black hat USA 2020
AUGUST 5-6, 2020
BRIEFINGS

About Directed Fuzzing and Use-After-Free: How to Find Complex & Silent Bugs?

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

Who Are We?

Sébastien Bardin

sebastien.bardin@cea.fr

Senior Researcher at CEA LIST

Université Paris-Saclay

Manh-Dung Nguyen

@dungnm1710

manh-dung.nguyen@cea.fr

PhD Student at CEA LIST & UGA

What's The Talk About?

- Fuzzing is great for finding vulnerabilities in the wild

- Directed fuzzing is a slightly different setting
  - Goal = reach a specific target
  - Bug reproduction, patch-oriented testing

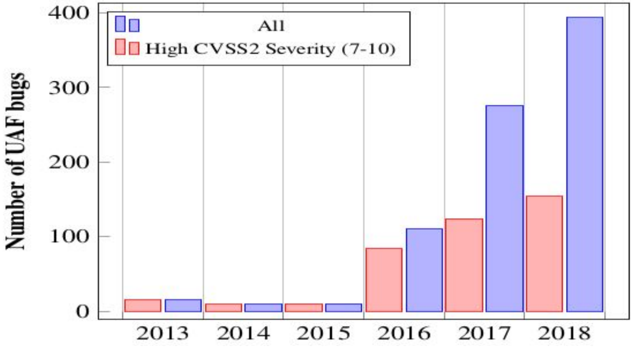
- The problem: Current fuzzing techniques are bad for some classes of issues
  - Here: "Use-After-Free" (UAF)
  - Important: sensitive info leaks, data corruption or first step to other attacks

- Proposal: A directed fuzzing approach tailored to UAF bugs
  - and applications to patch-oriented testing
  - and a tour on UAF and (directed) fuzzing

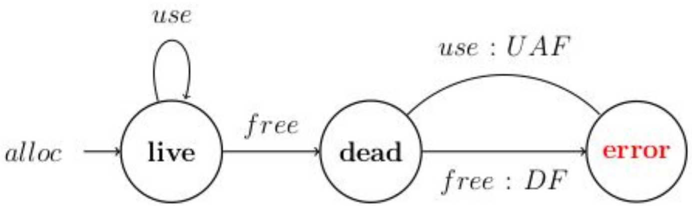
Use-After-Free

- Heap element is used after having been freed
- Critical exploits & serious consequences
  - Data corruption
  - Information leaks
  - Denial-of-service attacks

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



 # UAF bugs in National Vulnerability Database



Teaser

```
1  int *p, *p_alias;
2  char buf[10];
3  void bad_func(int *p) {free(p);} /* exit() is missing */
4  void func() {
5    if (buf[1] == 'F')
6      bad_func(p);
7    else /* lots more code ... */
8  }
9  int main (int argc, char *argv[]) {
10   int f = open(argv[1], O_RDONLY);
11   read(f, buf, 10);
12   p = malloc(sizeof(int));
13   if (buf[0] == 'A'){
14     p_alias = malloc(sizeof(int));
15     p = p_alias;
16   }
17   func();
18   if (buf[2] == 'U')
19     *p_alias = 1;
20   return 0;
21 }
```

free

alloc

use

- PoC: 'AFU' → no crash
- Bug Target: 14 (alloc) → 17 → 6 → 3 (free) → 19 (use)
- Timeout: 6h

AFL-QEMU (binary)	**AFLGo** (source)	**UAFuzz** (binary)
✗ (6 hours)	✗ (6 hours)	✓ (~ 20 mins)

1. Context

-- about fuzzing, directed fuzzing

Code-level Flaws: Fuzzing is The New Hype

Project Springfield

Fuzz your code before hackers do

AFL++

FUZZING/IO

Security Automation • Vulnerability Research

As Its Core, Fuzzing is Random Testing

-- and it starts a long time ago

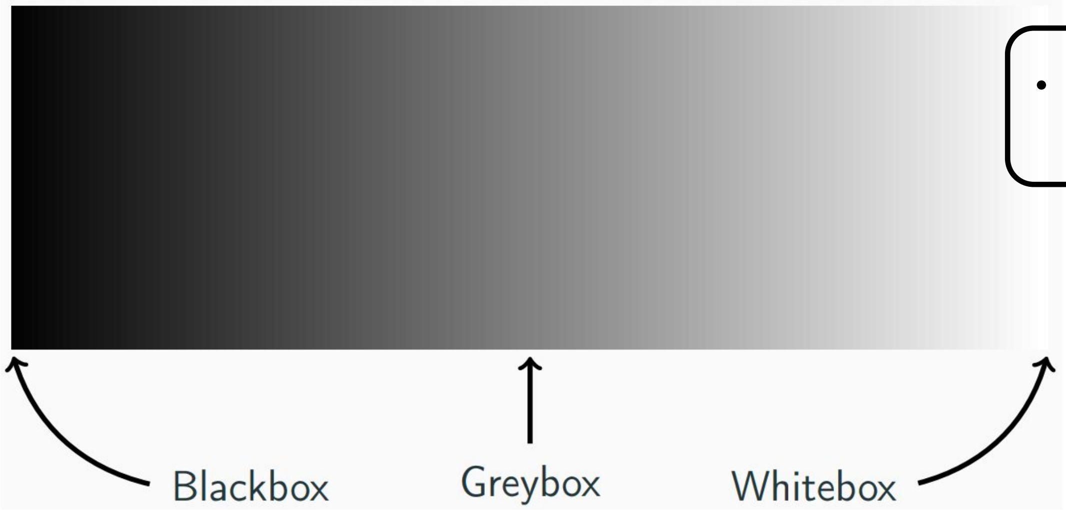
1981 Random testing is a cost-effective alternative to systematic testing techniques (Duran & Natos)

