HTFuzz: Heap Operation Sequence Sensitive Fuzzing

Yuanping Yu[1,2†], Xiangkun Jia[1†], Yuwei Liu[1,2], Yanhao Wang[4],
Qian Sang[1,2], Chao Zhang[5,6] and Purui Su[1,3*]

[1]TCA/SKLCS, Institute of Software, Chinese Academy of Sciences [2]University of Chinese Academy of Sciences
[3]School of Cyber Security, University of Chinese Academy of Sciences [4]NIO Security Research
[5]Tsinghua University, Beijing National Research Center for Information Science and Technology [6]Zhongguancun Lab
{yuanping2017, xiangkun, yuwei2018, purui}@iscas.ac.cn chaoz@tsinghua.edu.cn wangyanhao136@gmail.com
Beijing, China

ABSTRACT

Heap-based temporal vulnerabilities (i.e., use-after-free, double-free and null pointer dereference) are highly sensitive to heap operation (e.g., memory allocation, deallocation and access) sequences. To efficiently find such vulnerabilities, traditional code coverage-guided fuzzing solutions could be promoted by integrating heap operation sequence feedback. But current sequence sensitive solutions have limitations in practice.

In this paper, we propose a novel fuzzing solution named HTFuzz, to find heap-based temporal vulnerabilities. At the core, we utilize fuzzing to increase the coverage of runtime heap operation sequences and the diversity of pointers accessed by these operations, where the former reflects the control-flow and the latter reflects the data-flow of heap operation sequences. With such increases, the fuzzer could find more heap-based temporal vulnerabilities. We have developed a prototype of HTFuzz and evaluated it on 14 real-world applications, and compared it with 11 state-of-the-art fuzzers. The results showed that, HTFuzz outperformed all the baselines and was statistically better on the number of heap-based temporal vulnerabilities discovered. In detail, HTFuzz found (1.82x, 2.62x, 2.66x, 2.02x, 2.21x, 2.06x, 1.47x, 2.98x, 1.98x) more heap operation sequences and (1.45x, 3.56x, 3.56x, 4.57x, 1.78x, 1.78x, 1.68x, 4.00x, 1.45x) more 0day heap-based temporal vulnerabilities than (AFL, AFL-sensitive-ma, AFL-sensitive-mw, Memlock, PathAFL, TortoiseFuzz, MOPT, Angora, Ankou), respectively. HTFuzz discovered 37 new vulnerabilities with 37 CVEs assigned, including 32 new heap-based temporal vulnerabilities and 5 of other types.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

heap-based temporal vulnerability, fuzzing, heap operation sequence

*corresponding author. † co-leading authors.

ACM Reference Format:
Yuanping Yu, Xiangkun Jia, Yuwei Liu, Yanhao Wang, Qian Sang, Chao Zhang and Purui Su. 2022. HTFuzz: Heap Operation Sequence Sensitive Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3560415

1 INTRODUCTION

Memory corruptions remain a significant threat to software systems [1, 39]. Fuzzing is a widely utilized technique to find memory corruptions and has made great progress in recent years [24]. Existing fuzzers in general rely on code coverage feedback to guide the fuzzing campaign. For instance, AFL [52] utilizes code coverage feedback to explore more code, while TortoiseFuzz [43] and Ankou [25] try to explore more error-prone operations (e.g., system calls and loop iterations) as supplement feedbacks.

However, heap-based temporal vulnerabilities (HT-Vuls) – one of the most common types of memory corruptions – are sensitive to not only code coverage but also heap operation sequences. For example, a pointer dereference may turn into a null pointer dereference (NPD) or a use after free (UAF) vulnerability, when the pointed memory is freed and accessed in different manners. Even though a fuzzer has explored the code of the pointer dereference, it may miss the potential HT-Vuls if the vulnerable heap operation sequence (i.e., a free operation causing dangling pointers happens before the dereference) is not met. Thus, a heap-based temporal vulnerability is closely related to the accessed pointer and the operation sequence before it. Awareness of the heap operation (e.g., memory allocation, deallocation and access) sequence is a key to finding HT-Vuls.

Researchers have proposed several solutions to take into account heap operation sequences during fuzzing. They in general rely on prior knowledge or static analysis to *recognize candidate vulnerable operations*, and then *guide the fuzzer to conduct a directed-style fuzzing* to explore such operations. For instance, UAFL [40] utilizes static analysis to find candidate HT-Vuls and related heap operations, and takes transitions between such heap operations as new feedback to guide the fuzzer towards the sequences of target operations and trigger candidate HT-Vuls. LTL-Fuzzer [26] relies on expert knowledge to instrument applications at locations related to potential temporal violations, guides the fuzzer to explore such instrumented locations, and verifies violations at runtime. UAFuzz [28] conducts directed-fuzzing to explore UAF sites pre-defined by users. However, these solutions rely on either expert knowledge or imprecise static analysis, and thus have limitations in practice.

In this paper, we propose a novel heap operation sequence sensitive fuzzing solution HTFuzz, which does not rely on pre-defined or pre-analyzed candidate operation sequences. Instead, we propose to *increase the diversity of heap operation sequences* rather than focusing on exploring a limited set of candidate operation sequences. Specifically, we try to track heap operation sequences, use it as a new feedback in addition to code coverage, and guide the fuzzer to increase the diversity. With a greater diversity of heap operation sequences, it is more likely to find potential HT-Vuls. To accomplish such a solution, there are three research questions to address.

RQ1: Overhead of getting heap operation sequence feedback. The number of heap operations in an execution trace of a testcase in general is very large. It would cost a fuzzer too much time to record all operations and their orders at runtime, and too much memory to store them. Furthermore, it is difficult to compare two testcases' heap operation sequences. To balance the overhead and granularity of sequence feedback, in HTFuzz, we use a ring buffer to track the latest heap allocation and deallocation operations, and at each memory access operation we use the ring buffer's hash as the index to update *a sequence coverage bitmap*. In other words, we split a full heap allocation/deallocation operation sequence into several (maybe overlapping) fixed-length segments at memory access points, and record all segments in a sequence coverage bitmap, similar to saving edges into the code coverage bitmap in AFL.

