Breaking Through Binaries

Compiler-quality Instrumentation for Better Binary-only Fuzzing

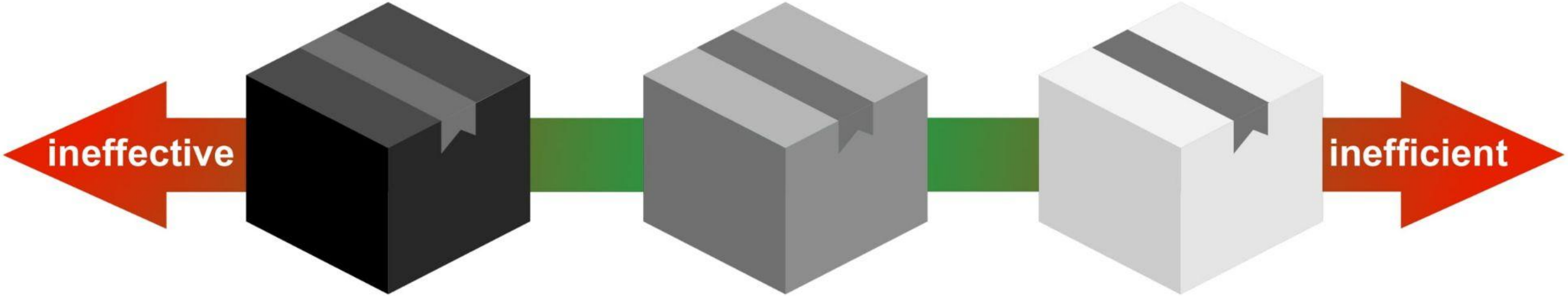
Grey-box Fuzzing

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***

Coverage Guide Fuzzing

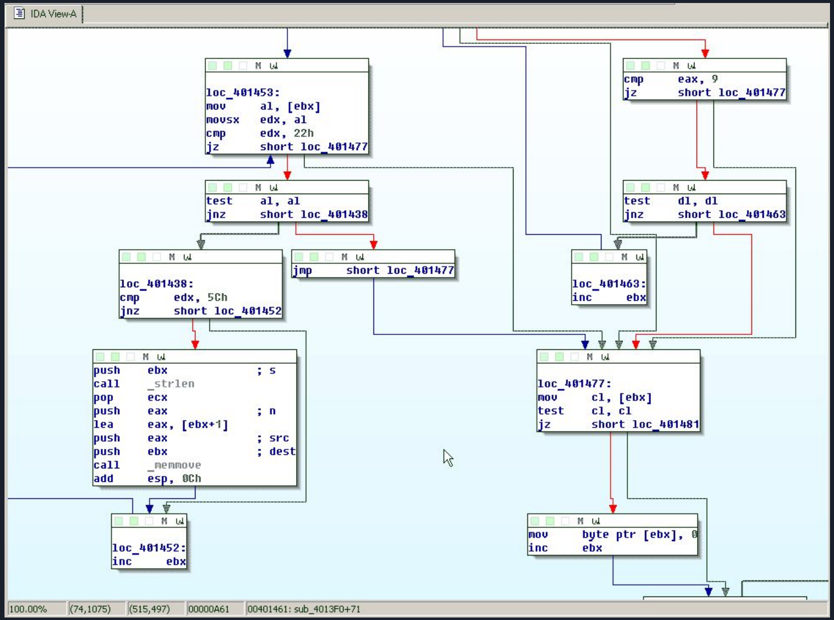
Use feedback to direct our fuzzing inputs, with the goal of providing more “coverage”. Coverage can be defined in multiple ways, e.g. line coverage, function coverage, block coverage, etc. In greybox, coverage usually defined as blocks.

Fuzzing inputs are “mutated” based on coverage feedback.

Directory	Line Coverage			Functions	
contrib/adminpack		45.7 %	79 / 173	70.8 %	17 / 24
contrib/amcheck		64.2 %	786 / 1225	91.8 %	45 / 49
contrib/auth_delay		0.0 %	0 / 13	0.0 %	0 / 3
contrib/auto_explain		89.1 %	90 / 101	100.0 %	6 / 6
contrib/basebackup_to_shell		85.3 %	99 / 116	100.0 %	14 / 14
contrib/basic_archive		40.2 %	39 / 97	87.5 %	7 / 8
contrib/bloom		90.3 %	467 / 517	96.7 %	29 / 30
contrib/bool_plperl		100.0 %	11 / 11	100.0 %	5 / 5

Blocks (Aka IDAs Graph View)

Blocks are chunks of instructions that are typically separated by branches or function calls.