1983 "The Monkey" (Capps)

1988 Birth of the term "fuzzing" (Miller)

Now: Three Shades of Fuzzing

- The original taste
- Scale but dumb



- Smart but don't scale too much

- The new prodigy
- Try to be smart & scale

Principle of Grey/Black Fuzzing

1. Preprocess
2. Scheduling
3. Input Generation
4. Input Evaluation
5. Configuration Updating
6. Continue

Choose “good” inputs

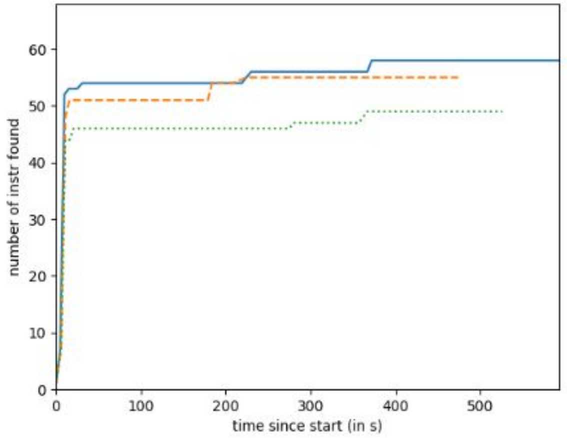
Mutations

Observe & compute score

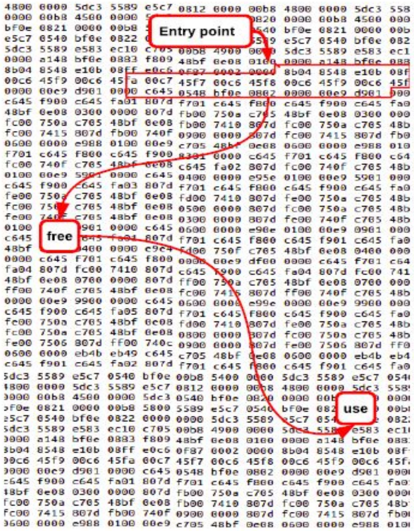
Greybox observes more

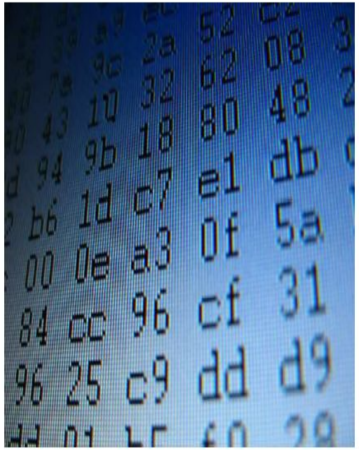
The art, science, and engineering of fuzzing: A survey (Manès et al. 2019)

No Silver Bullet



Plateau phenomenon





Complex Code Structure

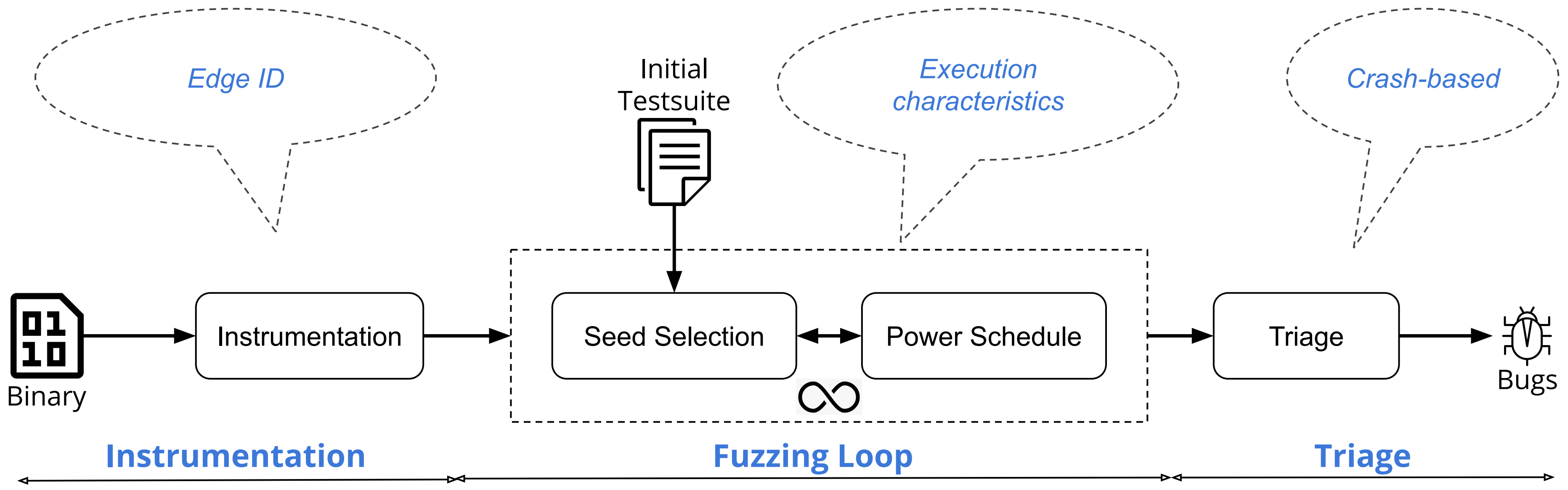
Complex Bugs

Target-oriented Testing?

