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

Manh-Dung Nguyen
Univ. Paris-Saclay, CEA LIST, France
manh-dung.nguyen@cea.fr

Sébastien Bardin
Univ. Paris-Saclay, CEA LIST, France
sebastien.bardin@cea.fr

Richard Bonichon
Tweag I/O, France
richard.bonichon@tweag.io

Roland Groz
Univ. Grenoble Alpes, France
roland.groz@univ-grenoble-alpes.fr

Matthieu Lemerre
Univ. Paris-Saclay, CEA LIST, France
matthieu.lemerre@cea.fr

Abstract

Directed fuzzing focuses on automatically testing specific parts of the code by taking advantage of additional information such as (partial) bug stack trace, patches or risky operations. Key applications include bug reproduction, patch testing and static analysis report verification. Although directed fuzzing has received a lot of attention recently, hard-to-detect vulnerabilities such as Use-After-Free (UAF) are still not well addressed, especially at the binary level. We propose UAFuzz, the first (binary-level) directed greybox fuzzer dedicated to UAF bugs. The technique features a fuzzing engine tailored to UAF specifics, a lightweight code instrumentation and an efficient bug triage step. Experimental evaluation for bug reproduction on real cases demonstrates that UAFuzz significantly outperforms state-of-the-art directed fuzzers in terms of fault detection rate, time to exposure and bug triaging. UAFuzz has also been proven effective in patch testing, leading to the discovery of 30 new bugs (7 CVEs) in programs such as Perl, GPAC and GNU Patch. Finally, we provide to the community a large fuzzing benchmark dedicated to UAF, built on both real codes and real bugs.

1 Introduction

Context. Finding bugs early is crucial in the vulnerability management process. The recent rise of fuzzing [50, 53] in both academia and industry, such as Microsoft's Springfield [52] and Google's OSS-Fuzz [14], shows its ability to find various types of bugs in real-world applications. *Coverage-based Greybox Fuzzing* (CGF), such as AFL [1] and libFuzzer [13], leverages code coverage information in order to guide input generation toward new parts of the program under test (PUT), exploring as many program states as possible in the hope of triggering crashes. On the other hand, *Directed Greybox Fuzzing* (DGF) [25, 28] aims to perform stress testing on pre-selected potentially vulnerable target locations, with applications to different security contexts: (1) *bug reproduction* [25, 28, 42, 61], (2) *patch testing* [25, 51, 59] or (3) static analysis report verification [31, 49]. Depending on the application, target locations are originated from *bug stack traces*, patches or static analysis reports.

We focus mainly on *bug reproduction*, which is the most common practical application of DGF [25, 28, 42, 49, 74]. It consists in generating Proof-of-Concept (PoC) inputs of disclosed vulnerabilities given bug report information. It is especially needed since only 54.9% of usual bug reports can be reproduced due to missing information and users' privacy violation [54]. Even with a PoC provided in the bug report, developers may still need to consider all corner cases of the bug in order to avoid regression bugs or incomplete fixes. In this case, providing more bug-triggering inputs becomes important to facilitate and accelerate the repair process. *Bug stack traces*, sequences of function calls at the time a bug is triggered, are widely used for guiding directed fuzzers [25, 28, 42, 49]. Running a code on a PoC input under profiling tools like AddressSanitizer (ASan) [65] or Valgrind [55] will output such a bug stack trace.

Problem. Despite tremendous progress in the past few years [5, 21, 23, 25, 28, 29, 39, 46, 47, 60, 62, 67, 73], current (directed or not) greybox fuzzers still have a hard time finding *complex vulnerabilities* such as Use-After-Free (UAF), non-interference or flaky bugs [24], which require bug-triggering paths satisfying very specific properties. For example, OSS-Fuzz [14, 15] or recent greybox fuzzers [25, 62, 73] only found a small number of UAF. Actually, Rode0day [16], a continuous bug finding competition, recognizes that fuzzers should aim to cover new bug classes like UAF in the future [37], moving further from the widely-used LAVA [36] bug corpora which only contains buffer overflows.

We focus on UAF bugs. They appear when a heap element is used after having been freed. The numbers of UAF bugs has increased in the National Vulnerability Database (NVD) [20]. They are currently identified as *one of the most critical exploitable vulnerabilities* due to the lack of mitigation techniques compared to other types of bugs, and they may have serious consequences such as data corruption, information leaks and denial-of-service attacks.

Goal and challenges. *We focus on the problem of designing an efficient directed fuzzing method tailored for UAF.* The technique must also be able to work at binary-level (no source-level instrumentation), as source codes of security-critical programs are not always available or may rely partly on third-party libraries. However, fuzzers targeting the detection of UAF bugs confront themselves with the following challenges.

C1. **Complexity** – Exercising UAF bugs require to generate inputs triggering a *sequence* of 3 events – *alloc*, *free* and *use* – *on the same memory location*, spanning multiple functions of the PUT, where buffer overflows only require a single out-of-bound memory access. This combination of both *temporal* and *spatial* constraints is extremely difficult to meet in practice;

C2. **Silence** – UAF bugs often have *no observable effect*,

Table 1: Summary of existing greybox fuzzing techniques.

	AFL	AFLGo	Hawkeye	UAFuzz
Directed fuzzing approach	✗	✓	✓	✓
Support binary	✓	✗	✗	✓
UAF bugs oriented	✗	✗	✗	✓
Fast instrumentation	✓	✗	✗	✓
UAF bugs triage	✗	✗	✗	✓

such as segmentation faults. In this case, fuzzers simply observing crashing behaviors do not detect that a test case triggered such a memory bug. Sadly, popular profiling tools such as ASan or Valgrind cannot be used in a fuzzing context due to their high runtime overhead.

Actually, current state-of-the-art directed fuzzers, namely AFLGo [25] and Hawkeye [28], fail to address these challenges. First, they are too generic and therefore do not cope with the specificities of UAF such as temporality – their guidance metrics do not consider any notion of sequenceness. Second, they are completely blind to UAF bugs, requiring to send all the many generated seeds to a profiling tool for an expensive extra check. Finally, current implementations of source-based DGF fuzzers typically suffer from an expensive instrumentation step [3], e.g., AFLGo spent nearly 2*h* compiling and instrumenting `cxxfilt` (Binutils).

Proposal. We propose UAFuzz, *the first (binary-level) directed greybox fuzzer tailored to UAF bugs.* A quick comparison of UAFuzz with existing greybox fuzzers in terms of UAF is presented in Table 1. While we follow mostly the generic scheme of directed fuzzing, we carefully tune several of its key components to the specifics of UAF:

- the *distance metric* favors shorter call chains leading to the target functions that are more likely to include both allocation and free functions – where sota directed fuzzers rely on a generic CFG-based distance;
- *seed selection* is now based on a *sequenceness-aware target similarity metric* – where sota directed fuzzers rely at best on target coverage;
- our *power schedule* benefits from these new metrics, plus another one called *cut-edges* favoring prefix paths more likely to reach the whole target.

Finally, the bug triaging step piggy-backs on our previous metrics to pre-identifies seeds as likely-bugs or not, sparing a huge amount of queries to the profiling tool for confirmation (Valgrind in our implementation).

Contributions. Our contribution is the following:

- We design the first directed greybox fuzzing technique *tailored to* UAF bugs (Section 4). Especially, we systematically revisit the three main ingredients of directed fuzzing (selection heuristic, power schedule, input metrics) and specialize them to UAF. These improvements are proven beneficial and complementary;
- We develop a toolchain [19] on top of the state-of-the-art greybox fuzzer AFL [1] and the binary analysis platform Binsec [4], named UAFuzz, implementing the above method for UAF directed fuzzing over binary codes (Section 5) and enjoying small overhead;
- We construct and openly release [18] the largest fuzzing benchmark dedicated to UAF, comprising **30** real bugs from **17** widely-used projects (including the few previous UAF bugs found by directed fuzzers), in the hope of facilitating future UAF fuzzing evaluation;
- We evaluate our technique and tool in a bug reproduction setting (Section 6), demonstrating that UAFuzz is highly effective and significantly outperforms state-of-the-art competitors: 2× faster in average to trigger bugs (up to 43×), +34% more successful runs in average (up to +300%) and 17× faster in triaging bugs (up to 130×, with 99% spare checks);
- Finally, UAFuzz is also proven effective in patch testing (§6.7), leading to the discovery of 30 unknown bugs (11 UAFs, 7 CVEs) in projects like Perl, GPAC, MuPDF and GNU Patch (including 4 buggy patches). So far, 17 have been fixed.

UAFuzz is the *first* directed greybox fuzzing approach tailored to detecting UAF vulnerabilities (in binary) given only bug stack traces. UAFuzz outperforms existing directed fuzzers on this class of vulnerabilities for bug reproduction and encouraging results have been obtained as well on patch testing. We believe that our approach may also be useful in slightly related contexts, for example partial bug reports from static analysis or other classes of vulnerabilities.

2 Background

Let us first clarify some notions used along the paper.

2.1 Use-After-Free

Execution. An *execution* is the complete sequence of states executed by the program on an *input*. An execution trace *crashes* when it ends with a visible error. The standard goal of fuzzers is to find inputs leading to crashes, as crashes are the first step toward exploitable vulnerabilities.

UAF bugs. *Use-After-Free* (UAF) bugs happen when dereferencing a pointer to a heap-allocated object which is no longer valid (i.e., the pointer is *dangling*). Note that *Double-Free* (DF) is a special case.

