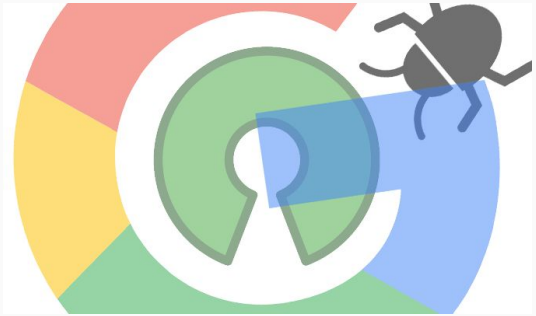
Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities

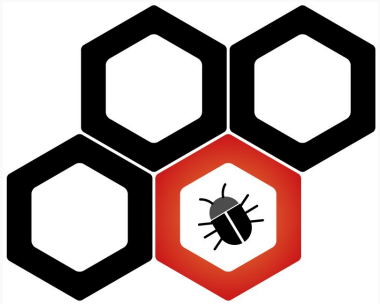
Manh-Dung Nguyen, Sébastien Bardin, Matthieu Lemerre (CEA LIST)
Richard Bonichon (Tweag I/O)
Roland Groz (Université Grenoble Alpes)

Fuzzing



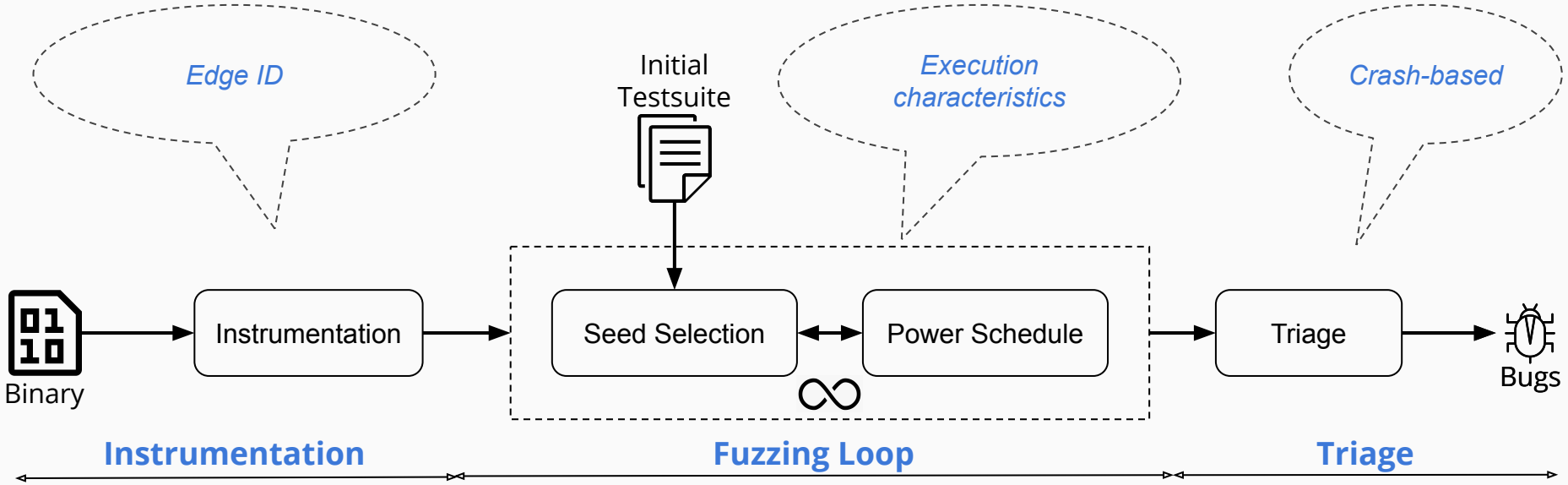
September 15, 2020

Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale

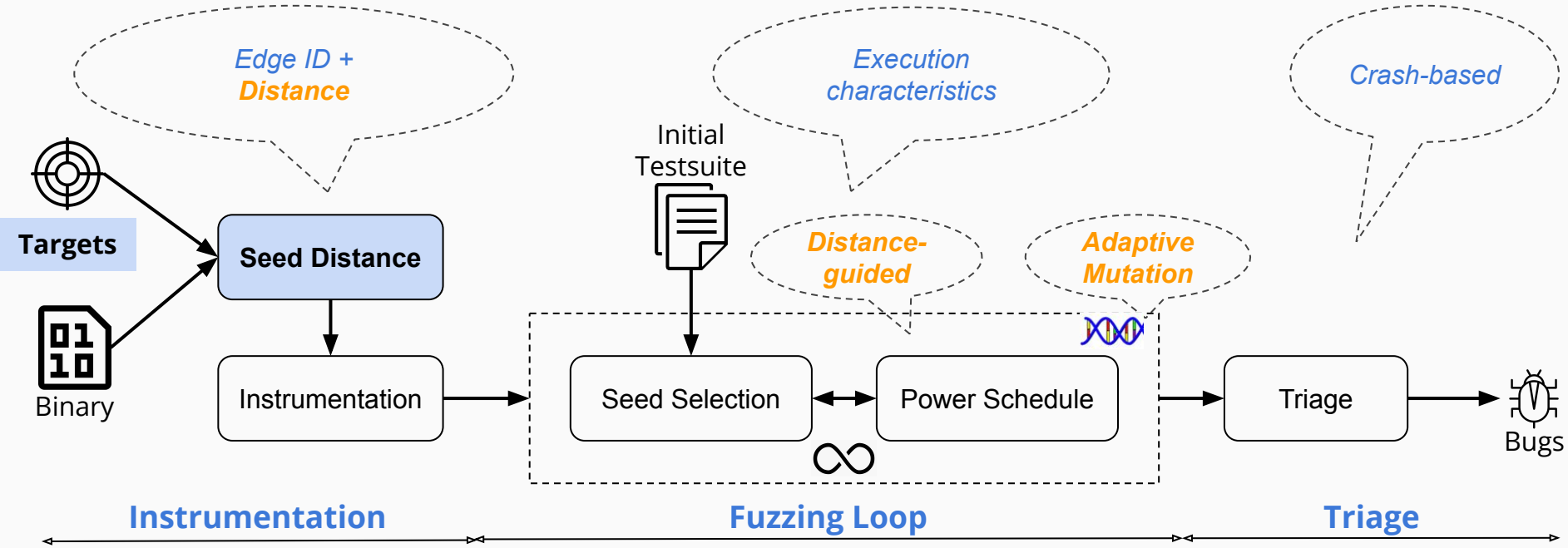


FUZZING/IO
Security Automation • Vulnerability Research

Coverage-guided Greybox Fuzzing AFL, libFuzzer