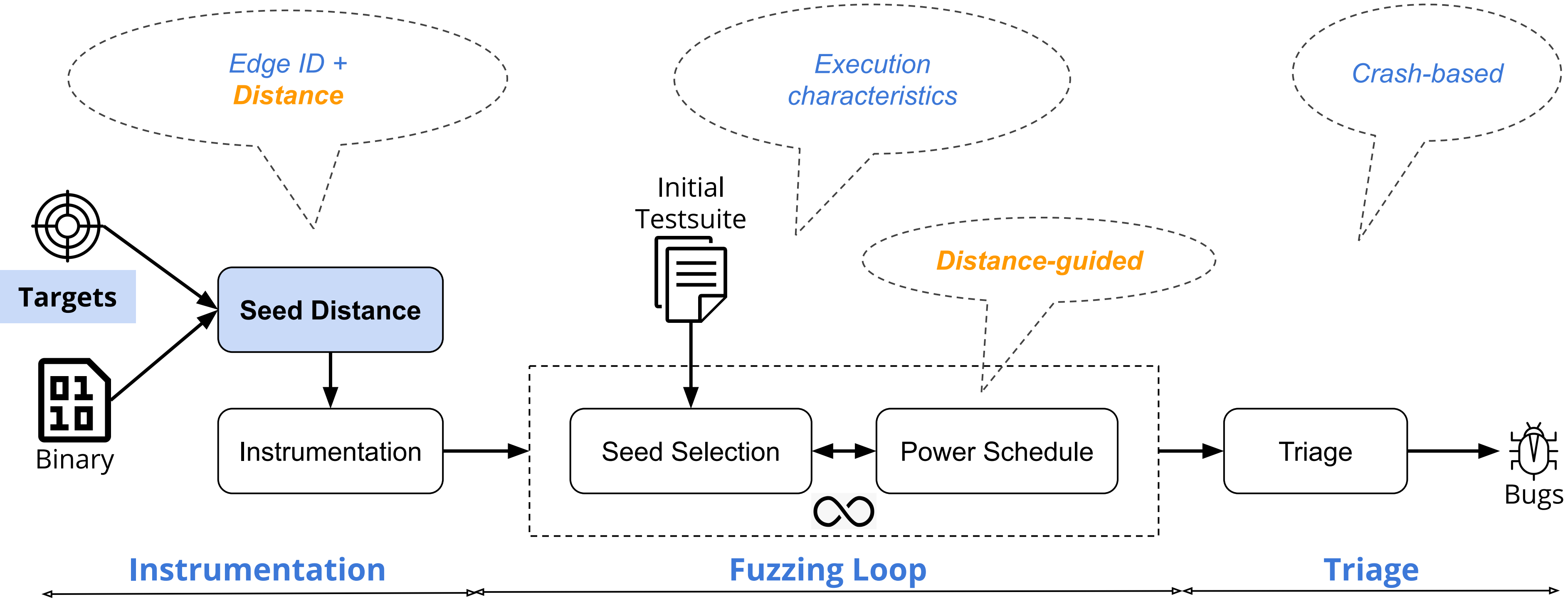
Directed Greybox Fuzzing (DGF)

- Input: code + target (trace, code location)
- Goal = Cover the target
- AFLGo (2017), Hawkeye (2018)
- Applications:
  - Bug reproduction
  - Patch-oriented testing
  - Static analysis report confirmation

Coverage-guided Greybox Fuzzing AFL



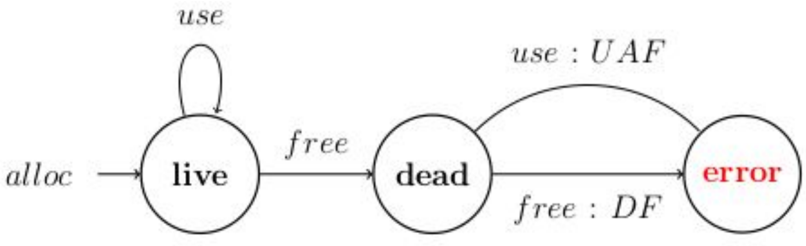
Directed Greybox Fuzzing AFLGo, Hawkeye

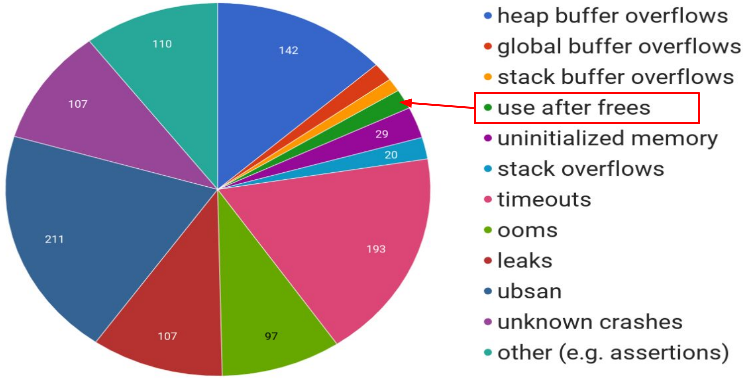


2. Back to Use-After-Free (UAF)

Why is Detecting UAF Hard for Fuzzing?