RQ2: Sensitivity to pointer aliases. Note that, a heap-based temporal vulnerability requires not only a special heap operation sequence, but also the pointers accessed by these operations. Pointers may have aliases and therefore different life cycles, causing troubles for developers to manage pointers. Therefore, a heap-based temporal vulnerability is more likely to happen if there are more aliases of pointers. Thus, it is important to track pointers and their aliases during fuzzing, though a precise tracking is expensive. For efficiency, we propose to *count the number of pointers (and aliases) accessed by heap operations* during fuzzing, and prioritize testcases that access more pointers when the fuzzer is about to select seeds to mutate.

RQ3: Coordination with code coverage feedback. As demonstrated by existing solutions, code coverage feedback is useful for fuzzers to find vulnerabilities. Thus, it is reasonable to keep it and combine it with our new heap operation sequence feedback. However, these two feedback mechanisms may conflict in certain scenarios. For instance, AFL identifies an *effector map* of interesting bytes that are likely to contribute to code coverage, and skips mutating other bytes during fuzzing. However, the skipped bytes may contribute to the diversity of heap operation sequences. In our solution HTFuzz, we upgrade the *effector map* identification algorithm to consider heap operation sequence as well, and utilize MOPT [22] to dynamically schedule the mutation operators to balance both the code coverage and the heap operation sequence coverage.

In summary, in our solution HTFuzz, we (1) introduce a novel heap operation sequence feedback to the fuzzer and tune its seed saving strategy, which will increase the diversity in control-flow of heap operation sequences; (2) count the number of pointers (and aliases) accessed by the heap operations to tune the seed selection strategy, which will prioritize seeds with more diversity in data-flow of heap operation sequences; and (3) tune seed mutation strategies to make the new feedback work with existing code coverage feedback. Based on this design, we have implemented a prototype of HTFuzz, and evaluated its effectiveness with a thorough evaluation.

Specifically, we collected 14 real-world applications from recent fuzzing papers and datasets [28, 40, 43] to evaluate the fuzzing performance. We compared HTFuzz with 11 fuzzers, consisting of 2 fuzzers (i.e., UAFL [40] and UAFuzz [28]) specifically designed for UAF vulnerabilities (i.e., a representative type of HT-Vuls) and 9 open-source greybox fuzzers (i.e., AFL [52], Angora [6], MOPT [22], AFL-sensitive [41], PathAFL [48], Memlock [44], Ankou [25], TortoiseFuzz [43] and one variant). We also did the Mann-Whitney U-test for statistic evaluation. The results showed that, HTFuzz outperformed all 11 baselines, finding 1.47x more heap operation sequences and 1.45x more 0day HT-Vuls than the best of all baselines. Moreover, during the whole fuzzing campaign of 6144 CPU days, HTFuzz discovered 37 zero-day vulnerabilities with 37 CVEs assigned (including 32 new HT-Vuls).

In summary, this work makes the following contributions.

- We proposed a novel heap operation sequence sensitive fuzzing solution HTFuzz to efficiently discover HT-Vuls. It tunes the seed saving strategy of a fuzzer with a lightweight heap operation sequence feedback tracking mechanism, tunes the seed selection strategy by counting the number of pointers accessed, and tunes seed mutation strategies to coordinate with traditional code coverage feedback.
- We have implemented a prototype of HTFuzz and evaluated its effectiveness. It outperformed all 11 state-of-the-art fuzzers in terms of both the number of heap operation sequences and HT-Vuls discovered. In addition, we have discovered 37 new vulnerabilities, including 32 new HT-Vuls.
- We open source HTFuzz and data in the online repository [1] for further study.

2 MOTIVATION

In this section, we demonstrate HT-Vuls with a motivation example, then present limitations of existing fuzzers in discovering HT-Vuls. At last, we discuss our observations and approach.

2.1 Motivation Example

Figure 1 shows a motivation example of a heap-based temporal vulnerability. It is extracted from a real-world program MP4Box which contains the UAF vulnerability CVE-2019-20164. Its dynamic call graph (i.e., CG) is shown in Figure 2, where the nodes represent the executed functions and the edges represent the call relationship.

There are three steps involved in the occurrence of this UAF vulnerability. First, the execution begins at the program entry *main* and arrives at the function *gf_isom_ box_new_ex* (i.e., an *alloc* operation) to generate a *heap memory object*. Second, in the subsequent execution, the function *stbl_AddBox* is invoked, and it calls the function *gf_isom_box_del* (i.e., a *dealloc* operation) to release the *heap memory object*. At last, the crash happens at another point (i.e., a *use* operation) in the function *gf_isom_box_del* when accessing the unsafely released *heap memory object*. In conclusion, the UAF vulnerability can be triggered by sequentially executing the following code lines, i.e., [56-45-49-24-28-21-22-36-41-24-26 (malloc)-42-31-34-13-14 (use)-16 (free)-50-13-14 (crashed use)]. It is noticed