UAF-triggering conditions. Triggering a UAF bug requires to find an input whose execution covers in sequence *three* UAF *events*: an allocation (*alloc*), a *free* and a *use* (typically, a dereference), *all three referring to the same memory object*, as shown in Listing 1.

```
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
```

Listing 1: Code snippet illustrating a UAF bug.

Furthermore, this last *use* generally does not make the execution immediately crash, as a memory violation crashes a

process only when it accesses an address outside of the address space of the process, which is unlikely with a dangling pointer. Thus, UAF bugs go often unnoticed and are a good vector of exploitation [45, 75].

2.2 Stack Traces and Bug Traces

By inspection of the state of a process we can extract a *stack trace*, i.e. the list of the function calls active in that state. Stack traces are easily obtained from a process when it crashes. As they provide (partial) information about the sequence of program locations leading to a crash, they are extremely valuable for bug reproduction [25, 28, 42, 49].

Yet, as crashes caused by UAF bugs may happen long after the UAF happened, standard stack traces usually do not help in reproducing UAF bugs. Hopefully, profiling tools for dynamically detecting memory corruptions, such as ASan [65] or VALGRIND [55], record the stack traces of *all memory-related events*: when they detect that an object is used after being freed, they actually report *three stack traces* (when the object is allocated, when it is freed and when it is used after being freed). *We call such a sequence of three stack traces a **(UAF) bug trace***. When we use a bug trace as an input to try to reproduce the bug, we call such a bug trace a *target*.

```
    // 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)
==4440== by 0x80AA58A: error (elfcomm.c:43)
==4440== by 0x8085384: process_archive (readelf.c:19063)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==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)
==4440== by 0x80857B4: process_archive (readelf.c:19178)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==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)
==4440== by 0x80AC687: make_qualified_name (elfcomm.c:906)
==4440== by 0x80854BD: process_archive (readelf.c:19089)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==4440== by 0x8085C6E: main (readelf.c:19318)
```

Figure 1: Bug trace of CVE-2018-20623 (UAF) produced by VALGRIND.

2.3 Directed Greybox Fuzzing

Fuzzing [50, 53] consists in stressing a code under test through massive input generation in order to find bugs. Recent *coverage-based greybox fuzzers* (CGF) [1, 13] rely on *lightweight* program analysis to guide the search – typically through coverage-based feedback. Roughly speaking, a *seed* (input) is *favored* (selected) when it reaches under-explored parts of the code, and such favored seeds are then *mutated* to create new seeds for the code to be executed on. CGF is geared toward covering code in the large, in the hope of finding unknown vulnerabilities.

On the other hand, *directed greybox fuzzing* (DGF) [25, 28] aims at reaching a *pre-identified potentially buggy part of the code* from a *target* (e.g., patch, static analysis report), as often and fast as possible. Directed fuzzers follow the general principles and architecture as CGF, but adapt the key components to their goal, essentially favoring seeds *"closer"* to the target. Overall directed fuzzers[1] are built upon three main steps: (1) *instrumentation* (distance pre-computation), (2) *fuzzing* (including seed selection, power schedule and seed mutation) and (3) *triage*.

The standard core algorithm of DGF is presented in Algorithm 1 (the parts we modify in UAFUZZ are in gray). Given a program P, a set of initial seeds S_0 and a target T, the algorithm outputs a set of bug-triggering inputs S'. The fuzzing queue S is initialized with the initial seeds in S_0 (line 1).

1. DGF first performs a static analysis (e.g., *target distance computation* for each basic block) and insert the instrumentation for dynamic coverage or distance information (line 2);
2. The fuzzer then repeatedly mutates inputs s chosen from the fuzzing queue S (line 4) until a timeout is reached. An input is selected either if it is *favored* (i.e., believed to be interesting) or with a small probablity α (line 5). Subsequently, DGF assigns the *energy* (a.k.a, the number M of mutants to be created) to the selected seed s (line 6). Then, the fuzzer generates M new inputs by randomly applying some predefined mutation operators on seed s (line 8) and monitors their executions (line 9). If the generated mutant s' crashes the program, it is added to the set S' of crashing inputs (line 11). Also, newly generated mutants are added to the fuzzing queue[2] (line 13);
3. Finally, DGF returns S' as the set of bug-triggering inputs (triage does nothing in standard DGF) (line 14).

Algorithm 1: Directed Greybox Fuzzing

Input : Program P; Initial seeds S_0; Target locations T
Output : Bug-triggering seeds S'

```
1  S' := ∅; S := S0;                                   ▷ S: the fuzzing queue
2  P' ← PREPROCESS(P, T);                              ▷ phase 1: Instrumentation
3  while timeout not exceeded do                       ▷ phase 2: Fuzzing
4      for s ∈ S do
5          if IS_FAVORED(s) or rand() ≤ α then
               ▷ seed selection, α: small probability
6              M := ASSIGN_ENERGY(s);                  ▷ power schedule
7              for i ∈ 1 ... M do
8                  s' := mutate_input(s);              ▷ seed mutation
9                  res := run(P', s', T);
10                 if is_crash(res) then
11                     S' := S' ∪ {s'};                ▷ crashing inputs
12                 else
13                     S := S ∪ {s'};
14 S' = TRIAGE(S, S');                                 ▷ phase 3: Triage
15 return S';
```

[1] And coverage-based fuzzers.
[2] This is a general view. In practice, seeds regarded as very uninteresting are already discarded at this point.

AFLGo [25] was the first to propose a CFG-based distance to evaluate the proximity between a seed execution and multiple targets, together with a simulated annealing-based power schedule. HAWKEYE [28] keeps the CFG-based view but improves its accuracy[3], brings a seed selection heuristic partly based on target coverage (seen as a set of locations) and proposes adaptive mutations.

3 Motivation

The toy example in Listing 2 contains a UAF bug due to a missing `exit()` call, a common root cause in such bugs (e.g., CVE-2014-9296, CVE-2015-7199). The program reads a file and copies its contents into a buffer `buf`. Specifically, a memory chunk pointed at by `p` is allocated (line 12), then `p_alias` and `p` become aliased (line 15). The memory pointed by both pointers is freed in function `bad_func` (line 10). The UAF bug occurs when the freed memory is dereferenced again via `p` (line 19).

Bug-triggering conditions. The UAF bug is triggered iff the first three bytes of the input are '`AFU`'. To quickly detect this bug, fuzzers need to explore the right path through the `if` part of conditional statements in lines 13, 5 and 18 in order to cover in sequence the three UAF events *alloc*, *free* and *use* respectively. It is worth noting that this UAF bug does not make the program crash, hence existing greybox fuzzers without sanitization will not detect this memory error.

Coverage-based Greybox Fuzzing. Starting with an empty seed, AFL quickly generates 3 new inputs (e.g., '`AAAA`', '`FFFF`' and '`UUUU`') triggering individually the 3 UAF events. None of these seeds triggers the bug. As the probability of generating an input starting with '`AFU`' from an empty seed is extremely small, the coverage-guided mechanism is not effective here in tracking a sequence of UAF events even though each individual event is easily triggered.

Directed Greybox Fuzzing. Given a bug trace (14 – *alloc*, 17, 6, 3 – *free*, 19 – *use*) generated for example by ASan, DGF prevents the fuzzer from exploring undesirable paths, e.g., the `else` part at line 7 in function `func`, in case the condition at line 5 is more complex. Still, directed fuzzers have their own blind spots. For example, standard DGF seed selection mechanisms favor a seed whose execution trace covers many locations in targets, instead of trying to reach these locations in a given order. For example, regarding a target (A, F, U), standard DGF distances [25, 28] do not discriminate between an input s_1 covering a path $A \rightarrow F \rightarrow U$ and another input s_2 covering $U \rightarrow A \rightarrow F$. The lack of ordering in exploring target locations makes UAF bug detection very challenging for existing directed fuzzers. Another example: the power function proposed by HAWKEYE [28] may assign much energy to a seed whose trace does not reach the target function, implying that it could get lost on the toy example in the `else` part at line 7 in function `func`.

[3] Possibly at the price of both higher pre-computation costs due to more precise static analysis and runtime overhead due to complex seed metrics.

```
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 = 1;
20   return 0;
21 }
```

Listing 2: Motivating example.

A glimpse at UAFuzz. We rely in particular on modifying the seed selection heuristic w.r.t. the number of targets covered by an execution trace (§4.2) and bringing target ordering-aware seed metrics to DGF (§4.3).

On the toy example, UAFuzz generates inputs to progress towards the expected target sequences. For example, in the same fuzzing queue containing 4 inputs, the mutant '`AFAA`', generated by mutating the seed '`AAAA`', is discarded by AFL as it does not increase code coverage. However, since it has maximum value of target similarity metric score (i.e., 4 targets including lines 14, 17, 6, 3) compared to all 4 previous inputs in the queue (their scores are 0 or 2), this mutant is selected by UAFuzz for subsequent fuzzing campaigns. By continuing to fuzz '`AFAA`', UAFuzz eventually produces a bug-triggering input, e.g., '`AFUA`'.

Evaluation. AFLGo *(source-level) cannot detect the UAF bug within 2 hours*[4][5]*, while* UAFuzz *(binary-level) is able to trigger it within 20 minutes. Also,* UAFuzz *sends a single input to* Valgrind *for confirmation (the right PoC input), while* AFLGo *sends 120 inputs.*

4 The UAFuzz Approach

UAFuzz is made out of several components encompassing seed selection (§4.2), input metrics (§4.3), power schedule (§4.4), and seed triage (§4.5). Before detailing these aspects, let us start with an overview of the approach.