- 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*

Recall: Motivation

```c
1  int *p, *p_alias;
2  char buf[10];
3  void bad_func(int *p) {free(p);} /* exit() is missing */
4  void func() {
5    if (buf[1] == 'F')
6      bad_func(p);
7    else /* lots more code ... */
8  }
9  int main (int argc, char *argv[]) {
10   int f = open(argv[1], O_RDONLY);
11   read(f, buf, 10);
12   p = malloc(sizeof(int));
13   if (buf[0] == 'A'){
14     p_alias = malloc(sizeof(int));
15     p = p_alias;
16   }
17   func();
18   if (buf[2] == 'U')
19     *p_alias = 1;
20   return 0;
21 }
```

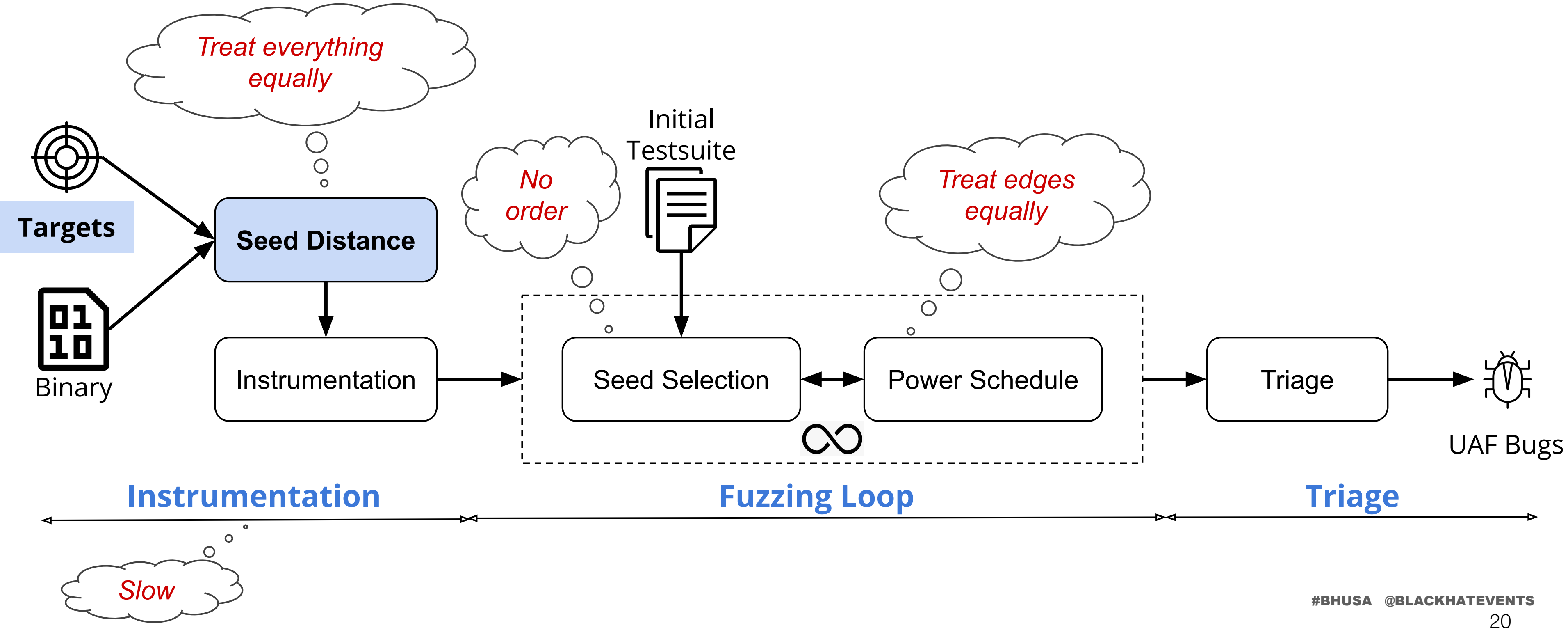
- PoC: 'AFU' → no crash
- Bug Target: 14 (alloc) → 17 → 6 → 3 (free) → 19 (use)
- Timeout: 6h

AFL-QEMU (binary)	**AFLGo** (source)	**UAFuzz** (binary)
✗ (6 hours)	✗ (6 hours)	✓ (~ 20 mins)

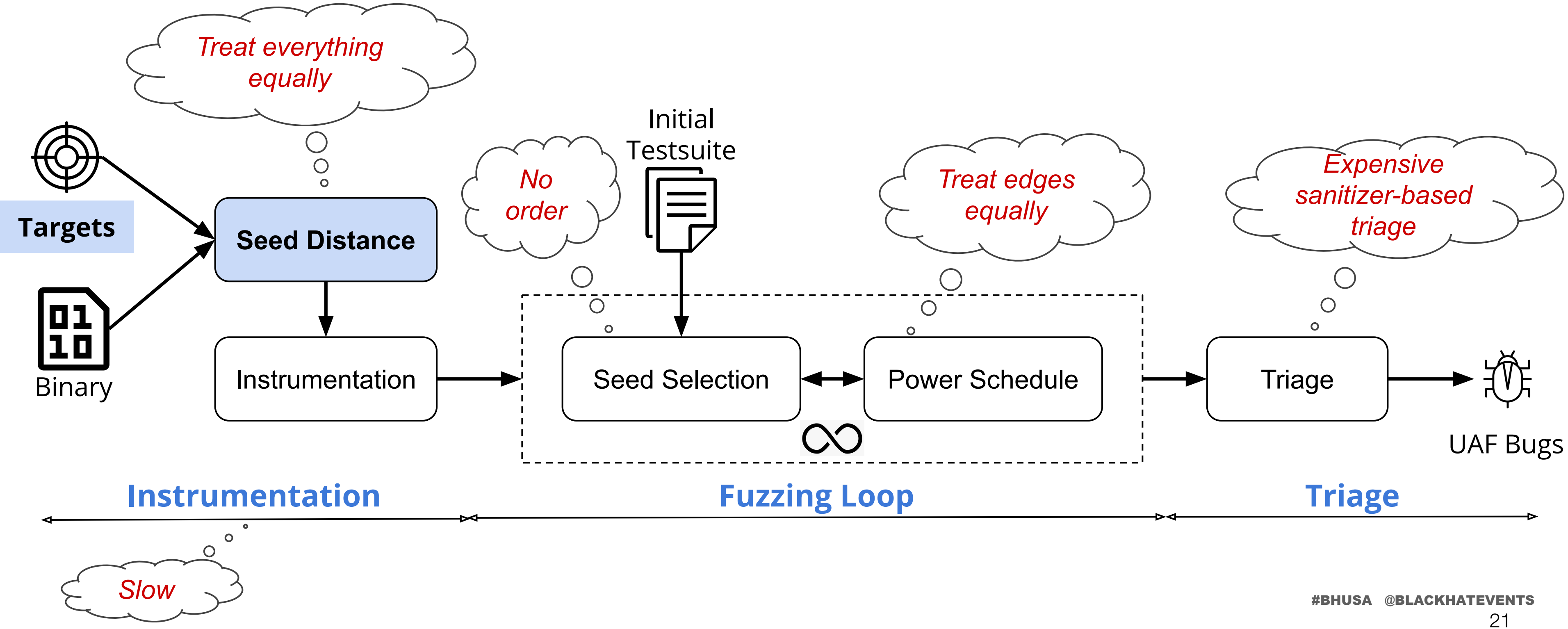
black hat
USA 2020
AUGUST 5-6, 2020
BRIEFINGS