[1]https://github.com/sharedata21/HTFuzz

```
1  typedef struct{
2      const struct box_registry_entry *registry;
3      ...
4  } GF_Box;
5  static struct box_registry_entry {
6      GF_Box * (*new_fn)();
7      void (*del_fn)(GF_Box *a);
8      GF_Err (*read_fn)(GF_Box *s, GF_BitStream *bs);
9      ...
10 } box_registry;
11
12 //isomedia/box_funcs.c
13 void gf_isom_box_del(GF_Box *a) {
14     other_boxes = a->other_boxes; { use()} //crash
15     if (!a->registry && use_dump_mode)
16         a->registry->del_fn(a); // del_fn =free,...
17 }
18 GF_Box *gf_isom_box_new_ex(...) {
19     a = box_registry[idx].new_fn(); //new_fn =stco_New,...
20 }
21 GF_Err gf_isom_box_read(GF_Box *a, GF_BitStream *bs) {
22     return a->registry->read_fn(a,bs); //read_fn =stbl_Read,minf_Read...
23 }
24 GF_Err gf_isom_box_parse_ex(...){
25     if (...)
26         newBox = gf_isom_box_new_ex(...); ->stco_New() ->malloc()
27     if (...)
28         e = gf_isom_box_read(newBox, bs);
29 }
30 //isomedia/box_code_base.c
31 GF_Err stbl_AddBox(GF_Box *s, GF_Box *a) {
32     switch (a->type) {
33         case GF_ISOM_BOX_TYPE_STCO:
34             gf_isom_box_del(s->ChunkOffset); ->free();
35 }}
36 GF_Err stbl_Read(GF_Box *s, GF_BitStream *bs) {
37      e = gf_isom_box_array_read(s, bs, stbl_AddBox);
38      CALL(gf_isom_box_array_read_ex(..., GF_Err (*add_box)))//do callback
39      {
40         while (parent->size>=8){
41             e = gf_isom_box_parse_ex(...); ->gf_isom_box_new_ex()
42             e = add_box(parent, a); //if add_box = stbl_AddBox
43 }}}
44 //isomedia/box_funcs.c
45 GF_Err gf_isom_box_array_read_ex(...,GF_Err(*add_box)(GF_Box*,GF_Box*)) {
46     GF_Err e;
47     GF_Box *a = NULL;
48     while (parent->size>=8){
49         e = gf_isom_box_parse_ex(&a, ...); ->gf_isom_box_read()
50         if (e && a) gf_isom_box_del(a);
51         e = add_box(parent, a);
52         ...
53 }}
54 int main(){
55     ...
56     -> gf_isom_box_array_read_ex(...);
57 }
```

Figure 1: Motivation example (UAF, CVE-2019-20164).

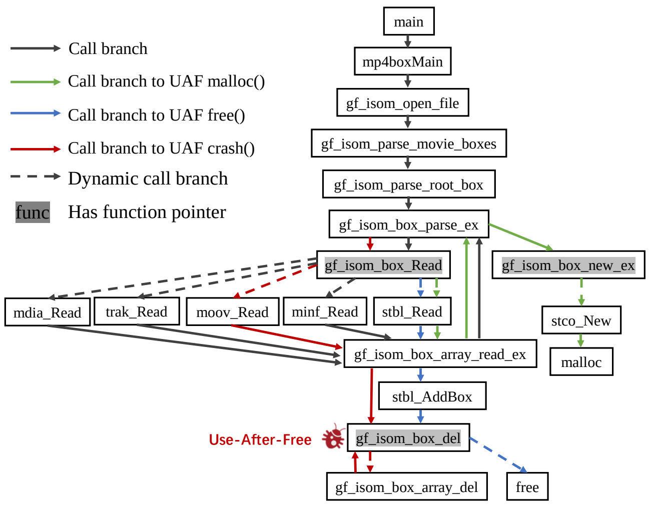

Figure 2: Call graph of the motivation example.

that, the first use at *line* 14 is not crashed. Specifically, the function pointer *a->registry->del_fn(a)* at *line* 16 points to *free*() and unsafely releases the *GF_Box struct a*, so the normal accessed pointer of *struct a* (e.g., *a->other_boxes* at *line* 14) becomes an abnormal accessed pointer. With the *struct a* turning into a dangling pointer, the pointer accessed again at *line* 14 results in a use-after-free issue and may crash.

2.2 Current Greybox Fuzzing and Limitation

Current greybox fuzzers take code coverage feedback to guide fuzzing that is verified effective in AFL [52]. However, code coverage is good for code exploration but not a good choice for HT-Vuls as it ignores the time-line information. Assuming that *seed*1 [malloc,use1,use2,free] and *seed*2 [malloc,use1,free,use2] are generated while fuzzing, their covered codes are the same. Thus, *seed*2 with dangling pointers would be discarded although it is more valuable.

As a supplementary to simple code coverage feedback, TortoiseFuzz [43], AFL-sensitive [41], and Ankou [25] use different methods to increase the feedback sensitivity (e.g., memory operations and combining branch differences), but all suffer from losing awareness of heap operation sequences. PathAFL [48] attempts to record the path coverage but has to give up the order information (i.e., it cannot differentiate *seed*1 and *seed*2, too.) because of the unaffordable recording overhead (e.g., the gf_isom_box_Read() can call many functions, but only the *stbl_Read*() is vulnerable to the UAF in Figure 2). Therefore, they are not suitable for HT-Vuls discovery.

Several fuzzing solutions, such as UAFL [40] and LTL-Fuzzer [26] , utilize heap operation sequences for HT-Vuls discovery. Specifically, they first utilize expert knowledge or perform static analysis to capture a candidate set of vulnerable operation sequences (e.g., the UAF sequence *malloc->free->use*). Then, they use the coverage of candidate sequences as targets to guide fuzzing. The candidate sequence is an essential premise for the fuzzing process. However, extracting candidate sequences is limited by prior knowledge or static analysis because sequences are easily broken and missed, due to the open challenges of static analysis (e.g., alias analysis [14]).

For example, in the motivation example, if the function pointer of *a->registry->del_fn(a)* at *line* 16 in *gf_isom_box_del* points to *free(a)*, the subsequent usages of *struct a* are candidate vulnerable pointer accesses and the related sequences should be added into the candidate set. However, since it is hard to decide which function the *a->registry->del_fn(a)* will point to before running, some vulnerable heap operation sequences (e.g., lines 16->50->13->14) are left out. In addition, the two function pointers (i.e., in *gf_isom_box_read* and *gf_isom_box_new_ex*) and a callback function (i.e., *gf_isom_box _array_read_ex*) as shown in Figure 2 are hard to be pre-analyzed or pre-defined, too. Even worse, if the heap operation sequence is controlled by loop variables, extracting sequences in advance is more complicated. For example, the *while loop* variable *parent->size* at *line* 48 affects the order and length of heap operation sequences (e.g., *alloc*, *dealloc*, *use* operations at *line* 49, 50 and etc.).