Directed Greybox Fuzzing AFLGo, Hawkeye

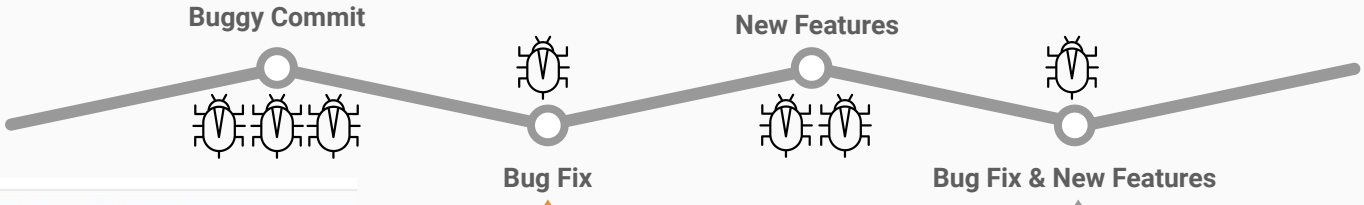


Applications of Directed Fuzzing (DGF)

Only 55% bugs reports are reproducible

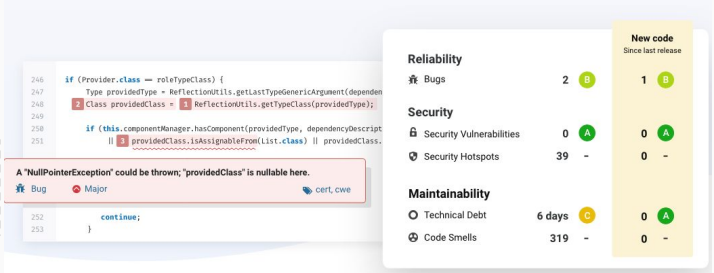
(1) Bug Reproduction
(lack of PoC, # env)

(2) Patch Testing
(vulnerable code)



(2) Patch Testing
(incomplete patch)

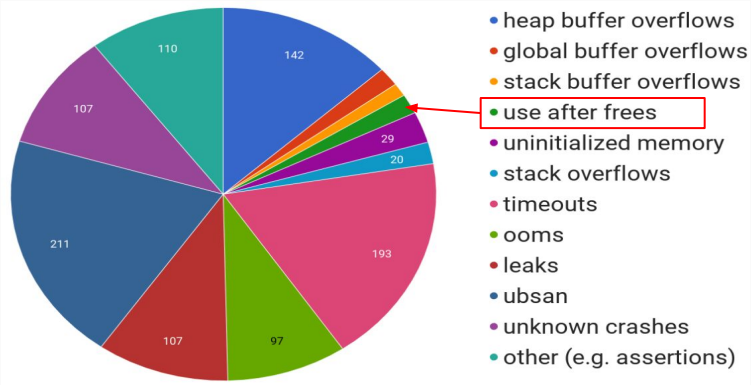
(3) Static Report Verification
(provide PoC inputs)



Focus on (1) Bug Reproduction & (2) Patch Testing

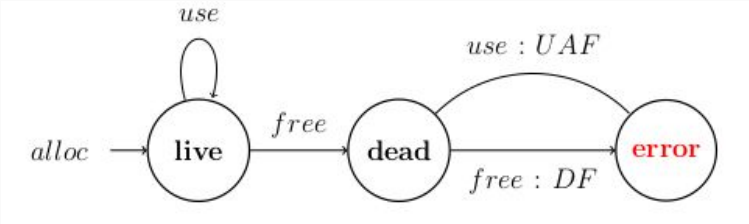
Why is Detecting UAF Hard ?

