- Windows PE32+

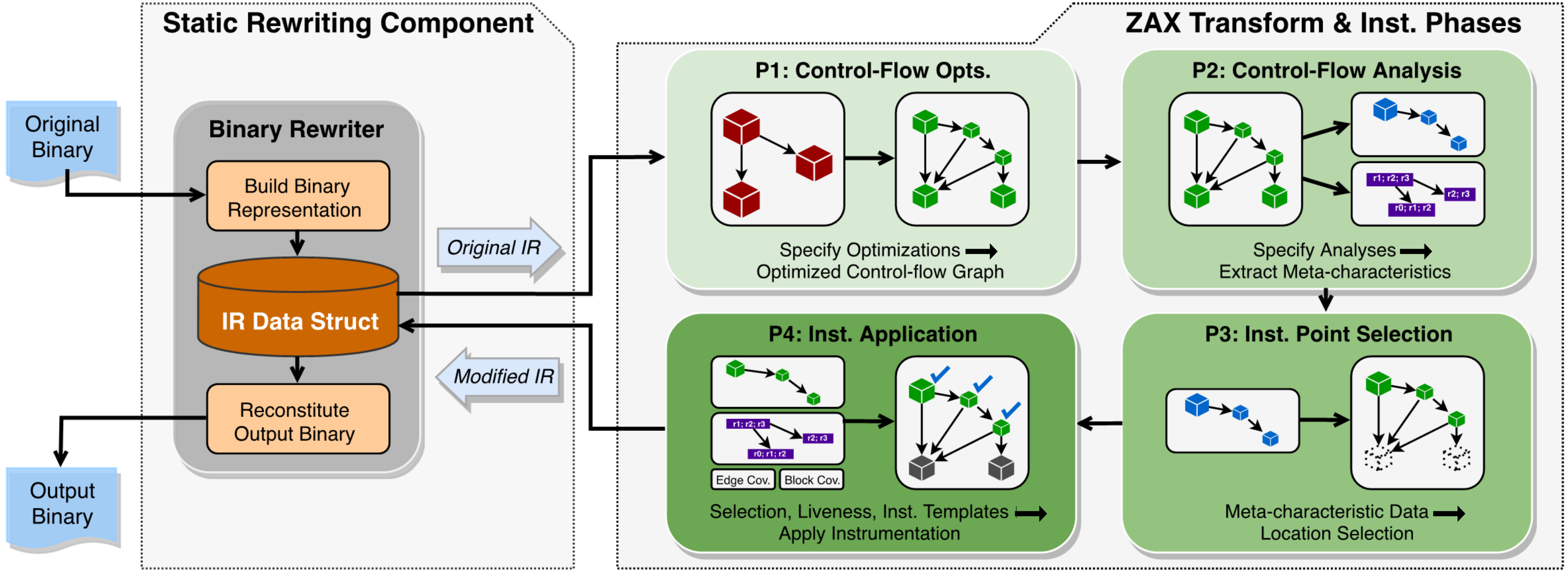
Common Characteristics

- C and C++
- PIE and non-PIE
- Stripped of debug symbols

Should scale to **all common formats**

The ZAFL Platform

- **Statically**-inserted, **inlined** instrumentation with **liveness awareness**
- Adapted from the Zipr binary rewriting project
- Support for **x86-64 ELF binaries** (and cross-platform support for **PE32+**)

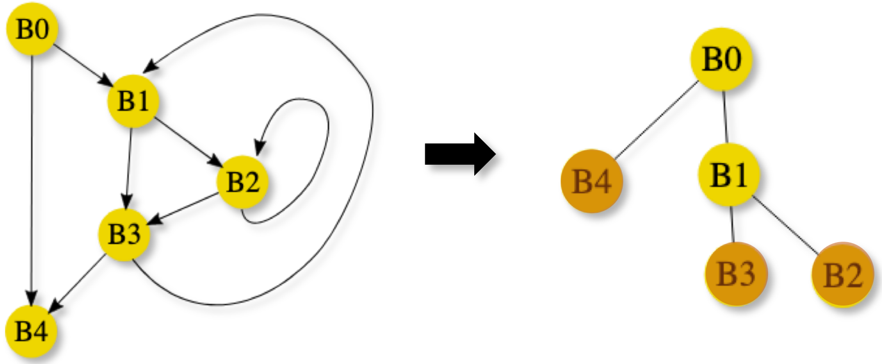


Extending Compiler-based Enhancements to Binary Fuzzing

Implement a **suite of 5** popular LLVM-based fuzzing transforms

Performance Transforms:

- Single-successor path pruning
- Dominator tree CFG pruning
- Instrumentation downgrading



Feedback Transforms:

- Sub-instruction profiling
- Context sensitivity tracking

```
if (var == 12345)
```

➡

```
if (var[0] == 1)
  if (var[1] == 2)
    if (var[2] == 3)
      if (var[3] == 4)
        if (var[4] == 5)
```

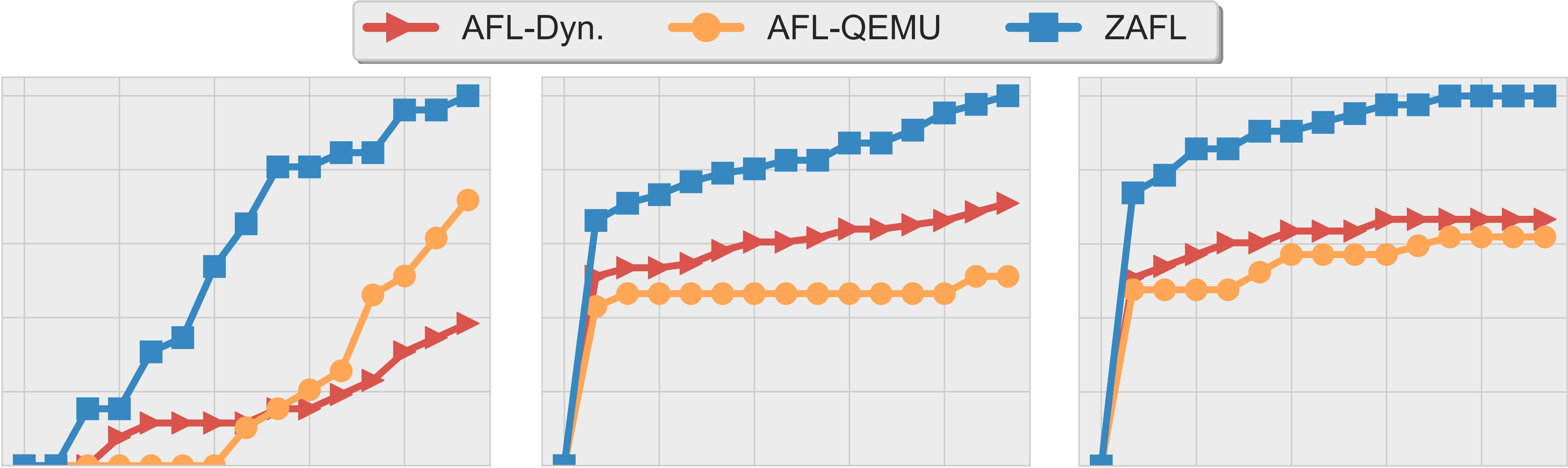
ZAFL's **low-level API** brings a **semantic richness** to the otherwise **semantically-opaque** world of binary fuzzing

Evaluation

Evaluation Components

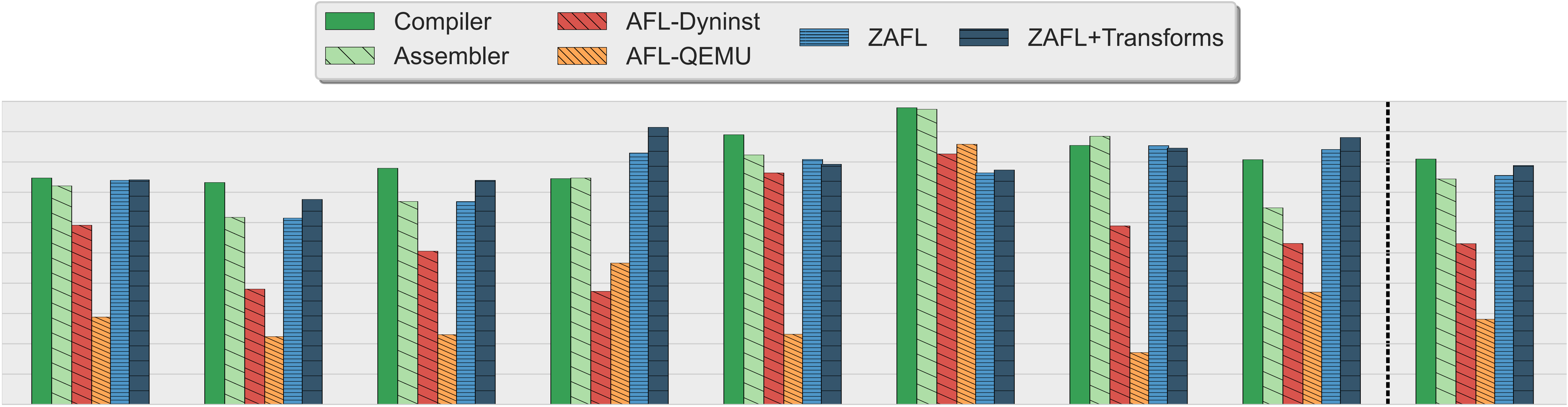
- **Benchmarks**: 8 diverse open-source + 5 closed-source binaries
- **Bug-finding**: 5x24-hr trials per benchmark run on cluster
- **Performance**: scale overhead relative to non-tracing speed
- **Precision**: enumerate erroneously-unrecovered instructions;
  compare true/false coverage signal to AFL-LLVM's
- **Scalability**: automated smoke tests and/or manual execution