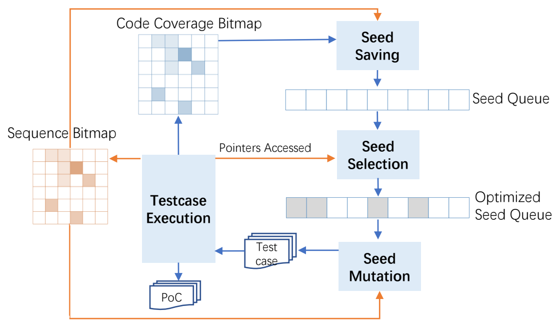

Figure 3: Framework of HTFuzz.

2.3 Observation and Solution

As aforementioned, the code coverage feedback used by existing fuzzers is insufficient for finding HT-Vuls, and fuzzers that utilize candidate heap operation sequences as guidance are limited by expert knowledge or imprecise static analysis. In this paper, we consider tracking heap operation sequences as a new feedback metric and increase the diversity of heap operation sequences.

Specifically, finding HT-Vuls requires two conditions: the vulnerable heap operation sequence (i.e., control-flow) and the intricate pointer management (i.e., data-flow). However, tracking heap operation sequences and the pointers accessed [2] with fine-grained analysis methods brings an extremely high overhead and seriously slows down the fuzzing throughput. For instance, we have tried dynamic taint propagation to associate the heap operations with heap addresses, which yields an inefficient fuzzer. Lastly, we choose to track coarse-grained heap operation (e.g., *malloc()* and *free()*) sequences, which is shown to be effective in our prototype.

Although we only track the coarse-grained heap operation sequences, the code/sequence explosion problem is still an open question [48]. To balance the efficiency and overhead, we propose a lightweight sequence feedback tracking mechanism, which splits heap operation sequences into segments by memory access operations, similar to breaking a program path into edges.

Besides heap operation sequence feedback, the data-flow constraint of heap operation sequences, i.e., pointers being accessed, is another condition to meet. As it is expensive to precisely track the pointers (and aliases) as well, we only count the coarse-grained frequency of pointers accessed.

Furthermore, the optimization techniques based on code coverage may interfere with the capability of discovering heap operation sequences. For example, the "effector map" optimization for saving mutation power [21] may cause a potential loss in heap operation sequence exploration. Thus, we coordinate the new heap operation sequence feedback and the code coverage feedback during mutation.

3 METHODOLOGY

3.1 Overview of HTFuzz

HTFuzz is built upon the popular greybox fuzzing tool AFL [52] and follows the same workflow, but comes with improvements in three parts, i.e., seed saving, selection, and mutation. The architecture is shown in Figure 3. Specifically, we introduce the new heap operation sequence feedback by recording the latest heap operation sequences (with a new sequence bitmap) at memory access operation points, and save testcases that contribute either to code coverage or heap operation sequence coverage, to explore more heap operation sequences via fuzzing (subsection 3.2). We also modify the seed selection module to prioritize seeds with more pointers accessed to increase the diversity of pointer aliases and the probability of potential bugs (subsection 3.3). Furthermore, we introduce heap operation sequence information in seed mutation and leverage MOPT [22] to schedule mutation operators, to balance both code coverage and sequence coverage (subsection 3.4).

[2]Read or write memory through pointer operations

3.2 Heap Operation Sequence Feedback

Heap operation sequence feedback is the basis of saving, selecting and mutating seeds in fuzzing. We design an efficient tracking method to deal with the sequence record explosion problem and be convenient for further calculation such as comparing differences.

On the one hand, we denote heap operations with binary numbers to save the recording space, e.g., assigning 1 for heap allocation and 0 for deallocation. Thus, the heap operation sequence can be presented as a string of numbers. On the other hand, we further improve the recording efficiency by separating the heap operation sequence into segments. Specifically, if we project each heap operation sequence before a memory access operation onto the execution time-line, the sequences are composed of segments with heap allocation, deallocation, and access operations. As the heap operation sequence at the subsequent memory access moment derives from the state of the previous memory access, the sequence record may overlap. Thus, we only record the latest heap operation sequence of length L before the memory access operation.

We track the latest heap operations with a ring buffer [3] of length L. Assuming a heap operation sequence represented as [1,1,access,0,0,access,1,access,access,0], if L is set to 3, the above sequence is separated as [0,1,1] [4], [1,0,0], [0,0,1] and [0,0,1]. The ring buffer is further encoded to a decimal number (e.g., [1,0,0] is encoded to 4), and then saved into a new bitmap using the memory access operation location as the index. Thus, the whole bitmap approximately records the heap operation sequence feedback during fuzzing.

Note that, the larger the ring buffer is, the more sensitive the feedback is to heap operation combinations. However, being too sensitive will reduce the fuzzing efficiency, because of the overhead of tracking and computation. Thus, we tried different configurations for the ring buffer's length and the sequence bitmap's size to select proper values, thereby mitigating explosion problems and increasing the efficiency for comparing differences. Detailed experiments are described in subsubsection 5.3.1.

3.3 Pointer Alias Tracking

The perception of heap operation sequence may make a fuzzer save too many interesting seeds, and we should prioritize valuable ones to accelerate the fuzzing process for HT-Vuls in practice.

[3]An end-to-end connected buffer that is usually used for buffering data streams.
[4]We initialize the record with all 0s, so that the first segment could be complemented to length L.

The original seed selection strategy of AFL is designed for fuzzing throughput, which prioritizes seeds with fewer execution time and smaller file sizes. Such prioritized seeds are marked as favored and will have a higher probability of being chosen to apply further mutation and generate offspring testcases.

As aforementioned, the diversity of pointer aliases is another critical factor in finding HT-Vuls. We therefore propose to prioritize seeds with more pointers getting accessed in the seed selection phase, in order to find HT-Vuls in a more efficient way. However, tracking precise pointers with heap operations brings unaffordable overhead, as demonstrated in subsection 2.3. We therefore only statically identify the instructions related to pointer (and aliases) access and count their frequency during each seed's execution, and use it to roughly represent the pointer access count of the seed. Seeds with a larger access count are prioritized as favored and get selected with a higher probability by the fuzzer.

3.4 Heap Operation Sequence Coordination

To find more HT-Vuls, we should provide well-organized seed mutation to offer abundant candidates. Code coverage feedback is helpful for greybox fuzzers in practice [24], and we combine it with our new heap operation sequence feedback. However, simply combing them together would cause some conflicts. We therefore introduce some modifications to AFL accordingly.