- Rarely found by fuzzers
  - *Complexity*: 3 events *in sequence* spanning multiple functions
  - *Temporal & Spatial constraints*: extremely difficult to meet in practice
  - *Silence*: no segmentation fault

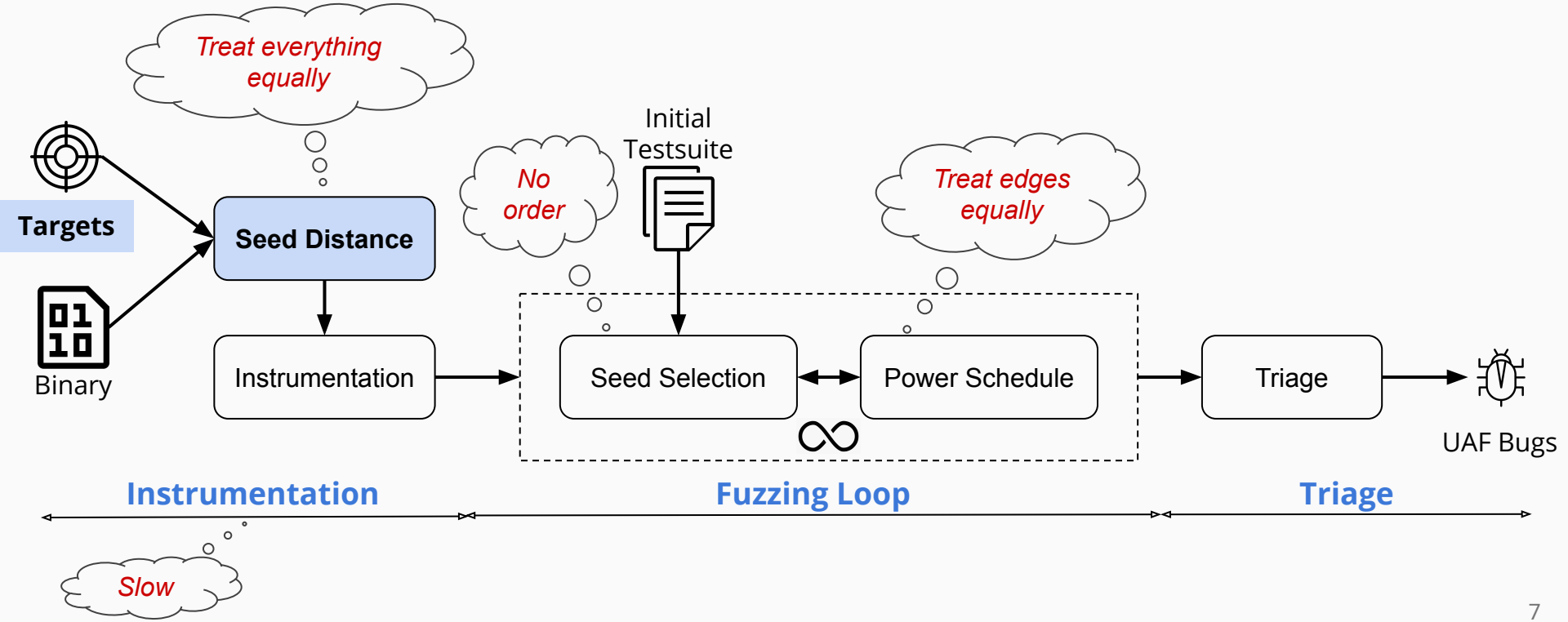


*# UAF bugs found (**1%**) by OSS-Fuzz in 2017*

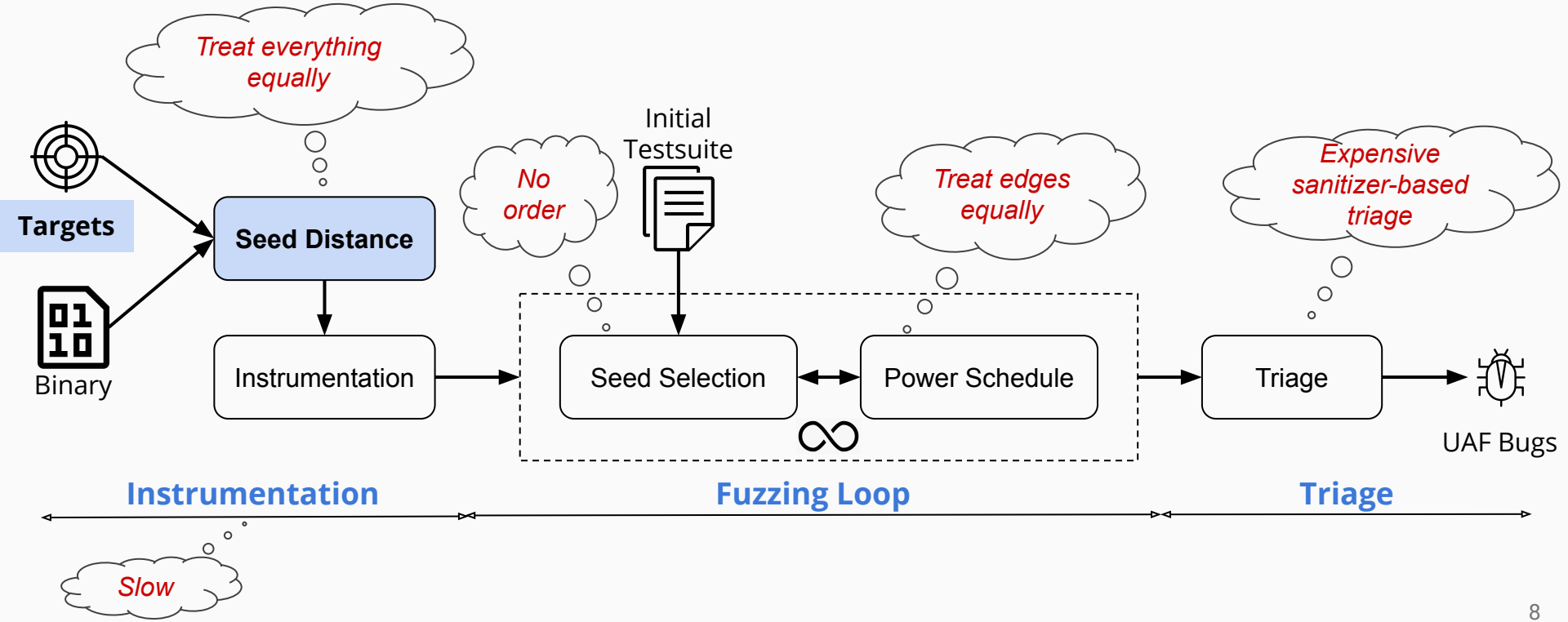
```
1 char *buf = (char *) malloc(BUF_SIZE);
2 free(buf); // pointer buf becomes dangling
3 ...
4 strncpy(buf, argv[1], BUF_SIZE-1); // Use-After-Free
```