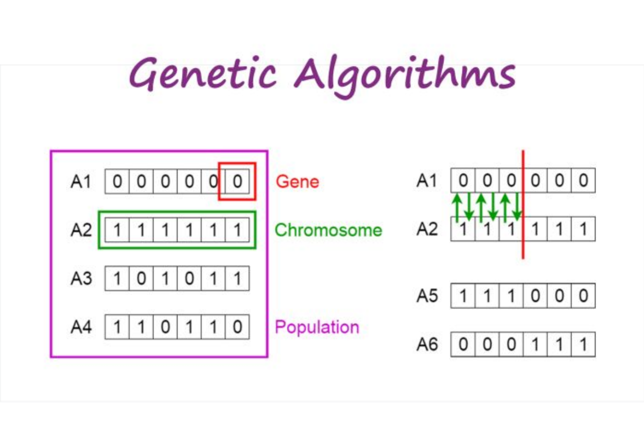


Mutations (Genetic Algorithms)

When we talk about mutations to fuzzing inputs, we are typically referring to genetic algorithms. Genetic algorithms are part of a class of heuristic algorithms (e.g. simulated annealing, tabu search, neighborhood search, etc.) Genetic algorithms are a class of global search algorithms.

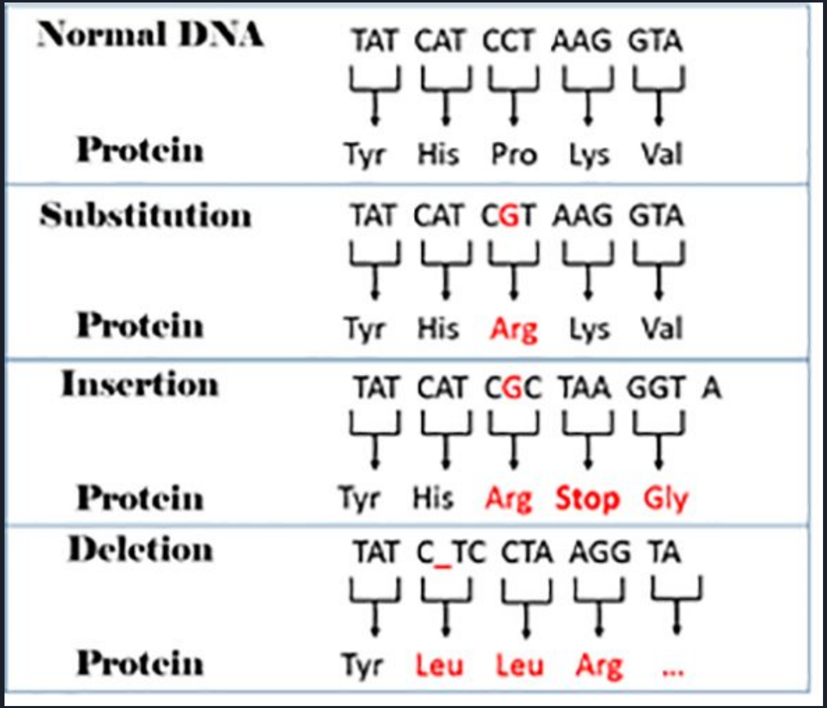
Genetic algorithms typically follow a simple flow of operations.

1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation
6. Repeat 2-4



Types of Mutations

- Substitution
- Insertion
- Deletion
- Deletion-Insertion
- Duplication
- Inversion



Instrumentation (Get That Feedback Yo)

Whitebox

- Insert coverage accounting directly into source.