In the seed mutation stage, AFL provides many mutation operators, and one of the commonly used mutation operators is "bitflip 8/8". AFL provides an extra optimization on the "bitflip 8/8" mutation operation to further improve efficiency, i.e., an "effector map" is generated to mark useful input bytes and avoid subsequent mutation operators on useless bytes. Lv et al. point out the effects of mutation operators vary greatly and propose a further optimization named MOPT [22] to schedule the operators towards code coverage exploration dynamically. Specifically, MOPT utilizes the amount distribution of mutation operators that contributes to *interesting test cases* (i.e., new code coverage) to adjust seed mutation.

However, both the above optimizations are designed for code coverage and sometimes harm the exploration of heap operation sequences. For example, MOPT schedules mutations towards the new code coverage, thus delaying the exploration for sequences and even skipping the new sequence because of the "effector map".

To cooperate with our perception of heap operation sequence, we modify the identification algorithm of "effector map", and mark bytes that produce more *interesting test cases* of new heap operation sequences as useful too. Furthermore, we reference MOPT to schedule mutation operators based on *new interesting test cases* comes from the heap operation sequence feedback besides the original code coverage feedback.

4 IMPLEMENTATION

We implement a prototype of HTFuzz based on AFL, and modify the instrumentation and fuzzing loop of AFL according to our designs.

4.1 Instrumentation

We instrument the target application with code snippets, based on the LLVM framework [16], to collect feedback information during testing to guide fuzzing. Besides the branch coverage tracking

Algorithm 1 Tracking Feedback with Instrumentation

```
 1: INITIALIZE(BranchCov, HeapOpSeq, PtrCount, Ring)
 2: function CovFb(BranchCov, cur_loc, prev_loc)      ▷ Coverage feedback as AFL
 3:     BranchCov[cur_loc ⊕ prev_loc] ++
 4: function PtrCountUpdate(ins, PtrCount)           ▷ Count of pointers accessed
 5:     PtrCount++
 6: function Shifting(Ins, Ring)
 7:     if IsAlloc(Ins) then Ring.push(1)
 8:     else if IsDealloc(Ins) then Ring.push(0)
 9: function SeqFb(HeapOpSeq, cur_loc, prev_loc, Ring)     ▷ Sequence feedback
10:     hash = Encoding(Ring)
11:     HeapOpSeq[cur_loc ⊕ prev_loc ⊕ hash]++
12: function Instrumentation
13:     for each BB in each Func do
14:         cur_loc = <CompileTime_Random>         ▷ Get basic block IDs as AFL
15:         Insert(CovFb(BranchCov, cur_loc, prev_loc))
16:         mflag = 0                          ▷ Ready for logging the ring buffer
17:         for each Ins in BB do
18:             if IsGetElementPtr(Ins) then
19:                 Insert(PtrCountUpdate(Ins, PtrCount))
20:             else if IsHeapAllocDeallocFunc(Ins) then
21:                 Insert(Shifting(Ins, Ring))
22:             else if IsMemAccess(Ins) and mflag == 0 then
23:                 Insert(SeqFb(HeapOpSeq, cur_loc, prev_loc, Ring))
24:                 mflag = 1                  ▷ Logging at most once in one BB.
25:         prev_loc = cur_loc >> 1
```

Algorithm 2 Fuzzing loop of HTFuzz

```
 1: INITIALIZE(Queue, Prog)
 2: repeat                                                    ▷ Fuzzing Loop
 3:     input ⟵ NextSeed(Queue)                   ▷ Select seed based on orders
 4:     NumChildren ⟵ MutateEnergy(input)
 5:     for i = 0 ⟶ NumChildren do
 6:         eff_map = IdentifyInterestingBytes(input)
 7:         child ⟵ MutateWithMOPT(input, eff_map)             ▷ Seed mutation
 8:         status, Time, BranchCov, HeapOpSeq, PtrCount ⟵ Execute(Prog, child)
 9:         if status == CRASHED then
10:             Report child as the bug PoC
11:         else if Is_Interesting(BranchCov, HeapOpSeq) then
12:             Queue ⟵ Queue ∪ child                            ▷ Saving seed
13:             Update_Bitmap_Score(child, Time, PtrCount) ▷ Adjust seed orders
14: until time out or manual stop
```

instrumented by AFL, our instrumentation tracks heap operation sequences and pointer information, as shown in Algorithm 1.

Specifically, we instrument at each basic block entry (line 15) and collect the branch coverage feedback *BranchCov* in a bitmap (line 2-3) like AFL. Further, we instrument each instruction related to pointer retrieval (line 18-19), i.e., the instructions whose opcode are `GetElementPtr` [20] or whose operands are `GetElementPtr` expressions, to count the number of pointers accessed at runtime (line 4-5).

Further, we instrument each invocation to allocation and deallocation functions (line 20-21), e.g., *malloc()*, *realloc()*, *free()*, to record the latest heap operations in the ring buffer *Ring* of length *L* (line 6-8). The value of L is set as 3 in our experiment, and more discussion could be found in the evaluation section.

Lastly, we instrument the first memory access instruction in the basic block (line 22-24), to log the ring buffer (line 9-11) into an additional map *HeapOpSeq*, which is used for recording the heap operation sequence feedback. To determine whether an instruction is a memory access, we use the API *mayReadFromMemory()* and *mayWriteToMemory()* provided by LLVM.

4.2 Fuzzing Loop

We slightly refine the fuzzing process of AFL-2.52b [52], to guide the fuzzing engine with the feedbacks collected by aforementioned instrumentation, i.e., branch coverage, heap operation sequences, and the number of pointers accessed. As shown in Algorithm 2, we upgrade the seed mutation, seed saving, and seed selection strategies of AFL.

When mutating a seed testcase, we follow AFL to build an *effector map* of this seed (line 6) to recognize *interesting* bytes. Mutations on such *interesting* bytes are likely to yield new testcases that could bring new feedback, e.g., code coverage. The fuzzer will skip mutating bytes that are not interesting to save computing resources. We update AFL's algorithm and consider bytes that would yield new testcases bringing heap operation sequence feedback as *interesting* too. Then, we utilize MOPT [22] to schedule mutation operators to mutate the seed and generate new testcases (line 7).