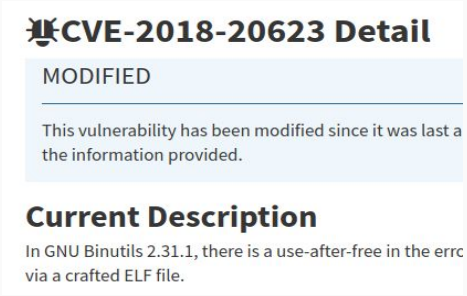
Existing DGF: #1 No Ordering & No Prioritization



Existing DGF: #2 Crash Assumption



UAF Stack Traces



CVE-2018-20623 Detail

MODIFIED

This vulnerability has been modified since it was last a
the information provided.

Current Description

In GNU Binutils 2.31.1, there is a use-after-free in the erro
via a crafted ELF file.

// stack trace for the bad Use

==4440== Invalid read of size 1
==4440== at 0x40A8383: vfprintf (vfprintf.c:1632)
==4440== by 0x40A8670: buffered_vfprintf (vfprintf.c:2320)
==4440== by 0x40A62D0: vfprintf (vfprintf.c:1293)
[6] ==4440== by 0x80AA58A: error (elfcomm.c:43)
[5] ==4440== by 0x8085384: process_archive (readelf.c:19063)
[1] ==4440== by 0x8085A57: process_file (readelf.c:19242)
[0] ==4440== by 0x8085C6E: main (readelf.c:19318)

// stack trace for the Free

==4440== Address 0x421fdc8 is 0 bytes inside a block of size 86 free'd
==4440== at 0x402D358: free (in vgpreload_memcheck-x86-linux.so)
[4] ==4440== by 0x80857B4: process_archive (readelf.c:19178)
[1] ==4440== by 0x8085A57: **process_file** (readelf.c:19242)
[0] ==4440== by **0x8085C6E: main** (readelf.c:19318)

// stack trace for the Alloc

==4440== Block was alloc'd at
==4440== at 0x402C17C: malloc (in vgpreload_memcheck-x86-linux.so)
[3] ==4440== by 0x80AC687: make_qualified_name (elfcomm.c:906)
[2] ==4440== by 0x80854BD: process_archive (readelf.c:19089)
[1] ==4440== by 0x8085A57: process_file (readelf.c:19242)
[0] ==4440== by 0x8085C6E: main (readelf.c:19318)

*At **0x8085C6E** in **main()**, there is a call to **process_file()***

*Target location: (**0x8085C6E**, **main**)*

UAF Bug Target

Bug Trace of CVE-2018-20623

// stack trace for the bad Use

==4440== Invalid read of size 1
==4440== at 0x40A8383: vfprintf (vfprintf.c:1632)
==4440== by 0x40A8670: buffered_vfprintf (vfprintf.c:2320)
==4440== by 0x40A62D0: vfprintf (vfprintf.c:1293)
[6] ==4440== by 0x80AA58A: error (elfcomm.c:43)
[5] ==4440== by 0x8085384: process_archive (readelf.c:19063)
[1] ==4440== by 0x8085A57: process_file (readelf.c:19242)
[0] ==4440== by 0x8085C6E: main (readelf.c:19318)