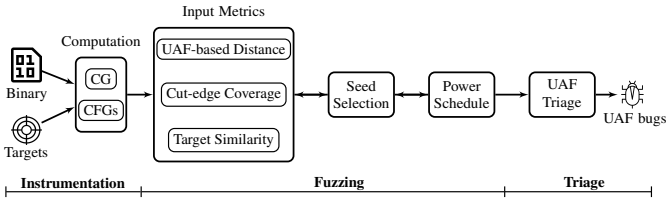

Figure 2: Overview of UAFuzz.

[4] AFL-QEMU did not succeed either.

[5] HAWKEYE is not available and thus could not be tested.

We aim to find an input fulfilling both control-flow (temporal) and runtime (spatial) conditions to trigger the UAF bug. We solve this problem by bringing UAF characteristics into DGF in order to generate more potential inputs reaching targets in sequence w.r.t. the UAF expected bug trace. Figure 2 depicts the general picture. Especially:

- We propose three dynamic seed metrics specialized for UAF vulnerabilities detection. The distance metric approximates how close a seed is to all target locations (§4.3), and takes into account the need for the seed execution trace to cover the three UAF events in order. The cut-edge coverage metric (§4.4.1) measures the ability of a seed to take the correct decision at important decision nodes. Finally, the target similarity metric concretely assesses how many targets a seed execution trace covers at runtime (§4.2.2);
- Our seed selection strategy (§4.2) favors seeds covering more targets at runtime. The power scheduler determining the energy for each selected seed based on its metric scores during the fuzzing process is detailed in §4.4;
- Finally, we take advantage of our previous metrics to pre-identify likely-PoC inputs that are sent to a profiling tool (here VALGRIND) for bug confirmation, avoiding many useless checks (§4.5).

4.1 Bug Trace Flattening

A bug trace (§2.2) is a sequence of stack traces, i.e. it is a large object not fit for the lightweight instrumentation required by greybox fuzzing. The most valuable information that we need to extract from a bug trace is the sequence of basic blocks (and functions) that were traversed, which is an easier object to work with. We call this extraction *bug trace flattening*.

The operation works as follows. First, each of the three stack-traces is seen as a path in a call tree; we thus merge all the stack traces to re-create that tree. Some of the nodes in the tree have several children; we make sure that the children are ordered according to the ordering of the UAF events (i.e. the child coming from the *alloc* event comes before the child coming from the *free* event). Figure 3 shows an example of tree for the bug trace given in Figure 1.

Finally, we perform a preorder traversal of this tree to get a sequence of target locations (and their associated functions), which we will use in the following algorithms.

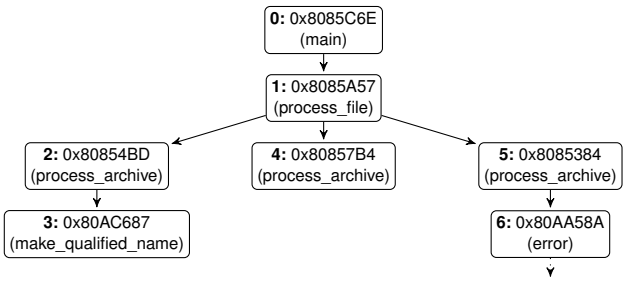

Figure 3: Reconstructed tree from CVE-2018-20623 (bug trace from Figure 1). The preorder traversal of this tree is simply $0 \rightarrow 1 \rightarrow 2 \rightarrow 3(n_{alloc}) \rightarrow 4(n_{free}) \rightarrow 5 \rightarrow 6(n_{use})$.

4.2 Seed Selection based on Target Similarity

Fuzzers generate a large number of inputs so that smartly selecting the seed from the fuzzing queue to be mutated in the next fuzzing campaign is crucial for effectiveness. Our seed selection algorithm is based on two insights. First, we *should prioritize seeds that are most similar to the target bug trace*, as the goal of a directed fuzzer is to find bugs covering the target bug trace. Second, *target similarity should take ordering (a.k.a. sequenceness) into account*, as traces covering sequentially a number of locations in the target bug trace are closer to the target than traces covering the same locations in an arbitrary order.

4.2.1 Seed Selection

Definition 1. A *max-reaching input* is an input s whose execution trace is the most similar to the target bug trace T so far, where most similar means "having the highest value as compared by a target similarity metric $t(s,T)$".

Algorithm 2: IS_FAVORED

Input : A seed s
Output : *true* if s is favored, otherwise *false*

```
1 global t_max = 0;          ▷ maximum target similar metric score
2 if t(s) ≥ t_max then t_max = t(s); return true;          ▷ update t_max
3 else if new_cov(s) then return true;          ▷ increase coverage
4 else return false;
```

We mostly select and mutate max-reaching inputs during the fuzzing process. Nevertheless, we also need to improve code coverage, thus UAFUZZ also selects inputs that cover new paths, with a small probability α (Algorithm 1). In our experiments, the probability of selecting the remaining inputs in the fuzzing queue that are less favored is 1% like AFL [1].

4.2.2 Target Similarity Metrics

A *target similarity metric* $t(s,T)$ measures the similarity between the execution of a seed s and the target UAF bug trace T. We define 4 such metrics, based on whether we consider ordering of the covered targets in the bug trace (P), or not (B) – P stands for Prefix, B for Bag; and whether we consider the full trace, or only the three UAF events ($3T$):

- Target prefix $t_P(s,T)$: locations in T covered in sequence by executing s until first divergence;
- UAF prefix $t_{3TP}(s,T)$: UAF events of T covered in sequence by executing s until first divergence;
- Target bag $t_B(s,T)$: locations in T covered by executing s;
- UAF bag $t_{3TB}(s,T)$: UAF events of T covered by s.

For example, using Listing 2, the 4 metric values of a seed s '`ABUA`' w.r.t. the UAF bug trace T are:
$t_P(s,T) = 2$, $t_{3PT}(s,T) = 1$, $t_B(s,T) = 3$ and $t_{3TB}(s,T) = 2$.

These 4 metrics have different degrees of *precision*. A metric t is said *more precise than* a metric t' if, for any two

seeds s_1 and s_2: $t(s_1,T) \geq t(s_2,T) \Rightarrow t'(s_1,T) \geq t'(s_2,T)$. Figure 4 compares our 4 metrics w.r.t their relative precision.

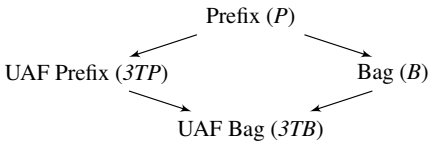

Figure 4: Precision lattice for Target Similarity Metrics

4.2.3 Combining Target Similarity Metrics

Using a precise metric such as P allows to better assess progression towards the goal. In particular, P can distinguish seeds that match the target bug trace from those that do not, while other metrics cannot. On the other hand, a less precise metric provides information that precise metrics do not have. For instance, P does not measure any difference between traces whose suffix would match the target bug trace, but who would diverge from the target trace on the first locations (like '`UUU`' and '`UFU`' on Listing 2), while B can.

To take benefit from both precise and imprecise metrics, we combine them using a lexicographical order. Hence, the *P-3TP-B* metric is defined as:

$$t_{P-3TP-B}(s,T) \triangleq \langle t_P(s,T), t_{3TP}(s,T), t_B(s,T) \rangle$$

This combination favors first seeds that cover the most locations in the prefix, then (in case of tie) those reaching the most number of UAF events in sequence, and finally (in case of tie) those that reach the most locations in the target. Based on preliminary investigation, we default to *P-3TP-B* for seed selection in UAFuzz.

4.3 UAF-based Distance

One of the main component of directed greybox fuzzers is the computation of a *seed distance*, which is an evaluation of a distance from the execution trace of a seed s to the target. The main heuristic here is that if the execution trace of s is close to the target, then s is close to an input that would cover the target, which makes s an interesting seed. In existing directed greybox fuzzers [2, 28], the seed distance is computed to a target which is a single location or a set of locations. This is not appropriate for the reproduction of UAF bugs, that must go through 3 different locations in sequence. Thus, we propose to modify the seed distance computation to take into account the need to reach the locations in order.

4.3.1 Zoom: Background on Seed Distances

Existing directed greybox fuzzers [2,28] compute the distance $d(s,T)$ from a seed s to a target T as follows.

AFLGo**'s seed distance [2].** The *seed distance* $d(s,T)$ is defined as the (arithmetic) mean of the *basic-block distances* $d_b(m,T)$, for each basic block m in the execution trace of s.

The *basic-block distance* $d_b(m,T)$ is defined using the length of the intra-procedural shortest path from m to the basic block of a "call" instruction, using the CFG of the function containing m; and the length of the inter-procedural shortest path from the function containing m to the target functions T_f (in our case, T_f is the function where the *use* event happens), using the call graph.

Hawkeye**'s enhancement [28].** The main factor in this seed distance computation is computing distance between functions in the call graph. To compute this, AFLGo uses the original call graph with every edge having weight 1. Hawkeye improves this computation by proposing the augmented adjacent-function distance (AAFD), which changes the edge weight from a caller function f_a and a callee f_b to $w_{Hawkeye}(f_a,f_b)$. The idea is to favor edges in the call graph where the callee can be called in a variety of situations, i.e. appear several times at different locations.