After testing the newly generated testcase, we report a bug PoC (Proof Of Concept) if the testing crashed or triggered alarms set by sanitizers (line 9-10), or save the testcase to the seed queue (line 11-12) if it is interesting. A testcase is interesting if and only if it brings new feedback, i.e., new code coverage or new heap operation sequence feedback, as shown in Formula 1.

$$\text{Is\_Interesting} = \begin{cases} \mathit{True}, & \text{hasNew(BranchCov)} \\ & \lor \text{hasNew(HeapOpSeq)} \\ \mathit{False}, & \text{otherwise} \end{cases} \tag{1}$$

Further, we also refine the seed selection strategy of AFL. As shown in line 13, we update the API *Update_Bitmap_score()* of AFL, which adjust the orders of seeds in the queue and marked some seeds as favored. The favored seeds will be picked for mutation in the following iterations (line 3). Originally, AFL prefers seeds that have shorter execution time and smaller file size. We update this strategy and consider seeds that have more pointers being accessed during testing as favored too. Formula 2 shows the detailed definition of whether a testcase is favored.

$$\text{Is\_Favored} = \begin{cases} \mathit{True}, & \text{IsLargest(PtrCount)} \\ & \lor \text{IsMinimum(Time * FileSize)} \\ \mathit{False}, & \text{otherwise} \end{cases} \tag{2}$$

5 EVALUATION

In this paper, we develop a new heap operation sequence sensitive fuzzer HTFuzz for HT-Vuls. To evaluate HTFuzz's effectiveness, we conduct several experiments on real-world applications and answer the following research questions:

RQ1. Is HTFuzz effective at finding real-world HT-Vuls?

RQ2. How well do the improvements made by HTFuzz contribute to the efficiency of fuzzing towards HT-Vuls?

RQ3. How does HTFuzz compare to other fuzzers?

5.1 Experiment Setup

Following the suggestions of Klees et al. [15], we conduct the experiments carefully as below.

5.1.1 Fuzzers to compare against. We list the related works in Table 1 with the description of their feedback metrics and fuzzing strategies. Totally, we compare HTFuzz against 11 fuzzers, including 2 fuzzers (i.e., UAFL [40] and UAFuzz [28]) specifically designed for UAF vulnerabilities, and 9 state-of-the-art fuzzers (i.e., AFL [52], Angora [6], MOPT [22], PathAFL [48], Memlock [44], Ankou [25], TortoiseFuzz [43], and two implementations of AFL-sensitive [41]).

These fuzzers are selected based on the following considerations. We try to compare UAFL [40] and UAFuzz [28] as they target HT-Vuls with candidate heap operation sequences. Since they are not open-source, we choose to compare with them on their published benchmark applications to test if HTFuzz can find the same vulnerabilities. For other state-of-the-art greybox fuzzers, we build them with their open-source codes and run them on our dataset with their default parameters. Among them, AFL and Angora are popular baseline fuzzers studied in fuzzing papers. MOPT is included as we leverage its mutation strategy. The other six fuzzers are compared because their supplement of code coverage feedback involves the information of our heap operation sequence feedback. AFL-sensitive (including afl-sen-wa and afl-sen-ma), TortoiseFuzz and Memlock pay more attention to memory operations, such as memory read, memory write and memory usage. PathAFL is aware of the sequence information but gives up order information because of the record explosion. Ankou uses the combinatorial branch differences as a feedback, which implicitly considers the sequence information. All these fuzzers are considered as partially related to HTFuzz.

Table 1: Related works of greybox fuzzing. AFL is presented as the baseline tool and it saves seeds with new covered branches (i.e., edge coverage), selects seeds based on file size and execution time, and mutates seeds with multiple mutation operations (multi-mu-ops). Other fuzzers are based on AFL and modify parts of it. The "HSeq" column presents if the fuzzer involve with the heap operation sequence information, i.e., ○means no, ◐means partially and ●means yes.

Work	Seq	Seed Saving	Seed Selection	Seed Mutatation
AFL	○	new branches	file_size*time	multi-mu-ops
Angora	○	context-aware branch	gradient decent	AFL's
MOPT	○	AFL's	AFL's	PSO algorithm
AFL-sen-wa	◐	memory-write-aware branch	AFL's	AFL's
AFL-sen-ma	◐	memory-access-aware branch	AFL's	AFL's
TortoiseFuzz	◐	coverage accounting	coverage accounting	AFL's
Memlock	◐	memory usage peak value	AFL's	AFL's
PathAFL	◐	path coverage	neighbour coverage	AFL's
Ankou	◐	combinatorial branch difference	combinatorial distance	AFL's
UAFL	●	typestate sequence	sequence coverage	seq-aware mutation
LTL-fuzzer	●	typestate sequence	sequence similarity	AFL's
UAFuzz	●	cut-edge coverage	sequence similarity	AFL's
HTFuzz	●	heap operation sequence	pointer accessed	seq-aware PSO algorithm

Table 2: Real-world programs used in experiment. We also present the version and fuzzing command for each program.

Program	Version	Command
jasper	1.900.2	jasper -f @@ -t mif -F /dev/null -T jpg
jpegoptim	d23abf2	jpegoptim @@
lrzip	9de7ccb	lrzip -t @@
yasm	6caf151	yasm @@
mjs	9eae0e6	mjs -f @@
binutils	2.28	readelf -w @@
boolector	3.0.1	boolector @@
GraphicsMagick	1.3.26	gm identify @@
ImageMagick	7.0.8	convert @@ /dev/null
gpac	4c19ae5	MP4Box -diso @@ -out /dev/null
gpac_latest	7804849	MP4Box -hint @@
nasm	2.14	nasm -f bin @@ -o ./tmp
mxml	2.12	mxmldoc @@
yara	3.5.0	yara @@ strings