3. UAFuzz: Directed Fuzzing for UAF

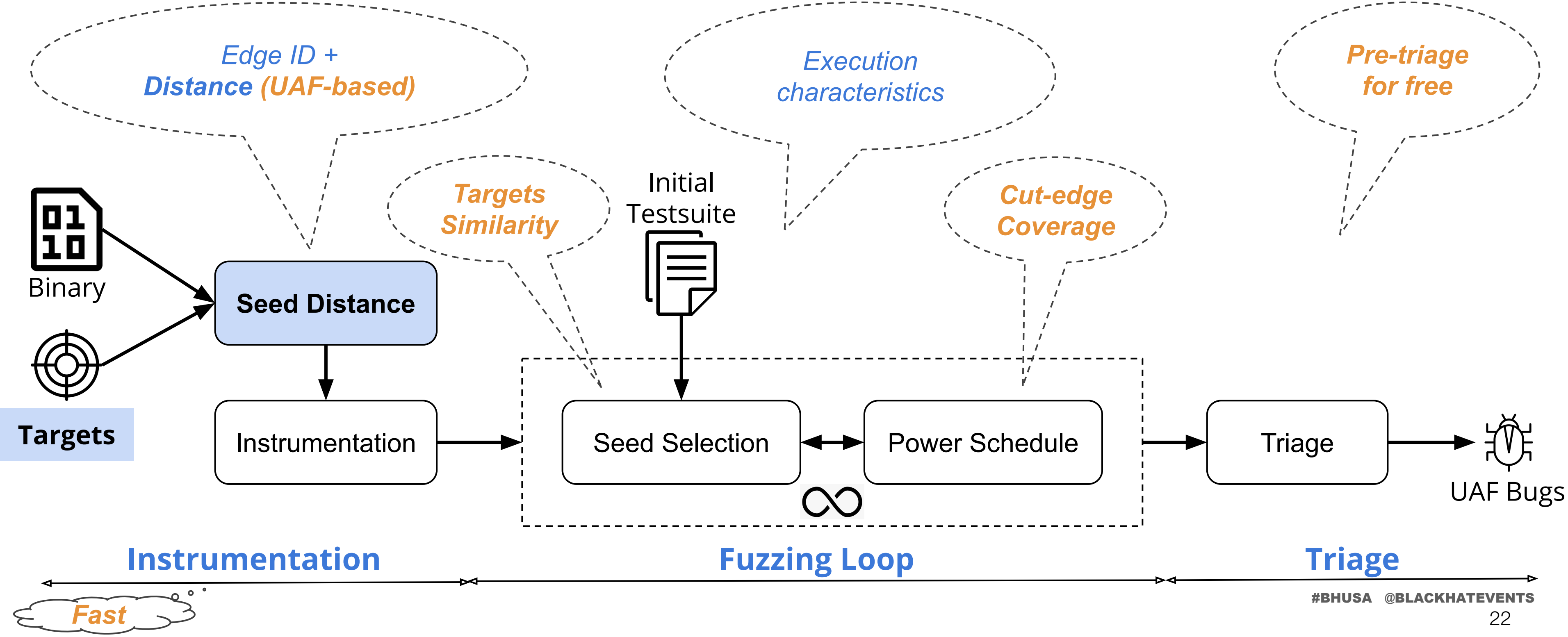
Existing DGF: #1 No Ordering & No Prioritization



Existing DGF: #2 Crash Assumption



Overview of UAFuzz [tailor every fuzzing step to UAF]