Grey/Blackbox

- Intel PT/ARM CoreSight (hardware)
- Dynamic Binary Translation
- Static Rewriting

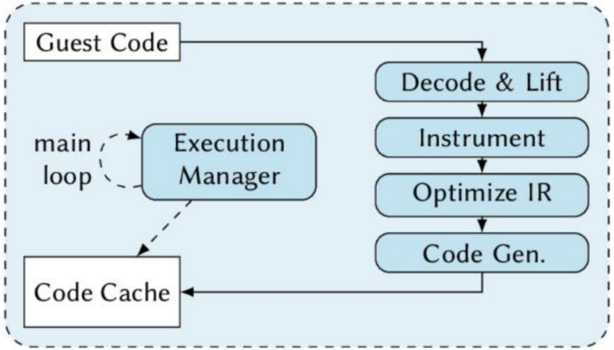
The Fuzzing Instrumentation Gap

Source-available Fuzzing	Binary-only Fuzzing
semantically **rich**	semantically **opaque**
GCC, LLVM COMPILER INFRASTRUCTURE	QEMU, Pin, DynamoRIO, Dyninst, intel
Low (**18–32%**) overhead	Up to **10,000%** slower
Enhanced via **code xform**	**Outweighed** by overhead

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

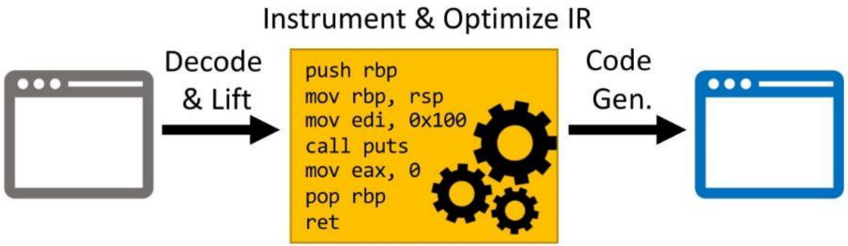
Consideration 1: Code Insertion

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~~

VIRGINIA TECH UNIVERSITY of VIRGINIA

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**

Compiler-based Fuzzing Enhancements

Compiler-based fuzzing has an advantage of higher level semantics are overall view of the system. This allows compiler-based fuzzing to implement a few enhancements.

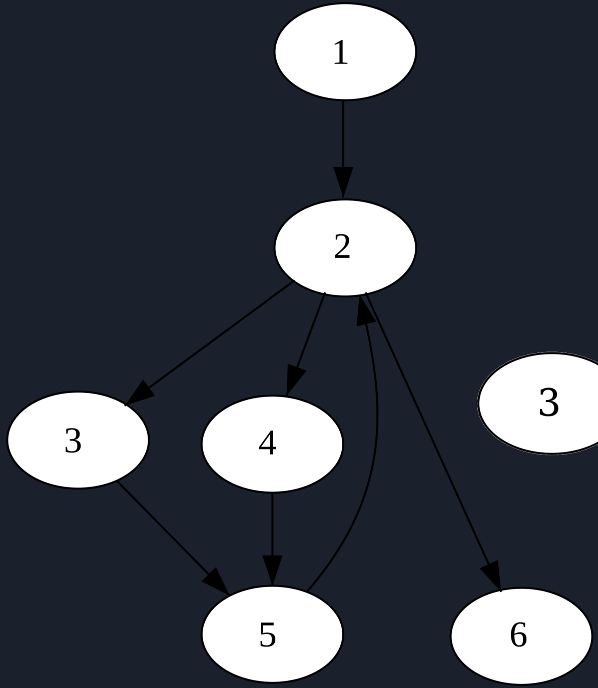
- Instrumentation Pruning
- Instrumentation Downgrading
- Sub-instruction Profiling
- Extras