5.1.2 Benchmark applications. We collect the benchmark applications from the published datasets of UAFL and UAFuzz, and from the repository of TortoiseFuzz with several factors. The collected application should be frequently tested and actively developed, contain HT-Vuls (especially with recent-reported vulnerabilities) and can be verified with the given PoC files. Thus, we exclude five applications (i.e., giflib, cxxfilt, gifsicle, bzip2 and openh264) as we could not manually reproduce the reported bugs with given PoCs to ensure the vulnerabilities existed, exclude three applications (i.e., boringssl, libpff, liblouis) because of compilation fail, and add two applications with new versions (i.e., gpac, nasm) as they contain HT-Vuls. In addition, we exclude the results of two applications (i.e., cflow and recutils) as all the fuzzing tools behave the same. Finally, we build a dataset of 14 widely-used real-world applications as shown in Table 2 with version information, including well-known developer tools (i.e., yasm, mjs, binutils, yara, boolector and nasm), graphics processing libraries (i.e., jasper, jpegoptim, GraphicsMagick, and ImageMagick), document processing tools (i.e., mxml and lrzip) and multimedia file and video processing tools (i.e., gpac).

5.1.3 Performance Metrics. We use HT-Vuls numbers, heap operation sequence amount, and code coverage as the metrics to evaluate fuzzers. Specifically, we calculate the total and the average number of vulnerabilities [5] instead of unique crash numbers. We collect heap operation sequence amount by counting the sub-sequences in the sequence bitmap with afl-showmap [52]. For coverage information, we use gcov [13] to get line coverage. We also do the Mann-Whitney U-test for statistic evaluation when comparing the vulnerability findings of different fuzzers.

5.1.4 Configuration Parameters. Since the fuzzers heavily rely on random mutation, there could be performance jitter during the fuzzing process. We take two actions to mitigate this. First, we test each program for a long time (i.e., 72 hours) until the fuzzer reaches a relatively stable state. Second, we perform each experiment for 8 times to mitigate the random noises during fuzzing. In total, our experiments constitute 6144 CPU days: it takes 3120 (10 x 6 x 8 times x3 days + 14 x 5 x 8 times x3 days) CPU days for RQ2, and takes 3024 (14 x 9 x 8 times x3 days) CPU days for RQ3. The applications are compiled with clang 6.0.0, and we run them in different fuzzers with the command options shown in Table 2. All our experiments are performed on four machines of the same configuration, i.e., with Intel(R) Xeon(R) CPU E5-2630 v3 Processor (2.40GHZ, 32 cores) and 32GB of RAM under 64-bit Ubuntu LTS 16.04.

5.2 RQ1. Finding Real-world Vulnerabilities

In our experiments, HTFuzz found 92 vulnerabilities totally in the benchmark applications, including 74 HT-Vuls. Among all the findings, 37 vulnerabilities are new, including 32 new HT-Vuls. We reported the findings to the maintainers and got 37 new CVEs assigned, and we are glad to see some of the early reported vulnerabilities are already fixed (e.g., GPAC). The details of each vulnerability, including its bug ID, the bug type, the bug status, the vulnerable application with version information and the vulnerable function, are shown in Table 3 sorted by bug type. Figure 4 shows the number of HT-Vuls found by HTFuzz accumulated over time. The numbers are counted every 10 hours. We can see that nearly half (i.e., 34/74) of the vulnerabilities are discovered within 10 hours.

Table 4 presents the total number of vulnerabilities found by HTFuzz and compared baseline fuzzers, including 0day and 1day vulnerabilities of HT-Vuls and other types. In addition, it also shows the number of HT-Vuls missed by HTFuzz but found by individual fuzzers. For instance, AFL found 64-48-4 vulnerabilities, which means AFL found 64 vulnerabilities in total, and 48 among them are HT-Vuls, while 4 HT-Vuls among them are missed by HTFuzz. We can see that HTFuzz outperforms all baselines in terms of real-world vulnerabilities discovery, including the total number and HT-Vuls.

Table 3: The zero-day vulnerabilities found by HTFuzz. UAF, NPD and BO are short for Use After Free, Null Pointer Dereference and Buffer Overflow vulnerability types, respectively. Vulnerabilities with gray background are all HT-Vuls.

Bug ID	Version	Type	Status	Vulnerable Function
CVE-2021-33453	LRZIP 0.641	UAF	accepted	ucompthread()
CVE-2019-20169	GPAC 0.8.0	UAF	accepted & fixed	trak_Read()
CVE-2019-20164	GPAC 0.8.0	UAF	accepted & fixed	gf_isom_box_del()
CVE-2019-20168	GPAC 0.8.0	UAF	accepted & fixed	gf_isom_box_dump_ex()
CVE-2020-35980	GPAC 0.8.0	UAF	accepted & fixed	gf_isom_box_del()
CVE-2021-33461	YASM 1.3.0	UAF	accepted	yasm_intnum_destroy()
CVE-2021-33462	YASM 1.3.0	UAF	accepted	expr_traverse_nodes_post()
CVE-2021-33467	YASM 1.3.0	UAF	accepted	pp_getline()
CVE-2021-33468	YASM 1.3.0	UAF	accepted	error()
CVE-2021-33439	MJS version 6	NPD	accepted	gc_compact_strings()
CVE-2021-33440	MJS version 6	NPD	accepted	mjs_bcode_commit()
CVE-2021-33441	MJS version 6	NPD	accepted	exec_expr()
CVE-2021-33442	MJS version 6	NPD	accepted	json_printf()
CVE-2021-33444	MJS version 6	NPD	accepted	getprop_builtin_foreign()
CVE-2021-33445	MJS version 6	NPD	accepted	mjs_string_char_code_at()
CVE-2021-33446	MJS version 6	NPD	accepted	mjs_next()
CVE-2021-33449	MJS version 6	NPD	accepted	mjs_bcode_part_get_by_offset()
CVE-2021-33437	MJS version 6	NPD	accepted	frozen_cb()
CVE-2021-33455	YASM 1.3.0	NPD	accepted	do_directive()
CVE-2021-33456	YASM 1.3.0	NPD	accepted	hash()
CVE-2021-33457	YASM 1.3.0	NPD	accepted	expand_mmac_params()
CVE-2021-33458	YASM 1.3.0	NPD	accepted	find_cc()
CVE-2021-33460	YASM 1.3.0	NPD	accepted	if_condition()
CVE-2021-33463	YASM 1.3.0	NPD	accepted	yasm_expr__copy_except()
CVE-2021-33465	YASM 1.3.0	NPD	accepted	expand_mmacro()
CVE-2021-33466	YASM 1.3.0	NPD	accepted	expand_smacro()
CVE-2019-20163	GPAC 0.8.0	NPD	accepted & fixed	gf_odf_avc_cfg_write_bs()
CVE-2020-35981	GPAC 0.8.0	NPD	accepted & fixed	SetupWriters()
CVE-2020-35982	GPAC 0.8.0	NPD	accepted & fixed	gf_hinter_track_finalize()
CVE-2021-33450	NASM 2.14rc0	NPD	accepted	nasm_calloc()
CVE-2021-33452	NASM 2.14rc0	NPD	accepted	nasm_malloc()
CVE-2021-33451	LRZIP 0.641	NPD	accepted	fill_buffer()
CVE-2021-33438	MJS version 6	BO	accepted	json_parse_array()
CVE-2021-33448	MJS version 6	BO	accepted	unknown-module>
CVE-2021-33443	MJS version 6	BO	accepted	mjs_execute()
CVE-2020-35979	GPAC 0.8.0	BO	accepted & fixed	gp_rtp_builder_do_avc()
CVE-2021-33464	YASM 1.3.0	BO	accepted	inc_fopen()