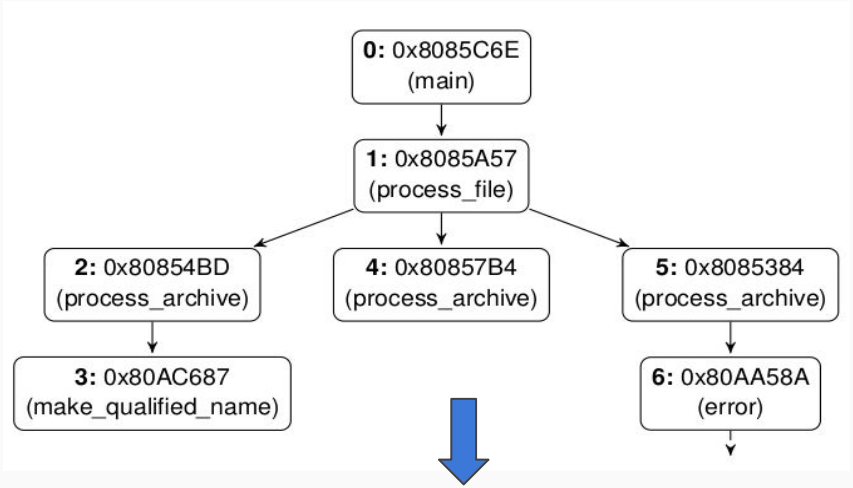
// stack trace for the Free

==4440== Address 0x421fdc8 is 0 bytes inside a block of size 86 free'd
==4440== at 0x402D358: free (in vgpreload_memcheck-x86-linux.so)
[4] ==4440== by 0x80857B4: process_archive (readelf.c:19178)
[1] ==4440== by 0x8085A57: process_file (readelf.c:19242)
[0] ==4440== by 0x8085C6E: main (readelf.c:19318)

// stack trace for the Alloc

==4440== Block was alloc'd at
==4440== at 0x402C17C: malloc (in vgpreload_memcheck-x86-linux.so)
[3] ==4440== by 0x80AC687: make_qualified_name (elfcomm.c:906)
[2] ==4440== by 0x80854BD: process_archive (readelf.c:19089)
[1] ==4440== by 0x8085A57: process_file (readelf.c:19242)
[0] ==4440== by 0x8085C6E: main (readelf.c:19318)

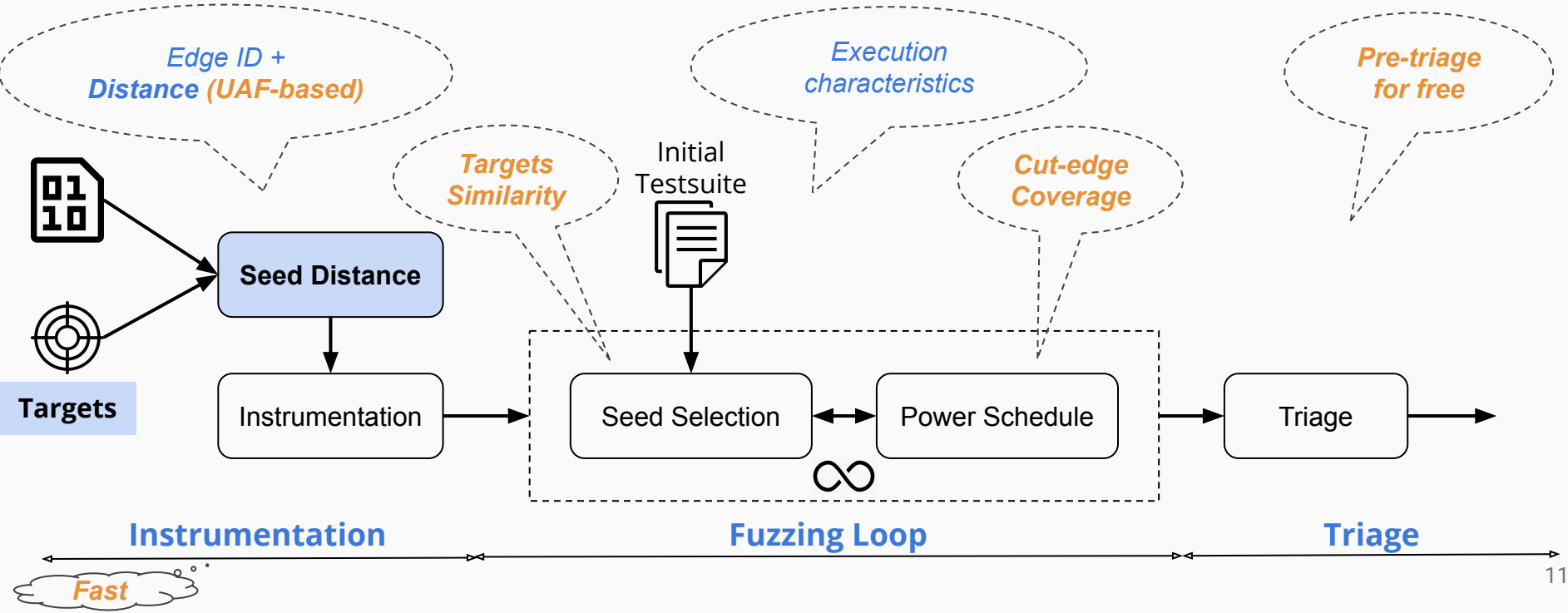
Dynamic Calling Tree



Bug Trace Flattening

UAF Bug Target:
0 *(0x8085C6E, main)* → ***1*** *(0x8085A57, process_file)* → ***2*** *(0x80854BD, process_archive)* → ***3*** *(0x80AC687, make_qualified_name)* → ***4*** *(0x80857B4, process_archive)* → ***5*** *(0x8085384, process_archive)* → ***6*** *(0x80AA58A, error)*

Overview of UAFuzz



Key Insights of UAFuzz

- ★ Seed Selection: based on similarity and ordering of input trace
- ★ Power Schedule: based on 3 seed metrics dedicated to UAF
  - [function level] UAF-based Distance: Prioritize call traces covering UAF events
  - [edge level] Cut-edge Coverage: Cover edge destinations reaching targets
  - [basic block level] Target Similarity: Cover targets