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 Bug Target

Stack Traces of CVE-2018-20623

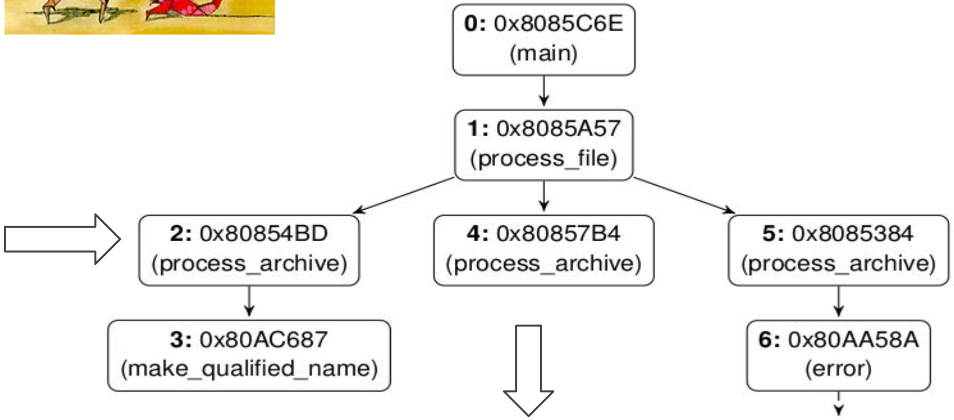
// stack trace for the bad Use
==4440== Invalid read of size 1
==4440== at 0x40A8383: vfprintf (vfprintf.c:1632)
==4440== by 0x40A8670: buffered_vfprintf (vfprintf.c:2320)
==4440== by 0x40A62D0: vfprintf (vfprintf.c:1293)
[6] ==4440== by 0x80AA58A: error (elfcomm.c:43)
[5] ==4440== by 0x8085384: process_archive (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)

Valgrind

Dynamic Calling Tree



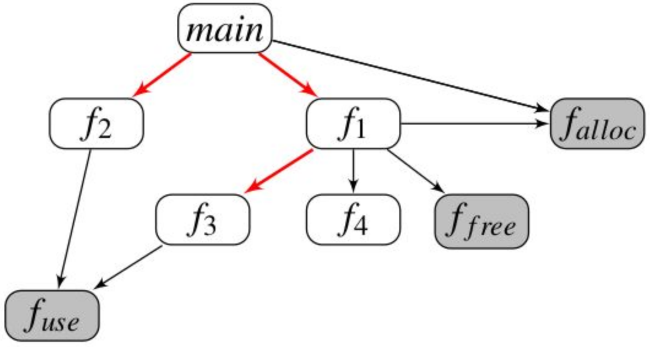
Bug Trace Flattening

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)*

UAF-based Distance Metric

- 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}>

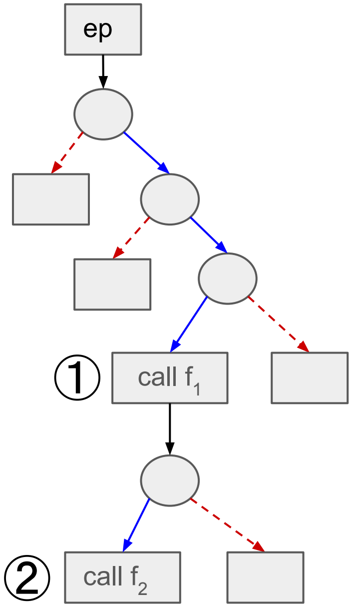


Example of Call Graph, favored pairs (caller, callee) are in red

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



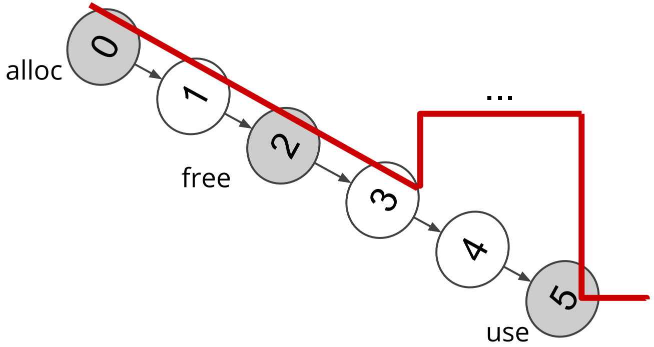
Control Flow Graph, cut edges are in blue

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

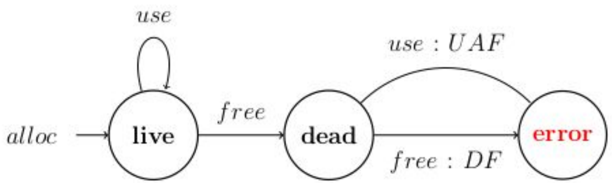
Intuition: UAFuzz assigns more energy (a.k.a, # mutants) to

- 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*)