4.3.2 Our UAF-based Seed Distance

Previous seed distances [2, 28] do not account for any order among the target locations, while it is essential for UAF. We address this issue by modifying the distance between functions in the call graph to favor paths that *sequentially* go through the three UAF events *alloc*, *free* and *use* of the bug trace. This is done by decreasing the weight of the edges in the call graph that are likely to be between these events, using lightweight static analysis.

This analysis first computes R_{alloc}, R_{free}, and R_{use}, i.e., the sets of functions that can call respectively the *alloc*, *free*, or *use* function in the bug trace – the *use* function is the one where the *use* event happens. Then, we consider each call edge between f_a and f_b as indicating a direction: either downward (f_a executes, then calls f_b), or upward (f_b is called, then f_a is executed). Using this we compute, for each direction, how many events in sequence can be covered by going through the edge in that direction. For instance, if $f_a \in R_{alloc}$ and $f_b \in R_{free} \cap R_{use}$, then taking the $f_a \rightarrow f_b$ call edge possibly allows to cover the three UAF events in sequence. To find double free, we also include, in this computation, call edges that allow to reach two *free* events in sequence.

Then, we favor a call edge from f_a to f_b by decreasing its weight, based on how many events in sequence the edge allows to cover. In our experiments, we use the following $\Theta_{UAF}(f_a,f_b)$ function, with a value of $\beta = 0.25$:

$$\Theta_{UAF}(f_a,f_b) \triangleq \begin{cases} \beta & \text{if } f_a \rightarrow f_b \text{ covers more than 2 UAF events in sequence} \\ 1 & \text{otherwise} \end{cases}$$

Figure 5 presents an example of call graph with edges favored using the above Θ_{UAF} function.

Finally, we combine our edge weight modification with that of Hawkeye:

$$w_{UAFuzz}(f_a,f_b) \triangleq w_{Hawkeye}(f_a,f_b).\Theta_{UAF}(f_a,f_b)$$

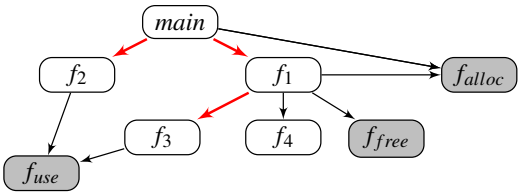

Figure 5: Example of a call graph. Favored edges are in red.

Like AFLGo, we favor the shortest path leading to the targets, since it is more likely to involve only a small number of control flow constraints, making it easier to cover by fuzzing. Our distance-based technique therefore considers both calling relations in general, via $w_{Hawkeye}$, and calling relations covering UAF events in sequence, via Θ_{UAF}.

4.4 Power Schedule

Coverage-guided fuzzers employ a power schedule (or energy assignment) to determine the number of extra inputs to generate from a selected input, which is called the *energy* of the seed. It measures how long we should spend fuzzing a particular seed. While AFL [1] mainly uses execution trace characteristics such as trace size, execution speed of the PUT and time added to the fuzzing queue for seed energy allocation, recent work [26, 48, 62] including both directed and coverage-guided fuzzing propose different power schedules. AFLGo employs simulated annealing to assign more energy for seeds closer to target locations (using the seed distance), while Hawkeye accounts for both shorter and longer traces leading to the targets via a power schedule based on trace distance and similarity at function level.

We propose here a new power schedule using the intuitions that we should assign more energy to seeds in these cases:

- seeds that are closer (using the seed distance, Section 4.3.2);
- seeds that are more similar to the target (using the target similarity, Section 4.2.2);
- *seeds that make better decisions at critical code junctions*.

We define hereafter a new metric to evaluate the latter case.

4.4.1 Cut-edge Coverage Metric

To track progress of a seed during the fuzzing process, a fine-grained approach would consist in instrumenting the execution to compare the similarity of the execution trace of the current seed with the target bug trace, at the basic block level. But this method would slow down the fuzzing process due to high runtime overhead, especially for large programs. A more coarse-grained approach, on the other hand, is to measure the similarity at function level as proposed in Hawkeye [28]. However, a callee can occur multiple times from different locations of single caller. Also, reaching a target function does not mean reaching the target basic blocks in this function.

Thus, we propose the lightweight *cut-edge coverage metric*, hitting a middle ground between the two aforementioned approaches by measuring progress *at the edge level* but *on the critical decision nodes only*.

Algorithm 3: Accumulating cut edges

Input : Program P; dynamic calling tree T of a bug trace
Output : Set of cut edges E_{cut}

```
1 E_cut ← ∅;
2 nodes ← flatten(T);
3 for n ∈ nodes ∧ pn the node before n in T do
4     if n.func == pn.func then
5         ce ← calculate_cut_edges(n.func, pn.bb, n.bb);
6     else if pn is a call to n.func then
7         ce ← calculate_cut_edges(n.func, n.func.entry_bb, n.bb);
8     E_cut ← E_cut ∪ ce;
9 return E_cut;
```

Algorithm 4: calculate_cut_edges inside a function

Input : A function f; Two basic blocks bb_{source} and bb_{sink} in f
Output : Set of cut edges ce

```
1 ce ← ∅;
2 cfg ← get_CFG(f);
3 decision_nodes ← {dn : ∃ a path bb_source →* dn →* bb_sink in cfg}
4 for dn ∈ decision_nodes do
5     outgoing_edges ← get_outgoing_edges(cfg, dn);
6     for edge ∈ outgoing_edges do
7         if reachable(cfg, edge, bb_sink) then
8             ce ← ce ∪ {edge};
9 return ce;
```

Definition 2. A *cut edge* between two basic blocks *source* and *sink* is an outgoing edge of a decision node so that there exists a path starting from *source*, going through this edge and reaching *sink*. A *non-cut edge* is an edge which is not a cut-edge, i.e. for which there is no path from *source* to *sink* that go through this edge.

Algorithm 3 shows how cut/non-cut edges are identified in UAFuzz given a tested binary program and an expected UAF bug trace. The main idea is to identify and accumulate the cut edges between all consecutive nodes in the (flattened) bug trace. For instance in the bug trace of Figure 3, we would first compute the cut edges between 0 and 1, then those between 1 and 2, etc. As the bug trace is a sequence of stack traces, most of the locations in the trace are "call" events, and we compute the cut edge from the function entry point to the call event in that function. However, because of the flattening, sometimes we have to compute the cut edges between different points in the same function (e.g. if in the bug trace the same function is calling *alloc* and *free*, we would have to compute the edge from the call to *alloc* to the call to *free*).

Algorithm 4 describes how cut-edges are computed inside a single function. First we have to collect the decision nodes, i.e. conditional jumps between the source and sink basic blocks. This can be achieved using a simple data-flow analysis. For each outgoing edge of the decision node, we check whether they allow to reach the sink basic block; those that can are cut edges, and the others are non-cut edges. Note that this

program analysis is intra-procedural, so that we do not need construct an inter-procedural CFG.

Our heuristic is that an input exercising more cut edges and fewer non-cut edges is more likely to cover more locations of the target. Let $E_{cut}(T)$ be the set of all cut edges of a program given the expected UAF bug trace T. We define the cut-edge score $e_s(s,T)$ of seed s as

$$e_s(s,T) \triangleq \sum_{e \in E_{cut}(T)} \lfloor (\log_2 hit(e)+1) \rfloor - \delta * \sum_{e \notin E_{cut}(T)} \lfloor (\log_2 hit(e)+1) \rfloor$$

where $hit(e)$ denotes the number of times an edge e is exercised, and $\delta \in (0,1)$ is the weight penalizing seeds covering non-cut edges. In our main experiments, we use $\delta = 0.5$ according to our preliminary experiments. To deal with the path explosion induced by loops, we use bucketing [1]: the hit count is bucketized to small powers of two.

4.4.2 Energy Assignment

We propose a power schedule function that assigns energy to a seed using a combination of the three metrics that we have proposed: the prefix target similarity metric $t_P(s,T)$ (Section 4.2.2), the UAF-based seed distance $d(s,T)$ (Section 4.3.2), and the cut-edge coverage metric $e_s(s,T)$ (Section 4.4.1). The idea of our power schedule is to assign energy to a seed s proportionally to the number of targets covered in sequence $t_P(s,T)$, with a corrective factor based on seed distance d and cut-edge coverage e_s. Indeed, our power function (corresponding to ASSIGN_ENERGY in Algorithm 1) is defined as:

$$p(s,T) \quad \triangleq \quad (1+t_P(s,T)) \times \tilde{e}_s(s,T) \times (1-\tilde{d}_s(s,T))$$

Because their actual value is not as meaningful as the length of the covered prefix, but they allow to rank the seeds, we apply a min-max normalization [2] to get a *normalized seed distance* ($\tilde{d}_s(s,T)$) and *normalized cut-edge score* ($\tilde{e}_s(s,T)$). For example, $\tilde{d}_s(s,T) = \frac{d_s(s,T)-minD}{maxD-minD}$ where *minD, maxD* denote the minimum and maximum value of seed distance so far. Note that both metric scores are in (0, 1), i.e. can only reduce the assigned energy when their score is bad.

4.5 Postprocess and Bug Triage

Since UAF bugs are often silent, all seeds generated by a directed fuzzer must *a priori* be sent to a *bug triager* (typically, a profiling tool such as VALGRIND) in order to confirm whether they are bug triggering input or not. Yet, this can be extremely expensive as fuzzers generate a huge amount of seeds and bug triagers are expensive.