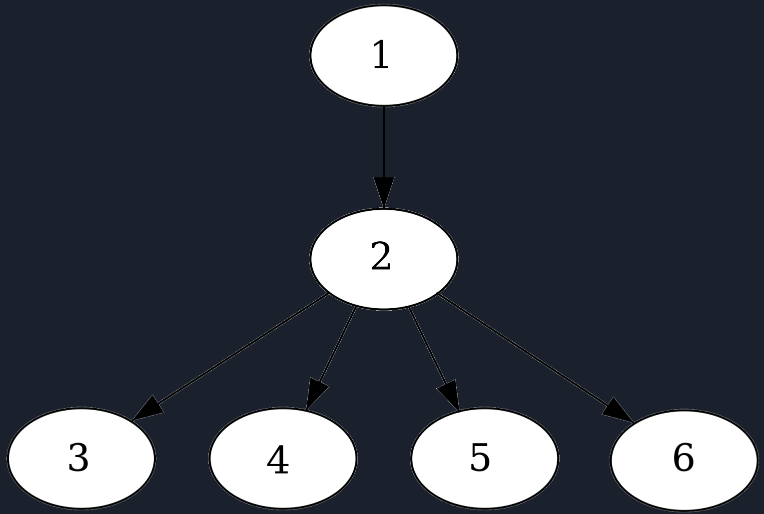
Instrumentation Pruning

Tracing coverage is practically constructing a control flow graph. We can then reduce the graph to reduce the amount of blocks to instrument.

If all possible paths to *B* include contains *A*, then block *A* dominates block *B*.

Remove all but unique block paths for instrumentation.





Instrumentation Downgrading

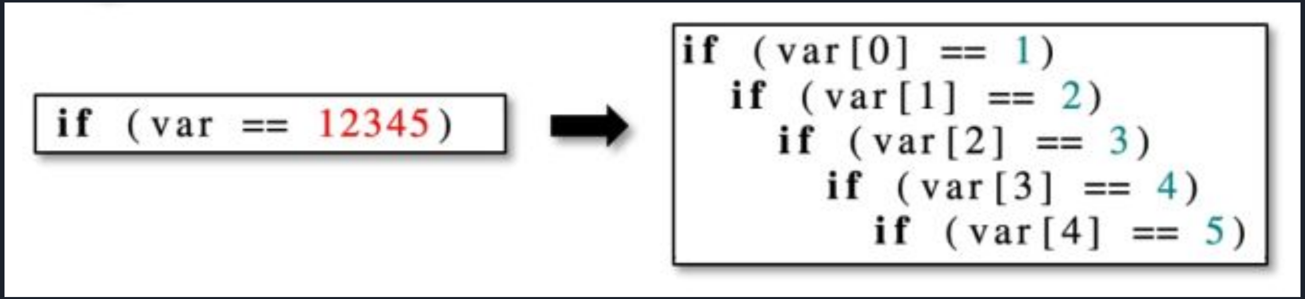
Majority of fuzzers track coverage using edges, typical recorded as hashes of starting and ending blocks.

Some compiler-based fuzzers reduce accounting of single predecessor blocks to only count the final block.

Sub-instruction Profiling

A graph view of blocks doesn’t provide insight into things like checksums or magic byte checks. It will view them as several blocks.

Sub-instruction Profiling can transpose such multi-byte comparison (one block) into nested single by comparisons (multiple blocks).



Fundamental Design Considerations

- Rewriting vs Translation
- Inlining vs Trampolining
- Register Allocation (Register Liveness Tracking)
- Scalability

ZAFL

The author applies compiler-based fuzzing enhancements by rewriting binaries, thus achieving near compiler level speeds for greybox fuzzing.

1. IR Extraction
2. ZAX (applying compiler-based fuzzing enhancements)
    a. Optimization
    b. Analysis
    c. Point Selection
    d. Application
3. Binary Reconsitution

ZAFL Improvements

Performance-enhancing Transformations:

- Single Successor Instrumentation Pruning
- Dominator Tree Pruning
- Edge Instrumentation Downgrading