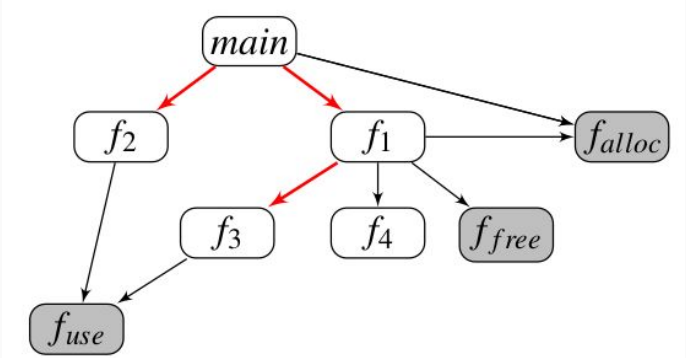
- ★ Triage only potential inputs covering all locations & pre-filter for free

- ★ Fast precomputation at binary-level

UAF-based Distance

- Existing works compute seed distance
  - *regardless of target ordering*
  - *regardless of UAF characteristic: call traces may contain in sequence alloc/free function and reach use function*

- Intuition: UAFuzz favors the shortest path that is likely to cover more than 2 UAF events in sequence
  - Statically identify and decrease weights of (caller, callee) in Call Graph
  - Ex: favored call traces <*main*, f_2, f_{use}>, <*main*, f_1, f_3, f_{use}>

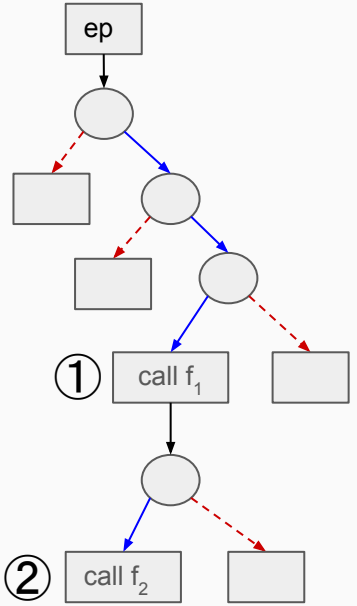


Example of Call Graph, favored pairs (caller, callee) are in red

Cut-edge Coverage Metric

- Existing works *treat edges equally* in terms of reaching in sequence targets

- Cut-edge
  - Edge destinations are more likely to reach the next target in the bug trace
  - Approximately identify via static intraprocedural analysis of CFGs
- Intuition: UAFuzz favors inputs exercising more cut edges via a score depending on # covered cut edges and their hit counts



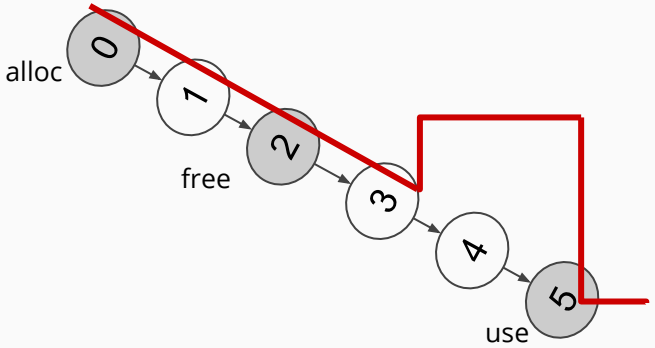
Control Flow Graph, cut edges are in blue

Target Similarity Metric

- Existing works select seeds to be mutated *regardless of number of covered target locations*

- Target Similarity Metric
  - Prefix: more precise
  - Bag: less precise, but consider the whole trace
- Intuition: Seed Selection heuristic based on both prefix and bag metrics
  - Select more frequently max-reaching inputs that have highest value of this metric (most similar to the bug trace) so far

trace of input s: *0 → 1 → 2 → 3 → 7 → 8 → 5*



Bug Trace : 0 (alloc) → 1 → 2 (free) → 3 → 4 → 5 (use)