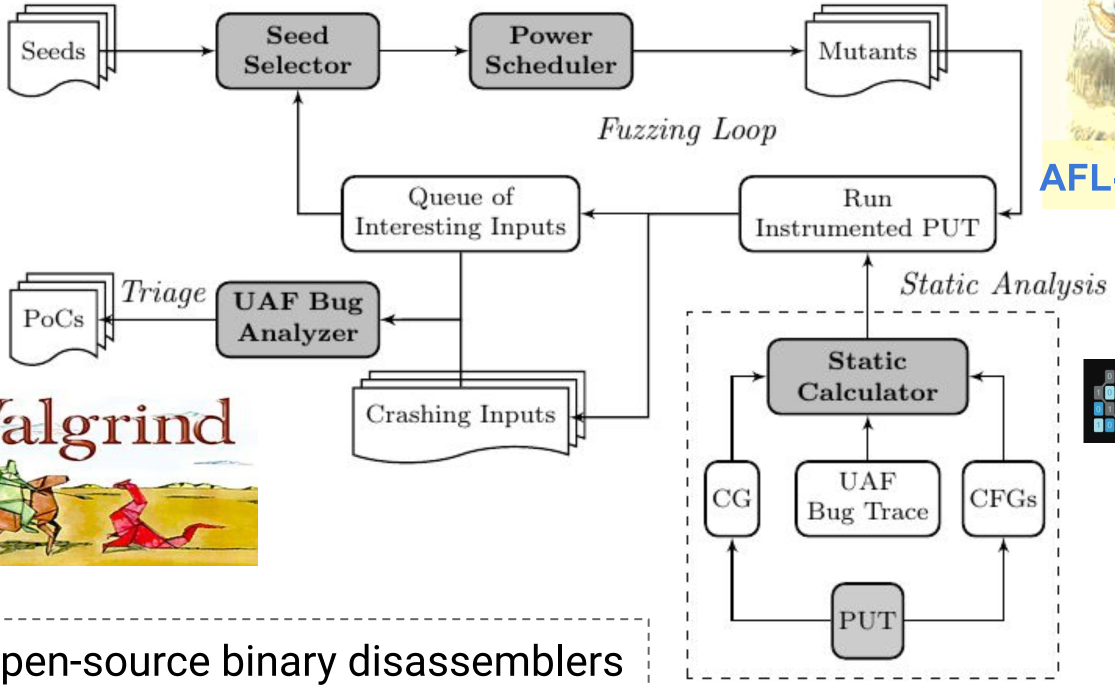
Pre-filter

- Existing works 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



AFL-QEMU

Support more open-source binary disassemblers

4. Experimental Evaluation

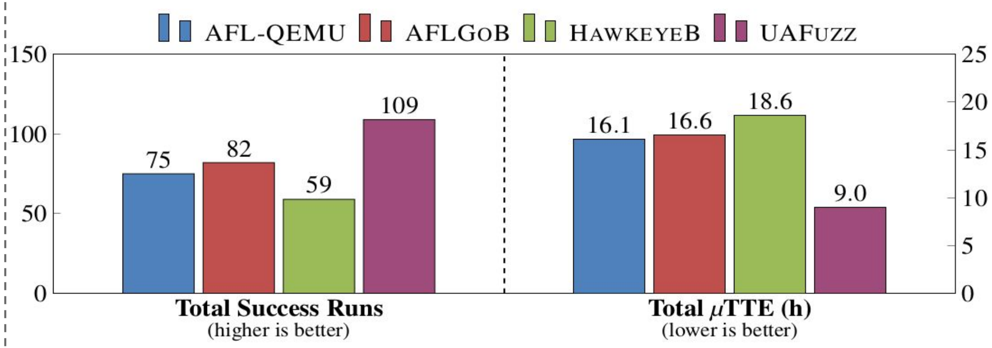
Evaluations

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

- Evaluated fuzzers
  - UAFuzz (BINSEC & AFL-QEMU)
  - AFL-QEMU
  - AFLGo (source - level) // Manh-Dung co-author
  - Our implementations AFLGoB & HawkeyeB
- Benchmark
  - 13 UAF bugs of real-world programs