Fortunately, the target similarity metric allows UAFUZZ to compute the sequence of covered targets of each fuzzed input at runtime. This information is *available for free* for each seed once it has been created and executed. We capitalize on it in order to *pre-identify* likely-bug triggering seeds, i.e. seeds that indeed cover the three UAF events in sequence. Then, the bug triager is run only over these pre-identified seeds, the other ones being simply discarded – potentially saving a huge amount of time in bug triaging.

5 Implementation

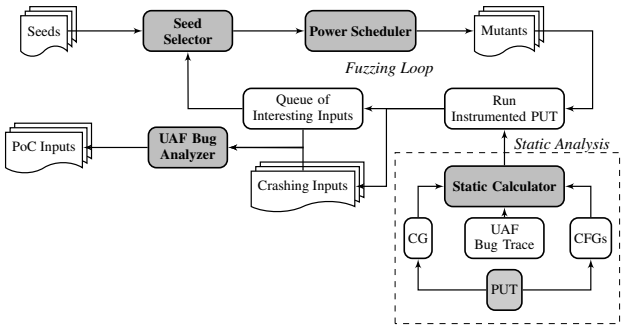

Figure 6: Overview of UAFUZZ workflow.

We implement our results in a UAF-oriented binary-level directed fuzzer named UAFUZZ. Figure 6 depicts an overview of the main components of UAFUZZ. The input of the overall system are a set of initial seeds, the PUT in binary and target locations extracted from the bug trace. The output is a set of unique bug-triggering inputs. The prototype is built upon AFL 2.52b [1] and QEMU 2.10.0 for fuzzing, and the binary analysis platform BINSEC [4] for (lightweight) static analysis. These two components share information such as target locations, time budget and fuzzing status.

6 Experimental Evaluation

6.1 Research Questions

To evaluate the effectiveness and efficiency of our approach, we investigate four principal research questions:

RQ1. UAF Bug-reproducing Ability Can UAFUZZ outperform other directed fuzzing techniques in terms of UAF bug reproduction in executables?

RQ2. UAF Overhead How does UAFUZZ compare to other directed fuzzing approaches w.r.t. instrumentation time and runtime overheads?

RQ3. UAF Triage How much does UAFUZZ reduce the number of inputs to be sent to the bug triage step?

RQ4. Individual Contribution How much does each UAFUZZ component contribute to the overall results?

We will also evaluate UAFUZZ in the context of ***patch testing***, another important application of directed fuzzing [25, 28, 59].

6.2 Evaluation Setup

Evaluation Fuzzers. We aim to compare UAFUZZ with state-of-the-art directed fuzzers, namely AFLGO [2] and HAWKEYE [28], using AFL-QEMU as a baseline (binary-level coverage-based fuzzing). Unfortunately, both AFLGO and HAWKEYE work on source code, and while AFLGO is open source, HAWKEYE is not available. Hence, we *implemented binary-level versions* of AFLGO and HAWKEYE, coined as

AFLGoB and HawkeyeB. We closely follow the original papers, and, for AFLGo, use the source code as a reference. AFLGoB and HawkeyeB are implemented on top of AFL-QEMU, following the generic architecture of UAFuzz but with dedicated distance, seed selection and power schedule mechanisms. Table 2 summarizes our different fuzzer implementations and a comparison with their original counterparts.

Table 2: Overview of main techniques of greybox fuzzers. Our own implementations are marked with ⋆.

Fuzzer	Directed	Binary?	Distance	Seed Selection	Power Schedule	Mutation
AFL-QEMU	✗	✓	–	AFL	AFL	AFL
AFLGo	✓	✗	CFG-based	∼ AFL	Annealing	∼ AFL
AFLGoB⋆	✓	✓	∼ AFLGo	∼ AFLGo	∼ AFLGo	∼ AFLGo
Hawkeye	✓	✗	AAFD	distance-based	Trace fairness	Adaptive
HawkeyeB⋆	✓	✓	∼ Hawkeye	∼ Hawkeye	≈ Hawkeye	∼ AFLGo
UAFuzz⋆	✓	✓	UAF-based	Targets-based	UAF-based	∼ AFLGo

We evaluate the implementation of AFLGoB (Appendix B, Appendix) and find it very close to the original AFLGo after accounting for emulation overhead.

UAF Fuzzing Benchmark. The standard UAF micro benchmark Juliet Test Suite [56] for static analyzers is too simple for fuzzing. No macro benchmark actually assesses the effectiveness of UAF detectors – the widely used LAVA [36] only contains buffer overflows. Thus, we construct a new UAF benchmark according to the following rationale:

1. The subjects are real-world popular and fairly large security-critical programs;
2. The benchmark includes UAF bugs found by existing fuzzers from the fuzzing litterature [1, 26, 28, 40] or collected from NVD [20]. Especially, we include *all* UAF bugs found by directed fuzzers;
3. The bug report provides detailed information (e.g., buggy version and the stack trace), so that we can identify target locations for fuzzers.

In summary, we have 13 known UAF vulnerabilities (2 from directed fuzzers) over 11 real-world C programs whose sizes vary from 26 Kb to 3.8 Mb. Furthermore, selected programs range from image processing to data archiving, video processing and web development. Our benchmark is therefore representative of different UAF vulnerabilities of real-world programs. Table 3 presents our evaluation benchmark.

Evaluation Configurations. We follow the recommendations for fuzzing evaluations [43] and use the same fuzzing configurations and hardware resources for all experiments. Experiments are conducted 10 times with a time budget depending on the PUT. We use as input seed either an empty file or existing valid files provided by developers. We do not use any token dictionary. All experiments were carried out on an Intel Xeon CPU E3-1505M v6 @ 3.00GHz CPU with 32GB RAM and Ubuntu 16.04 64-bit.

6.3 UAF Bug-reproducing Ability (RQ1)

Protocol. We compare the different fuzzers on our 13 UAF vulnerabilities using *Time-to-Exposure* (TTE), i.e. the time elapsed until first bug-triggering input, and *number of success runs* in which a fuzzer triggers the bug. In case a fuzzer cannot detect the bug within the time budget, the run's TTE is set to the time budget. Following existing work [25, 28], we use the *Vargha-Delaney statistic* ($\hat{A}_{12}$) metric [69] [6] to assess the confidence that one tool outperforms another. Code coverage is not relevant for directed fuzzers.

Table 3: Overview of our evaluation 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

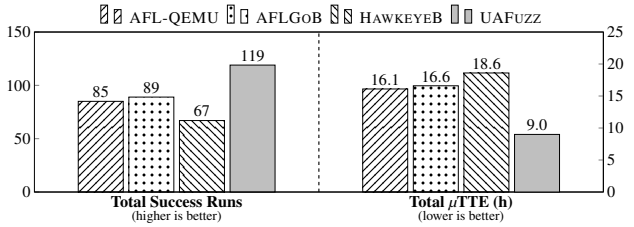

Figure 7: Summary of fuzzing performance (RQ1)

Results. Figure 7 presents a consolidated view of the results (total success runs and TTE – we denote by μTTE the average TTE observed for each sample over 10 runs). Appendix A contains additional information about consolidated Vargha-Delaney statistics (Table 4).

Figure 7 (and Table 4) show that UAFuzz clearly outperforms the other fuzzers both in total success runs (vs. 2nd best AFLGoB: +34% in total, up to +300%) and in TTE (vs. 2nd best AFLGoB, total: 2.0×, avg: 6.7×, max: 43×). In some specific cases,UAFuzz saves roughly 10,000s of TTE over AFLGoB or goes from 0/10 successes to 7/10. The $\hat{A}_{12}$ value of UAFuzz against other fuzzers is also significantly above the conventional large effect size 0.71 [69], as shown in Table 4 (vs. 2nd best AFLGoB, avg: 0.78, median: 0.80, min: 0.52).

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

Note that performance of AFLGoB and HawkeyeB w.r.t. their original source-level counterparts are representative (cf. Appendix B).

[6] Value between 0 and 1, the higher the better. Values above the conventionally large effect size of 0.71 are considered highly relevant [69].

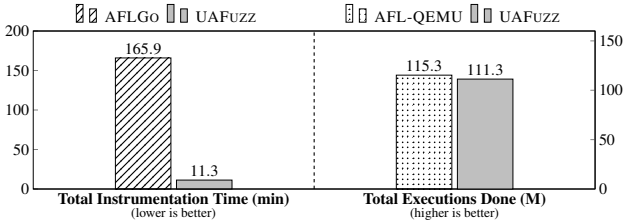

Figure 8: Global overhead (RQ2)

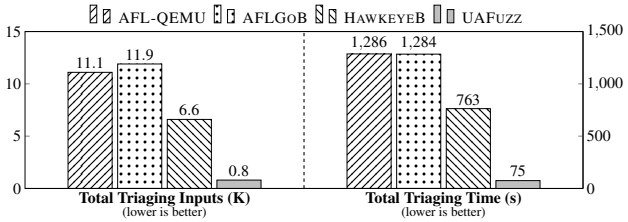

Figure 9: Summary of bugs triage (RQ3)

6.4 UAF Overhead (RQ2)

Protocol. We are interested in both (1) instrumentation-time overhead and (2) runtime overhead. For (1), we simply compute the total instrumentation time of UAFuzz and we compare it to the instrumentation time of AFLGo. For (2), we compute the total number of executions per second of UAFuzz and compare it AFL-QEMU taken as a baseline.