Feedback-enhancing Transformations

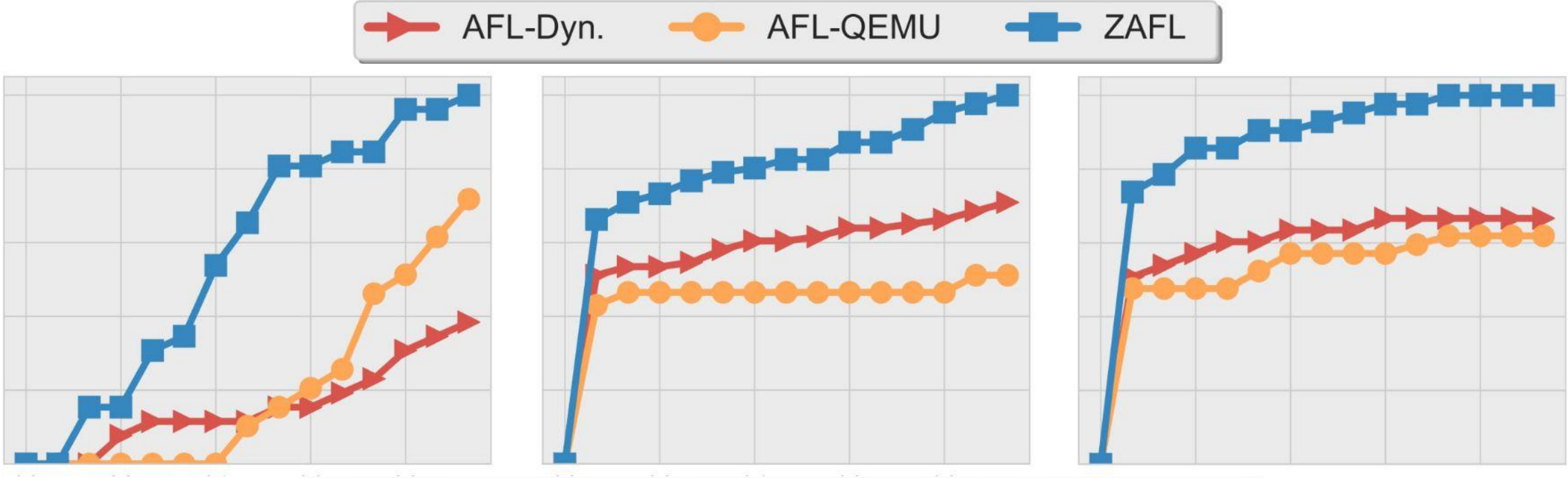
- Sub-instruction profiling
- Context-sensitive coverage

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

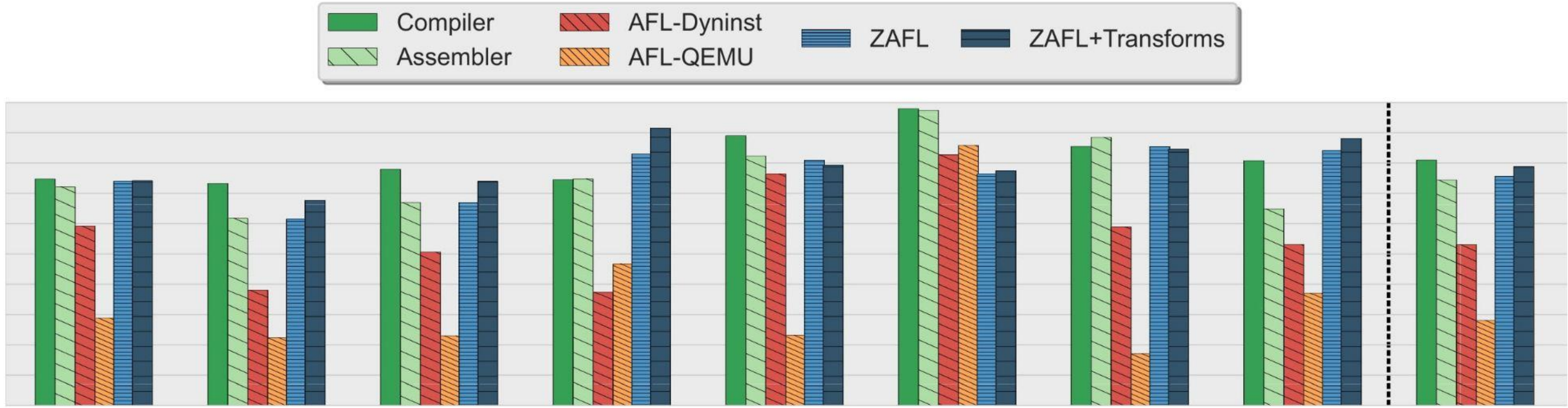
Does ZAFL enhance binary fuzzing?



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

VIRGINIA TECH
UNIVERSITY of VIRGINIA

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			Binary Ninja			ZAFL		
		Unrecov	Reached	FalseNeg	Unrecov	Reached	FalseNeg	Unrecov	Reached	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

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

VIRGINIA TECH University of Virginia

Hybrid Fuzzing

- Taint tracking (Angora, REDQUEEN)
- Concolic Execution (Driller)
- Parallelization