Power Schedule

Intuition: UAFuzz assigns more energy (a.k.a, # mutants) to

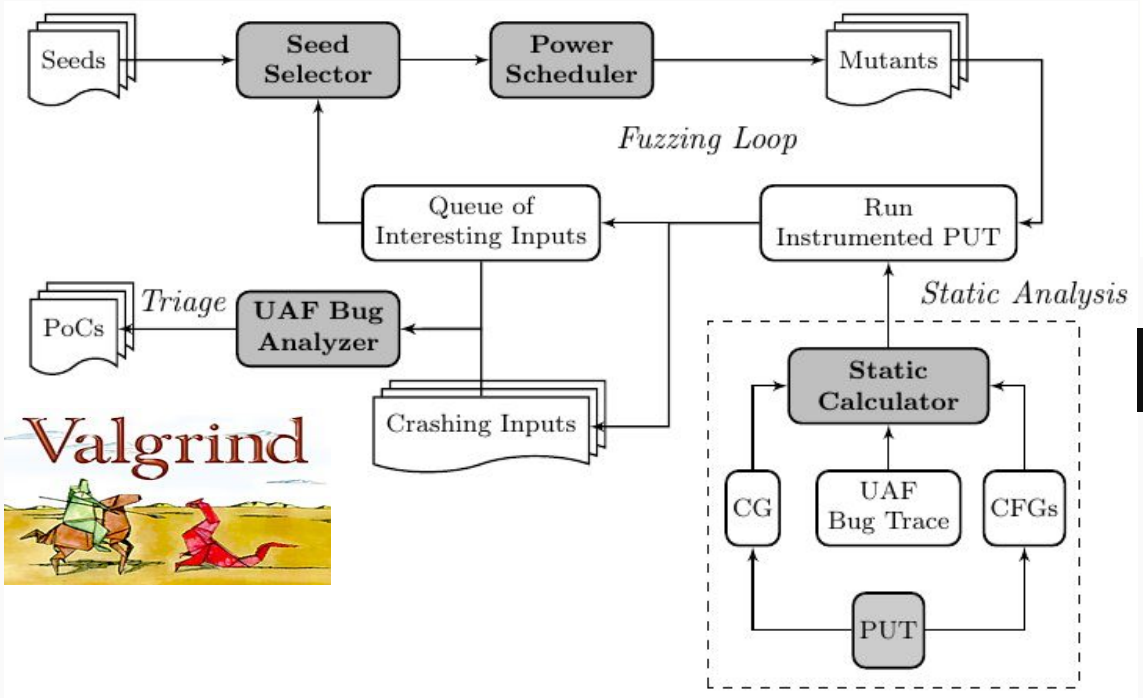
- seeds that are closer (using *UAF-based Distance*)
- seeds that are more similar to the bug trace (using *Target Similarity Metric*)
- seeds that make better decisions at critical code junctions (using *Cut-edge Coverage Metric*)

Pre-filter

- Existing work simply send *all* fuzzed inputs to the bug triager

- Potential inputs: cover in sequence all target locations in the bug trace
- UAFuzz triages only potential inputs & safely discards others
  - Available for free after the fuzzing process via Target Similarity Metric
  - Saving a huge amount of time in bug triaging

Implementation



Experimental Evaluation

- Bug Reproduction
  - Time-to-Exposure, # bugs found, overhead, # triaging inputs
- Patch-Oriented Testing

- Evaluated fuzzers
  - UAFuzz (BINSEC & AFL-QEMU)
  - AFL-QEMU
  - AFLGo (source - level, co-author)
  - Our implementations AFLGoB & HawkeyeB

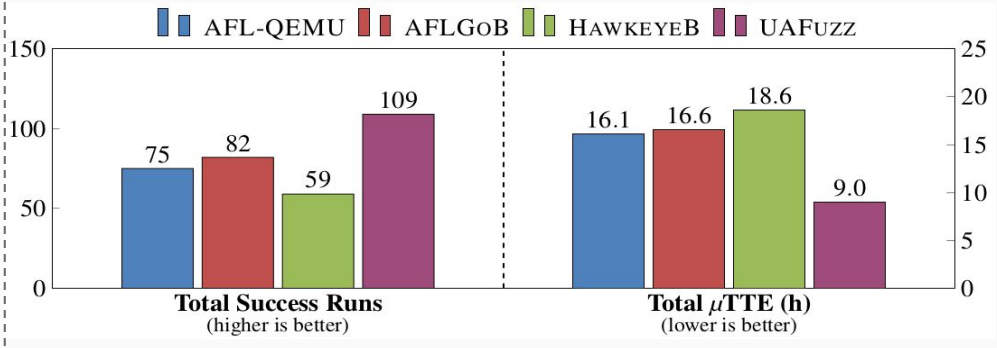
Our UAF Fuzzing Benchmark

Bug ID	Program		Bug		#Targets in trace
	Project	Size	Type	Crash	
giflib-bug-74	GIFLIB	59 Kb	DF	✗	7
CVE-2018-11496	lrzip	581 Kb	UAF	✗	12
yasm-issue-91	yasm	1.4 Mb	UAF	✗	19
CVE-2016-4487	Binutils	3.8 Mb	UAF	✓	7
CVE-2018-11416	jpegoptim	62 Kb	DF	✗	5
mjs-issue-78	mjs	255 Kb	UAF	✗	19
mjs-issue-73	mjs	254 Kb	UAF	✗	28
CVE-2018-10685	lrzip	576 Kb	UAF	✗	7
CVE-2019-6455	Recutils	604 Kb	DF	✗	15
CVE-2017-10686	NASM	1.8 Mb	UAF	✓	10
gifsicle-issue-122	Gifsicle	374 Kb	DF	✗	11
CVE-2016-3189	bzip2	26 Kb	UAF	✓	5
CVE-2016-20623	Binutils	1.0 Mb	UAF	✗	7

Bug Reproduction: Fuzzing Performance

RQ1: Bug-reproducing Ability (1)

- *Total success runs* vs. 2nd best AFLGoB: **+34%** in total, up to +300%
- *Time-to-Exposure (TTE)* vs. 2nd best AFLGoB: **2.0x**, avg 6.7x, max 43x
- *Vargha-Delaney metric* vs. 2nd best AFLGoB: avg **0.78**