Results. Consolidated results for both instrumentation-time and runtime overhead are presented in Figure 8 (number of executions per second is replaced by the total number of executions performed in the same time budget). This figure shows that UAFuzz is *an order of magnitude faster than the state-of-the-art source-based directed fuzzer* AFLGo *in the instrumentation phase*, and has almost the same total number of executions per second as AFL-QEMU. Appendix C contains additional results with detailed instrumentation time (Figure 12) and runtime statistics (Figure 14), as well as instrumentation time for AFLGoB and HawkeyeB (Figure 13).

UAFuzz enjoys both a *lightweight instrumentation time* and a *minimal runtime overhead.*

6.5 UAF Triage (RQ3)

Protocol. We consider the total number of triaging inputs (number of inputs sent to the triaging step), the triaging inputs rate TIR (ratio between the total number of generated inputs and those sent to triaging) and the total triaging time (time spent within the triaging step). Since other fuzzers cannot identify inputs reaching targets during the fuzzing process, we conservatively analyze *all* inputs generated by the these fuzzers in the bug triage step (TIR = 1).

Results. Consolidated results are presented in Figure 9, detailed results in Appendix D, Table 6 and Figure 15.

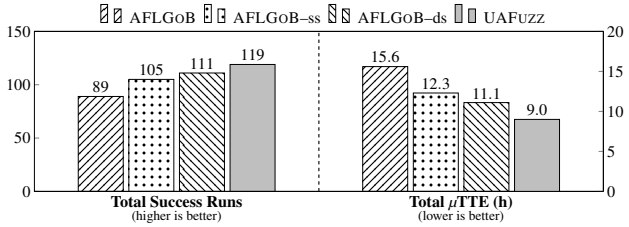

Figure 10: Impact of each components (RQ4)

- The TIR of UAFuzz is 9.2% in total (avg: 7.25%, median: 3.14%, best: 0.24%, worst: 30.22%) – sparing up to 99.76% of input seeds for confirmation, and is always less than 9% except for sample `mjs`;
- Figure 15 shows that UAFuzz spends the smallest amount of time in bug triage, i.e. 75s (avg: 6s, min: 1s, max: 24s) for a total speedup of 17× over AFLGoB (max: 130×, avg: 39×).

UAFuzz reduces a *large* portion (i.e., more than 90%) of triaging inputs in the post-processing phase. Subsequently, UAFuzz only spends several seconds in this step, winning an order of magnitude compared to standard directed fuzzers.

6.6 Individual Contribution (RQ4)

Protocol. We compare four different versions of our prototype, representing a continuum between AFLGo and UAFuzz: (1) the basic AFLGo represented by AFLGoB, (2) AFLGoB–ss adds our seed selection metric to AFLGoB, (3) AFLGoB–ds adds the UAF-based function distance to AFLGoB–ss, and finally (4) UAFuzz adds our dedicated power schedule to AFLGoB–ds. We consider the previous RQ1 metrics: number of success runs, TTE and Vargha-Delaney. Our goal is to assess whether or not these technical improvements do lead to fuzzing performance improvements.

Results. Consolidated results for success runs and TTE are represented in Figure 10. Appendix E includes detailed results plus Vargha-Delaney metric (Table 7).

As summarized in Figure 10, we can observe that each new component does improve both TTE and number of success runs, leading indeed to fuzzing improvement. Detailed results in Table 7 with $\hat{A}_{12}$ values show the same clear trend.

The UAF-based distance computation, the power scheduling and the seed selection heuristic *individually* contribute to improve fuzzing performance, and combining them yield even further improvements, demonstrating their interest and complementarity.

6.7 Patch Testing & Zero-days

Patch testing. The idea is to use bug stack traces of *known* UAF bugs to guide testing on the *patched* version of the PUT

– instead of the buggy version as in bug reproduction. The benefit from the bug hunting point of view [17] is both to try finding buggy or incomplete patches *and* to focus testing on *a priori* fragile parts of the code, possibly discovering bugs unrelated to the patch itself.

How to. We follow bug hunting practice [17]. Starting from the recent publicly disclosed UAF bugs of open source programs, we manually identify addresses of relevant call instructions in the reported bug stack traces since the code has been evolved. We focus mainly on 3 widely-used programs that have been well fuzzed and maintained by the developers, namely GNU patch, GPAC and Perl 5 (737K lines of C code and 5 known bug traces in total). We also consider 3 other codes: MuPDF, Boolector and fontforge (+1,196Kloc).

Results. Overall UAFuzz has found and reported **30 new bugs**, including **11 new UAF bugs** and **7 new CVE** (details in Appendix F, Table 8). *At this time, 17 bugs have been fixed by the vendors*. Interestingly, the bugs found in GNU patch (Appendix F) and GPAC were actually *buggy patches*.

> UAFuzz has been proven effective in a patch testing setting, allowing to find 30 new bugs (incl. 7 new CVE) in 6 widely-used programs.

6.8 Threats to Validity

Implementation. Our prototype is implemented as part of the binary-level code analysis framework BINSEC [34, 35], whose efficiency and robustness have been demonstrated in prior large scale studies on both adversarial code and managed code [22,33,63], and on top of the popular fuzzer AFL-QEMU. Effectiveness and correctness of UAFuzz have been assessed on several bug traces from real programs, as well as on small samples from the Juliet Test Suite. All reported UAF bugs have been manually checked.

Benchmark. Our benchmark is built on both real codes *and* real bugs, and encompass several bugs found by recent fuzzing techniques of well-known open source codes (including all UAF bugs found by directed fuzzers).

Competitors. We consider the best state-of-the-art techniques in directed fuzzing, namely AFLGo [25] and Hawkeye [28]. Unfortunately, Hawkeye is not available and AFLGo works on source code only. Thus, we re-implement these technologies in our own framework. We followed the available information (article, source code if any) as close as possible, and did our best to get precise implementations. They have both been checked on real programs and small samples, and the comparison against AFLGo source (Appendix B) and our own AFLGoB implementation is conclusive.

7 Related Work

Directed Greybox Fuzzing. AFLGo [25] and Hawkeye [28] have already been discussed. LOLLY [49] provides a lightweight instrumentation to measure the sequence basic block coverage of inputs, yet, at the price of a large runtime overhead. SeededFuzz [71] seeks to generate a set of initial seeds that improves directed fuzzing performance. SemFuzz [74] leverages vulnerability-related texts such as CVE reports to guide fuzzing. 1dVul [59] discovers 1-day vulnerabilities via binary patches.

UAFuzz *is the first directed fuzzer tailored to UAF bugs, and one of the very few [59] able to handle binary code.*

Coverage-based Greybox Fuzzing. AFL [1] is the seminal coverage-guided greybox fuzzer. Substantial efforts have been conducted in the last few years to improve over it [26, 40, 46]. Also, many efforts have been fruitfully invested in combining fuzzing with other approaches, such as static analysis [40, 47], dynamic taint analysis [29, 30, 62], symbolic execution [58, 67, 76] or machine learning [41, 66].

Recently, UAFL [70] - another independent research effort on the same problem, specialized coverage-guided fuzzing to detect UAFs by finding operation sequences potentially violating a typestate property and then guiding the fuzzing process to trigger property violations. However, this approach relies heavily on the static analysis of source code, therefore is not applicable at binary-level.

Our technique is orthogonal to all these improvements, they could be reused within UAFuzz *as is.*

UAF Detection. Precise static UAF detection is difficult. GUEB [12] is the only binary-level static analyzer for UAF. The technique can be combined with dynamic symbolic execution to generate PoC inputs [38], yet with scalability issues. On the other hand, several UAF source-level static detectors exist, based on abstract interpretation [32], pointer analysis [72], pattern matching [57], model checking [44] or demand-driven pointer analysis [68]. A common weakness of all static detectors is their inability to infer triggering input – they rather prove their absence.

Dynamic UAF detectors mainly rely on heavyweight instrumentation [9, 27, 55] and result in high runtime overhead, even more for closed source programs. ASan [65] performs lightweight instrumentation, but at source level only.

UAF Fuzzing Benchmark. While the Juliet Test Suite [56] (CWE-415, CWE-416)[7] contains only too small programs, popular fuzzing benchmarks [7, 11, 16, 36, 64] comprise only very few UAF bugs. Moreover, many of these benchmarks contain either artificial bugs [7, 16, 36, 64] or artificial programs [56].

Merging our evaluation benchmark (known UAF) and our new UAF bugs, we provide the largest fuzzing benchmark dedicated to UAF – 17 real codes and 30 real bugs

8 Conclusion

UAFuzz is the *first directed* greybox fuzzing approach tailored to detecting UAF vulnerabilities (in binary) given only the bug stack trace. UAFuzz outperforms existing directed fuzzers, both in terms of time to bug exposure and number of

[7] Juliet is mostly used for the evaluation of C/C++ static analysis tools.

successful runs. UAFuzz has been proven effective in both bug reproduction and patch testing. We release the source code of UAFuzz and the UAF fuzzing benchmark at:

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

Acknowledgement

This work was supported by the H2020 project C4IIoT under the Grant Agreement No 833828 and FUI CAESAR.

References

[1] Afl. http://lcamtuf.coredump.cx/afl/, 2020.