Bug Reproduction: Fuzzing Performance

- 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**

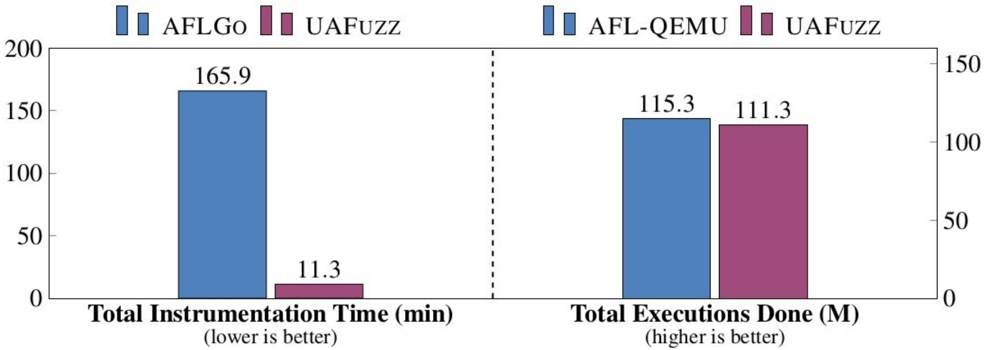


Bug-reproducing performance of binary-based DGFs

UAFuzz *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

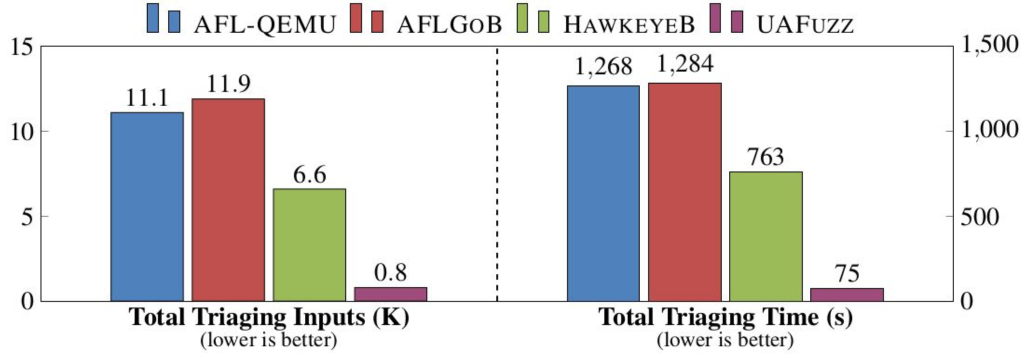


Global Overhead

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

5. Patch-Oriented Testing

Patch-Oriented 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

Patch-Oriented Testing: Zero-day Bugs

Program	Code Size	Version (Commit)	Bug ID	Vulnerability Type	Crash	Vulnerable Function	Status	CVE
GPAC	545K	0.7.1 (987169b)	#1269	**User after free**	✗	gf_m2ts_process_pmt	Fixed	CVE-2019-20628
		0.8.0 (56eaea8)	#1440-1	**User after free**	✗	gf_isom_box_del	Fixed	CVE-2020-11558
		0.8.0 (56eaea8)	#1440-2	**User after free**	✗	gf_isom_box_del	Fixed	Pending
		0.8.0 (56eaea8)	#1440-3	**User after free**	✗	gf_isom_box_del	Fixed	Pending
		0.8.0 (5b37b21)	#1427	**User after free**	✓	gf_m2ts_process_pmt		
		0.7.1 (987169b)	#1263	NULL pointer dereference	✓	ilst_item_Read	Fixed	
		0.7.1 (987169b)	#1264	Heap buffer overflow	✓	gf_m2ts_process_pmt	Fixed	CVE-2019-20629
		0.7.1 (987169b)	#1265	Invalid read	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1266	Invalid read	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1267	NULL pointer dereference	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1268	Heap buffer overflow	✓	BS_ReadByte	Fixed	CVE-2019-20630
		0.7.1 (987169b)	#1270	Invalid read	✓	gf_list_count	Fixed	CVE-2019-20631
		0.7.1 (987169b)	#1271	Invalid read	✓	gf_odf_delete_descriptor	Fixed	CVE-2019-20632
		0.8.0 (5b37b21)	#1445	Heap buffer overflow	✓	gf_bs_read_data	Fixed	
		0.8.0 (5b37b21)	#1446	Stack buffer overflow	✓	gf_m2ts_get_adaptation_field	Fixed	
GNU patch	7K	2.7.6 (76e7758)	#56683	**Double free**	✓	another_hunk	Confirmed	CVE-2019-20633
		2.7.6 (76e7758)	#56681	Assertion failure	✓	pch_swap	Confirmed	
		2.7.6 (76e7758)	#56684	Memory leak	✗	xmalloc	Confirmed	
Perl 5	184K	5.31.3 (a3c7756)	#134324	**Use after free**	✓	S_reg	Confirmed	
		5.31.3 (a3c7756)	#134326	**Use after free**	✓	Perl_regnext	Fixed	
		5.31.3 (a3c7756)	#134329	**User after free**	✓	Perl_regnext	Fixed	
		5.31.3 (a3c7756)	#134322	NULL pointer dereference	✓	do_clean_named_objs	Confirmed	
		5.31.3 (a3c7756)	#134325	Heap buffer overflow	✓	S_reg	Fixed	
		5.31.3 (a3c7756)	#134327	Invalid read	✓	S_regmatch	Fixed	
		5.31.3 (a3c7756)	#134328	Invalid read	✓	S_regmatch	Fixed	
		5.31.3 (45f8e7b)	#134342	Invalid read	✓	Perl_mro_isa_changed_in	Confirmed	
MuPDF	539K	1.16.1 (6566de7)	#702253	**Use after free**	✗	fz_drop_band_writer	Fixed	
Boolector	79K	3.2.1 (3249ae0)	#90	NULL pointer dereference	✓	set_last_occurrence_of_symbols	Confirmed	
fontforge	578K	20200314 (1604c74)	#4266	**Use after free**	✓	SFDGetBitmapChar		
		20200314 (1604c74)	#4267	NULL pointer dereference	✓	SFDGetBitmapChar		

Buggy Patch in GNU Patch CVE-2019-20633

```
==330== Invalid free() / delete / delete[] / realloc()
==330== at 0x402D358: free (in vgpreload_memcheck-x86-linux.so)
==330== by 0x8052E11: another_hunk (pch.c:1185)
==330== by 0x804C06C: main (patch.c:396)
==330== Address 0x4283540 is 0 bytes inside a block of size 2 free'd
==330== at 0x402D358: free (in vgpreload_memcheck-x86-linux.so)
==330== by 0x8052E11: another_hunk (pch.c:1185)
==330== by 0x804C06C: main (patch.c:396)
==330== Block was alloc'd at
==330== at 0x402C17C: malloc (in vgpreload_memcheck-x86-linux.so)
==330== by 0x805A821: savebuf (util.c:861)
==330== by 0x805423C: another_hunk (pch.c:1504)
==330== by 0x804C06C: main (patch.c:396)
```

Using the bug trace of CVE-2018-6952 produced by Valgrind, we found an incomplete fix of GNU Patch with one different call in red

6. Conclusion

Conclusion & Takeaways

- UAFuzz: A directed fuzzing framework to detect UAF bugs at binary level
- Find more bugs in bug reproduction than state-of-the-art tools
- New bugs and CVEs in patch-oriented testing

1. Directed Fuzzing exists, and it is practical
   -- should be integrated into dev. process in addition to standard fuzzing
2. Recent trend toward dedicated fuzzers (UAFuzz, PerfFuzz, MemLock ...)
   -- perform better than general fuzzers
3. Patch-oriented fuzzing is bigger than patch testing
4. Patching a PoC is not enough, we should find and fix variants of the bug class

Thank you ! Q & A

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

~~~

Paper: Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities (RAID'20)
UAFuzz: https://github.com/strongcourage/uafuzz
UAF Fuzzing Benchmark: https://github.com/strongcourage/uafbench
BINSEC v0.3: https://binsec.github.io/

Partially funded by European H2020 project C4IIOT
~~~