Does ZAFL enhance binary fuzzing?



26% more crashes than AFL-Dyninst
131% more crashes than AFL-QEMU

Is ZAFL's speed near compilers'?



Compiler: **24%**, Assembler: **34%**
AFL-Dyninst: **88%**, AFL-QEMU: **256%**
ZAFL: **32%**, ZAFL+Transforms: **27%**

Can ZAFL support *real* closed-source?

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%**	

55% more crashes than AFL-Dyninst

38% more crashes than AFL-QEMU

Is ZAFL precise?

Binary	Total Insns	IDA Pro Unrecov	IDA Pro Reached	IDA Pro FalseNeg	Binary Ninja Unrecov	Binary Ninja Reached	Binary Ninja FalseNeg	ZAFL Unrecov	ZAFL Reached	ZAFL FalseNeg
idat64	268K	1681	0	0	5342	2	0	958	0	0
nconvert	458K	105K	3117	0.68%	3569	0	0	33.0K	0	0
nvdisasm	162K	180	0	0	3814	21.4	0.01%	0	0	0
pngout	16.8K	645	0	0	752	112.5	0.67%	1724	0	0
unrar	37.8K	1523	0	0	1941	138.2	0.37%	40	0	0

Highest overall instruction recovery

Mean coverage accuracy of **99.99%**

VIRGINIA TECH University of Virginia

Does ZAFL scale?

Apply ZAFL to **56 total binaries** (**33** open- and **23** closed-src)
Linux and **Windows** binaries
Stripped, **PIE**, and **non-PIE**
100KB–100MB binary size
100–1,000,000 basic blocks

Conclusions: Why ZAFL?

- Much of today's commodity software is distributed as **binary-only**
- Yet, **instrumenting**—and hence, **fuzzing**—it far less effective due to binary code's **semantic opaqueness**

> Mitigating these challenges demands closing fuzzing's *instrumentation gap*!

By carefully matching compilers' key attributes, **ZAFL** attains **compiler-quality speed** ***and*** fuzzing-enhancing **program transformation** for binary fuzzing:

- **Bug-finding:** **26—131%** superior to Dyninst/QEMU
- **Performance:** Within **10%** of LLVM's runtime speed
- **Scalability:** Linux and Windows, 10KB-100MB filesizes, 100-1M basic blocks, and other characteristics

Thank you!

Find ZAFL and our evaluation benchmarks at:

`git.zephyr-software.com/opensrc/zafl`

Happy (*binary*) fuzzing!

Appendix: **The Binary Fuzzing Instrumentation Landscape**

Code Insertion	Static Rewriting	~~DynamoRIO~~ ~~Pin~~ ~~QEMU~~ Dyninst ~~intel~~
Code Invocation	Inlined Invocation	DynamoRIO Pin QEMU ~~Dyninst~~ ~~intel~~
Register Usage	Liveness Aware	DynamoRIO Pin QEMU ~~Dyninst~~ ~~intel~~
Scalability	Support Broad Formats	DynamoRIO Pin QEMU ~~Dyninst~~ intel

Until **all four** properties are met, the gap between **source-** and **binary-level** fuzzing will remain