[2] Aflgo. https://github.com/aflgo/aflgo, 2020.

[3] Aflgo's issues. https://github.com/aflgo/aflgo/issues, 2020.

[4] Binsec. https://binsec.github.io/, 2020.

[5] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2020.

[6] Cve-2018-6952. https://savannah.gnu.org/bugs/index.php?53133, 2020.

[7] Darpa cgc corpus. http://www.lungetech.com/2017/04/24/cgc-corpus/, 2020.

[8] Double free in gnu patch. https://savannah.gnu.org/bugs/index.php?56683, 2020.

[9] Dr.memory. https://www.drmemory.org/, 2020.

[10] Gnu patch. https://savannah.gnu.org/projects/patch/, 2020.

[11] Google fuzzer testsuite. https://github.com/google/fuzzer-test-suite, 2020.

[12] Gueb: Static analyzer detecting use-after-free on binary. https://github.com/montyly/gueb, 2020.

[13] Libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2020.

[14] OSS-Fuzz: Continuous Fuzzing Framework for Open-Source Projects. https://github.com/google/oss-fuzz/, 2020.

[15] Oss-fuzz: Five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2020.

[16] Rode0day. https://rode0day.mit.edu/, 2020.

[17] Sockpuppet: A walkthrough of a kernel exploit for ios 12.4. https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html, 2020.

[18] Uaf fuzzing benchmark. https://github.com/strongcourage/uafbench, 2020.

[19] Uafuzz. https://github.com/strongcourage/uafuzz, 2020.

[20] Us national vulnerability database. https://nvd.nist.gov/vuln/search, 2020.

[21] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[22] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. 2017.

[23] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: synthesizing structure while fuzzing. In *USENIX Security Symposium (USENIX Security 19)*, 2019.

[24] Marcel Böhme. Assurance in software testing: A roadmap. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '19, 2019.

[25] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017.

[26] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[27] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.

[28] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[29] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[30] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. *arXiv preprint arXiv:1905.12228*, 2019.

[31] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[32] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*, 2012.

[33] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA*, 2016.

[34] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, 2016.

[35] Adel Djoudi and Sébastien Bardin. Binsec: Binary code analysis with low-level regions. In *TACAS*, 2015.

[36] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, 2016.

[37] Andrew Fasano, Tim Leek, Brendan Dolan-Gavitt, and Josh Bundt. The rode0day to less-buggy programs. *IEEE Security & Privacy*, 2019.

[38] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 2016.

[39] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, 2020.

[40] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[41] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[42] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[44] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[45] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[46] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[47] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[48] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[49] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proceedings of the 27th International Conference on Program Comprehension*, 2019.

[50] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[51] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[52] Microsoft. Project springfield. https://www.microsoft.com/en-us/security-risk-detection/, 2020.

[53] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 1990.

[54] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[55] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan*, 2007.

[56] NIST. Juliet test suite for c/c++. https://samate.nist.gov/SARD/testsuite.php, 2020.

[57] Mads Chr Olesen, René Rydhof Hansen, Julia L Lawall, and Nicolas Palix. Coccinelle: tool support for automated cert c secure coding standard certification. *Science of Computer Programming*, 2014.

[58] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[59] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[60] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[61] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[62] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[63] Frédéric Recoules, Sébastien Bardin, Richard Bonichon Bonichon, Laurent Mounier, and Marie-Laure Potet. Get rid of inline assembly through verification-oriented lifting. In *ASE*, 2019.

[64] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[65] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[66] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[68] Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[69] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 2000.

[70] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *42nd International Conference on Software Engineering*, 2020.

[71] Weiguang Wang, Hao Sun, and Qingkai Zeng. Seededfuzz: Selecting and generating seeds for directed fuzzing. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2016.

[72] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

[73] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[74] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[75] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[76] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

A UAF Bug-reproducing Ability (RQ1)

We present in this section additional results regarding RQ1 including more detailed experimental reports.

Experimental results. Table 4 summarizes the fuzzing performance of 4 binary-based fuzzers against our benchmark by providing the total number of covered paths, the total number of success runs and the max/min/average/median values of *Factor* and $\hat{A}_{12}$. Table 5 compares our fuzzer UAFuzz with several variants of directed fuzzers AFLGo.

Table 4: Summary of bug reproduction of UAFuzz compared to other fuzzers against our fuzzing benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold.

Fuzzer	Total Avg Paths	Success Runs	Factor				$\hat{A}_{12}$			
			Mdn	Avg	Min	Max	Mdn	Avg	Min	Max
AFL-QEMU	10.6K	85 (+40%)	2.01	6.66	0.60	46.63	**0.82**	**0.78**	0.29	**1.00**
AFLGoB	11.1K	89 (+34%)	1.96	6.73	0.96	43.34	**0.80**	**0.78**	0.52	**1.00**
HawkeyeB	7.3K	67 (+78%)	2.90	8.96	1.21	64.29	**0.88**	**0.86**	0.56	**1.00**
UAFuzz	8.2K	119	–	–	–	–	–	–	–	–

Table 5: Bug reproduction of AFLGo against our benchmark except `CVE-2017-10686` due to compilation issues of AFLGo. Numbers in red are the best μTTEs.

Bug ID	AFLGo (source)		$AFLGo_F$ (source)		AFLGoB		UAFuzz	
	Runs	μTTE(s)	Runs	μTTE(s)	Runs	μTTE(s)	Runs	μTTE(s)
`giflib-bug-74`	10	62	10	281	9	478	10	209
`CVE-2018-11496`	10	2	10	38	10	22	10	14
`yasm-issue-91`	10	307	8	2935	8	2427	10	56
`CVE-2016-4487`	10	676	10	1386	6	2427	6	2110
`CVE-2018-11416`	10	78	7	1219	10	303	10	235
`mjs-issue-78`	10	1417	3	9706	4	8755	9	4197
`mjs-issue-73`	9	5207	3	34210	0	10800	7	4881
`CVE-2018-10685`	10	74	9	1072	9	305	10	156
`CVE-2019-6455`	5	1090	0	20296	5	1213	10	438
`gifsicle-issue-122`	8	4161	7	25881	6	9811	7	9853
`CVE-2016-3189`	10	72	10	206	10	158	10	141
`CVE-2018-20623`	10	177	10	1329	9	3169	10	128
Total Success Runs	**112**		**87**		**86**		**109**	
Total μTTE (h)	**3.7**		**27.4**		**10.1**		**6.2**	

B Regarding implementations of AFLGoB and HawkeyeB

Comparison between AFLGoB and source-based AFLGo. We want to evaluate how close our implementation of AFLGoB is from the original AFLGo, in order to assess the degree of confidence we can have in our results – we do not do it for HawkeyeB as Hawkeye is not available.

AFLGo unsurprisingly performs better than AFLGoB and UAFuzz (Figure 11, Table 5 in Appendix). This is largely due to the emulation runtime overhead of QEMU, a well-documented fact. Still, *surprisingly enough*, UAFuzz can find the bugs faster than AFLGo in 4 samples, demonstrating its efficiency.

Yet, more interestingly, Figure 11 also shows that once emulation overhead [8] is taken into account (yielding $AFLGo_F$, the expected *binary-level* performance of AFLGo), then AFLGoB is in line with $AFLGo_F$ (and even shows better TTE) – UAFuzz even significantly outperforms $AFLGo_F$.

[8] We estimate for each sample an overhead factor f by comparing the number of executions per second in both AFL and AFL-QEMU, then multiply the computation time of AFLGo by f – f varies from 2.05 to 22.5 in our samples.

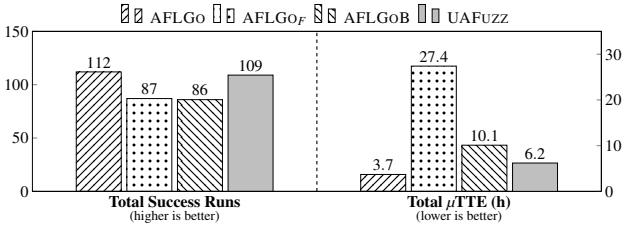

Figure 11: Summary of fuzzing performance of 4 fuzzers against our benchmark, except `CVE-2017-10686` due to compilation issues of AFLGo.

Performance of AFLGoB is in line with the original AFLGo once QEMU overhead is taken into account, allowing a fair comparison with UAFuzz. UAFuzz nonetheless performs relatively well on UAF compared with the source-based directed fuzzer AFLGo, demonstrating the benefit of our original fuzzing mechanisms.

About performance of HawkeyeB in RQ1. HawkeyeB performs significantly worse than AFLGoB and UAFuzz in §6.3. We cannot compare HawkeyeB with Hawkeye as Hawkeye is not available. Still, we investigate that issue and found that this is mostly due to a large runtime overhead spent calculating the target similarity metric. Indeed, according to the Hawkeye original paper [28], this computation involves some *quadratic computation* over the *total number of functions* in the code under test. On our samples this number quickly becomes important (up to 772) while the number of targets (UAFuzz) remains small (up to 28). A few examples: `CVE-2017-10686`: 772 functions vs 10 targets; `gifsicle-issue-122`: 516 functions vs 11 targets; `mjs-issue-78`: 450 functions vs 19 targets. Hence, we can conclude that on our samples the performance of HawkeyeB are in line with what is expected from Hawkeye algorithm.