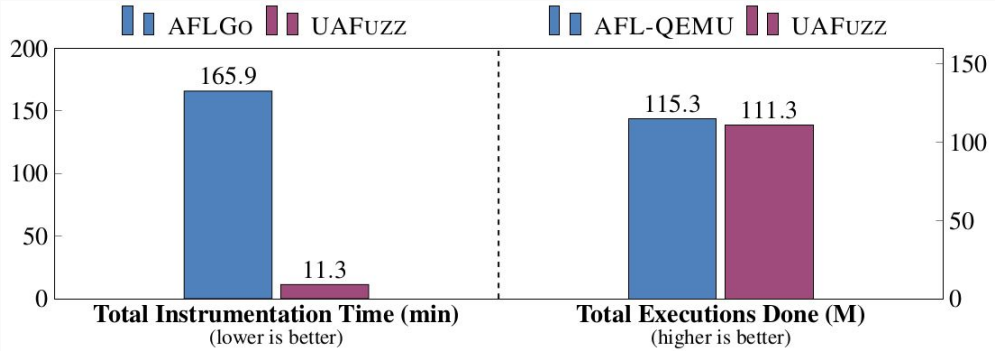


Bug-reproducing performance of binary-based DGFs

UAFuzz *significantly outperforms* state-of-the-art directed fuzzers in terms of UAF bugs reproduction with a *high confidence level*

Bug Reproduction: Overhead

- *Instrumentation overhead*
  - **15x** faster in total than AFLGo-source
- *Runtime overhead*
  - UAFuzz has the same total executions done compared to AFL-QEMU

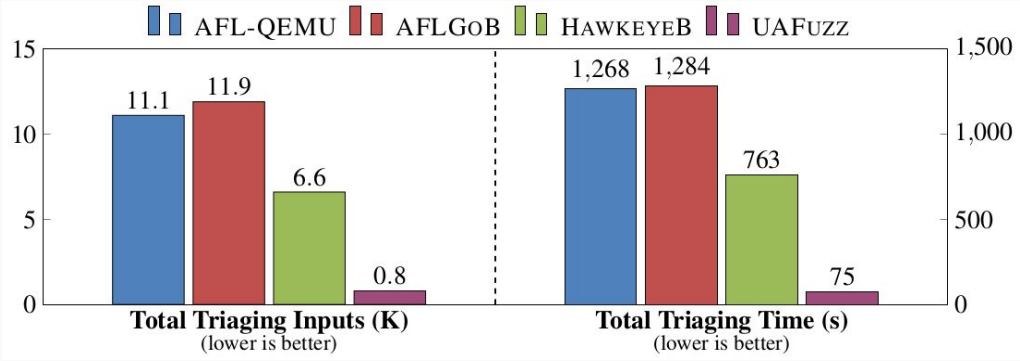


Global Overhead

UAFUZZ enjoys both a *lightweight instrumentation time* and a *minimal runtime overhead*

Bug Reproduction: Triage

- *Total triaging inputs*
  - UAFuzz only triages *potential* inputs (**9.2%** in total – sparing up to **99.76%** of input seeds for confirmation)
- *Total triaging time*
  - UAFuzz only spends several seconds (avg **6s**; **17x** over AFLGoB, max 130x)



Bug Triaging Performance

UAFuzz reduces a *large* portion (i.e., more than *90%*) of triaging inputs in the post-processing phase

Patch Testing

How to find

- Identify recently discovered UAF bugs
- Manually extract call instructions in bug traces
- Guide the directed fuzzer on the patch code

Targets

- Incomplete patches, regression bugs
- Weak parts of code

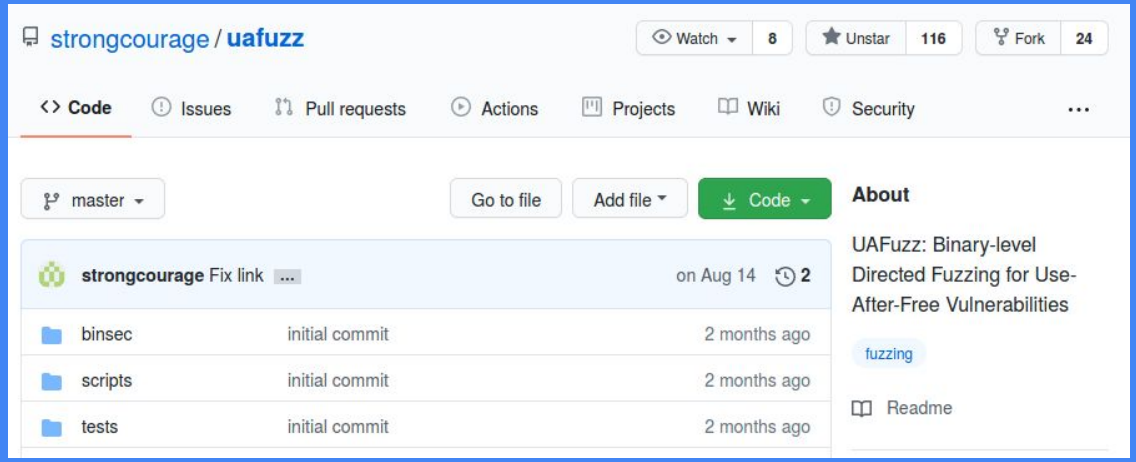
UAFuzz has been proven *effective in a patch-oriented* setting, allowing to find *30 new bugs (4 incomplete patches, 7 CVEs)* in 6 open-source programs

Thank you ! Q & A

~~~

UAFuzz: https://github.com/strongcourage/uafuzz

UAF Fuzzing Benchmark: https://github.com/strongcourage/uafbench


~~~