C UAF Overhead (RQ2)

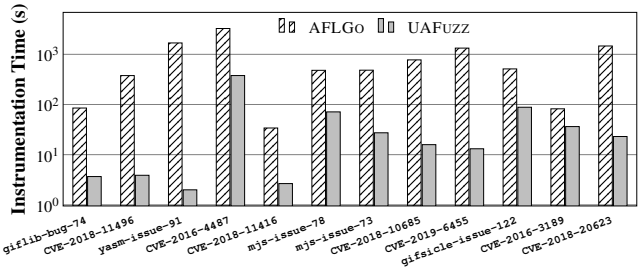

Figure 12: Average instrumentation time in seconds (except `CVE-2017-10686` due to compilation issues of AFLGo).

Additional data. We first provide additional results for RQ2. Figures 12 and 13 compare the average instrumentation time between, respectively, UAFuzz and the source-based directed

fuzzer AFLGo; and UAFuzz and the two binary-based directed fuzzers AFLGoB and HawkeyeB. Figure 14 shows the total execution done of AFL-QEMU and UAFuzz for each subject in our benchmark.

Detailed results. We now discuss experimental results regarding overhead in more depth than what was done in §6.4.

- Figures 8 and 12 show that UAFuzz is *an order of magnitude faster than the state-of-the-art source-based directed fuzzer* AFLGo *in the instrumentation phase* (14.7× faster in total). For example, UAFuzz spends only 23s (i.e., 64× less than AFLGo) in processing the large program `readelf` of Binutils;
- Figures 8 and 14 show that UAFuzz has almost the same total number of executions per second as AFL-QEMU (-4% in total, -12% in average), meaning that its overhead is negligible.
- Figure 13 shows that HawkeyeB is sometimes significantly slower than UAFuzz (2×). This is mainly because of the cost of target function trace closure calculation on large examples with many functions (cf. §6.3).

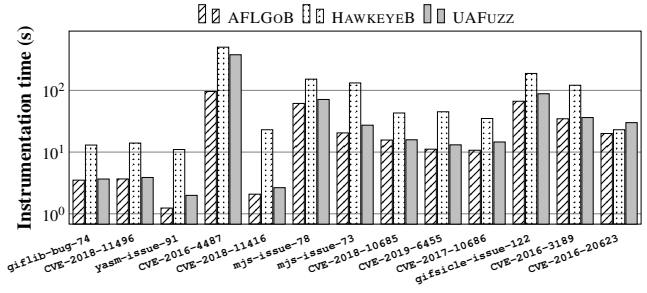

Figure 13: Average instrumentation time in seconds.

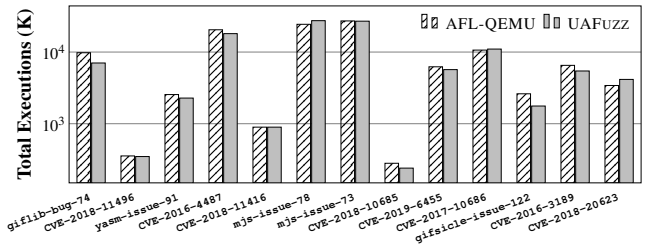

Figure 14: Total executions done in all runs.

D UAF Triage (RQ3)

We provide additional results for RQ3: Figure 15 and Table 6 show the average triaging time and number of triaging inputs (including TIR values for UAFuzz) of 4 fuzzers against our benchmark.

E Individual Contribution (RQ4)

We provide additional results for RQ4. Table 7 shows the fuzzing performance of 2 AFLGoB-based variants AFLGoB–ss and AFLGoB–ds compared to AFLGoB and our tool UAFuzz against our benchmark.

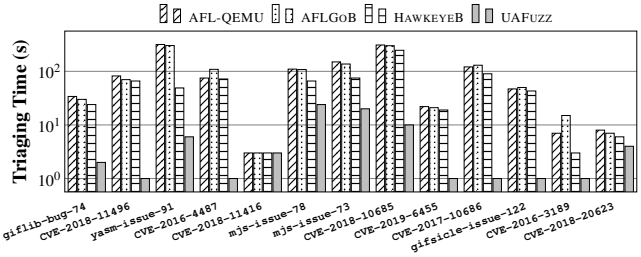

Figure 15: Average triaging time in seconds.

Table 6: Average number of triaging inputs of 4 fuzzers against our tested subjects. For UAFuzz, the TIR values are in parentheses.

Bug ID	AFL-QEMU	AFLGoB	HawkeyeB	UAFuzz
giflib-bug-74	200.9	177.0	139.9	10.0 (5.31%)
CVE-2018-11496	409.6	351.7	332.5	5.4 (4.08%)
yasm-issue-91	2115.3	2023.0	326.6	37.4 (2.72%)
CVE-2016-4487	933.1	1367.2	900.2	2.5 (0.24%)
CVE-2018-11416	21.5	21.0	21.0	1.0 (4.76%)
mjs-issue-78	1226.9	1537.8	734.6	262.3 (30.22%)
mjs-issue-73	1505.6	1375.9	745.6	252.2 (29.25%)
CVE-2018-10685	414.2	402.1	328.9	12.6 (3.14%)
CVE-2019-6455	243.2	238.1	211.1	6.9 (1.57%)
CVE-2017-10686	2416.9	2517.0	1765.2	214.3 (8.96%)
gifsicle-issue-122	405.0	431.7	378.5	3.3 (0.86%)
CVE-2016-3189	377.9	764.7	126.4	7.1 (1.69%)
CVE-2018-20623	804.0	724.2	625.1	5.4 (1.39 %)
Total	**11.1K**	**11.9K**	**6.6K**	**820 (7.25%)**

Table 7: Bug reproduction on 4 fuzzers against our benchmark. $\hat{A}_{12A}$ and $\hat{A}_{12U}$ denote the Vargha-Delaney values of AFLGoB and UAFuzz. Statistically significant results for $\hat{A}_{12}$ (e.g., $\hat{A}_{12A} \leq 0.29$ or $\hat{A}_{12U} \geq 0.71$) are in bold.

Fuzzers	AFLGoB	AFLGoB–ss	AFLGoB–ds	UAFuzz
Total Success Runs	89	105 (+18.0%)	111 (+24.7%)	119 (+33.7%)
Total μTTE (h)	15.6	12.3	11.1	9.0
Average $\hat{A}_{12A}$	–	**0.29**	0.37	**0.22**
Average $\hat{A}_{12U}$	**0.78**	0.54	0.64	–

F Patch Testing & Zero-days

We provide additional results for patch testing (Table 8), as well as a **detailed discussion on the GNU Patch buggy patch**.

Zoom: GNU Patch buggy patch. We use `CVE-2018-6952` [6] to demonstrate the effectiveness of UAFuzz in exposing unknown UAF vulnerabilities. GNU patch [10] takes a patch file containing a list of differences and applies them to the original file. Listing 3 shows the code fragment of `CVE-2018-6952` which is a double free in the latest version 2.7.6 of GNU patch. Interestingly, by using the stack trace of this CVE as shown in Figure 16, UAFuzz successfully discovered an *incomplete bug fix* [8] `CVE-2019-20633` in the latest commit `76e7758`, with a slight difference of the bug stack trace (i.e., the call of `savebuf()`

```
1  File: src/patch.c
2  int main (int argc, char **argv) {...
3    while (0 < (got_hunk = another_hunk (diff_type, reverse
         ))) { /* Apply each hunk of patch */ ... }
4  ...}
5
6  File: src/pch.c
7  int another_hunk (enum diff difftype, bool rev) { ...
8    while (p_end >= 0) {
9      if (p_end == p_efake) p_end = p_bfake;
10     else free(p_line[p_end]); /* Free and Use event */
11     p_end--;
12   } ...
13   while (p_end < p_max) { ...
14     switch(*buf) { ...
15     case '+': case '!': /* Our bug CVE-2019-20633 */ ...
16       p_line[p_end] = savebuf (s, chars_read); ...
17     case ' ': /* CVE-2018-6952 */ ...
18       p_line[p_end] = savebuf (s, chars_read); ...
19     ...}
20   ...}
21 ... }
22
23 File: src/util.c
24 /* Allocate a unique area for a string. */
25 char *savebuf (char const *s, size_t size) { ...
26   rv = malloc (size); /* Alloc event */ ...
27   memcpy (rv, s, size);
28   return rv;
29 }
```

Listing 3: Code fragment of GNU patch pertaining to the UAF vulnerability `CVE-2018-6952`.

in `another_hunk()`).

Technically, GNU patch takes an input patch file containing multiple hunks (line 3) that are split into multiple strings using special characters as delimiter via `*buf` in the switch case (line 14). GNU patch then reads and parses each string stored in `p_line` that is dynamically allocated on the memory using `malloc()` in `savebuf()` (line 26) until the last line of this hunk has been processed. Otherwise, GNU patch deallocates the most recently processed string using `free()` (line 10). Our reported bug and `CVE-2018-6952` share the same *free* and *use* event, but have a different stack trace leading to the same *alloc* event. Actually, while the PoC input generated by UAFUZZ contains two characters '`!`', the PoC of `CVE-2018-6952` does not contain this character, consequently the case in line 16 was previously uncovered, and thus this CVE had been incompletely fixed. This case study shows the importance of producing different unique bug-triggering inputs to favor the repair process and help complete bug fixing.

```
==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)
```

Figure 16: The bug trace of CVE-2018-6952 produced by VALGRIND.

Table 8: Summary of zero-day vulnerabilities reported by our fuzzer.

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		