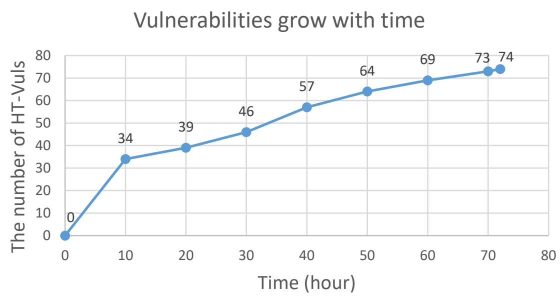

Figure 4: Unique HT-Vuls accumulated over time.

[5] During fuzzing, we follow AFL [52] to save unique crashes that trigger program segment faults. Then, we use ASAN [33] to deduplicate the unique crashes with crash stack hash. Finally, we perform manual verification to associate crashes with the corresponding CVE-IDs or bug reports. The total number is meaningful as it is a direct metric for the fuzzer's capability.

Table 4: The vulnerabilities found by the fuzzers. For X-Y-Z, X is the number of vulnerabilities of all types, Y is the number of HT-Vuls vulnerabilities, and Z is the number of HT-Vuls vulnerabilities missed by HTFuzz. The `Total` column shows the union of all fuzzers. The `0day-non-CVE` row shows 0day vulnerabilities that have not been assigned with CVE as they are fixed in the latest version or not our target bug types.

	HTFuzz	AFL	Memlock	AFL-sen-ma	AFL-sen-mw	PathAFL	Tofuzz	MOPT	Angora	Ankou	Total
0day	37-**32**-0	24-22-0	9-7-0	9-9-0	9-9-0	21-18-0	20-18-0	21-19-0	8-8-0	26-22-0	37-32-0
0day-non-CVE	-	3-0-0	3-0-0	-	-	2-0-0	1-0-0	-	3-2-2	6-1-1	11-3-3
1day	55-**42**-0	37-26-4	29-20-2	16-12-0	11-9-0	28-23-4	30-20-2	46-32-1	27-25-10	49-36-2	87-66-24
Sum	92-**74**-0	64-48-4	41-27-2	25-21-0	20-18-0	51-41-4	51-38-2	67-51-1	42-37-14	81-59-3	135-101-27

Table 5: The recording map density and the number of vulnerabilities for the different map size and latest heap operation sequence length. The map density reveals the ratio that the recording map is utilized. 16-8 means the map size is 2^{16} and the latest heap operation sequence length is 8. `Den` means the map density (with %). GMagick is short for GraphicsMagick; IMagick is short for ImageMagick. The Avg/Sum means the average of `Den` and the total summation of `Bug`.

Program	AFL		16-16		16-8		16-4		16-3		16-1	
	Den	Bug	Den	Bug	Den	Bug	Den	Bug	Den	Bug	Den	Bug
jpegoptim	0.34	1	0.49	1	0.49	1	0.40	1	0.39	1	0.35	1
lrzip	2.07	2	19.3	2	11.1	3	4.30	2	3.42	8	2.41	2
yasm	13.2	13	**100**	2	**99.9**	7	**55.1**	7	40.1	13	18.0	12
mjs	6.19	13	**100**	8	**95.2**	6	34.5	10	20.6	13	8.23	13
binutils	7.59	3	**100**	3	**75.2**	3	15.1	3	14.1	2	7.53	2
boolector	7.87	1	**65.4**	1	19.4	1	11.9	1	9.97	1	8.02	1
gpac	8.64	4	**100**	3	**86.3**	3	36.3	4	21.7	3	11.0	3
gpac_latest	13.2	3	**100**	3	**85.7**	2	41.2	4	30.5	4	16.5	3
GMagick	9.58	2	8.25	2	9.54	2	9.23	2	9.80	2	9.55	2
IMagick	14.4	1	8.36	0	14.9	0	9.24	0	14.6	1	9.82	0
Avg/Sum	8.30	43	60.2	25	49.8	28	21.7	34	16.5	**48**	9.14	39

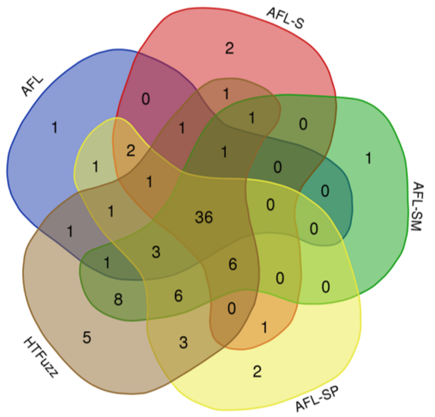

Figure 5: The HT-Vuls findings of three options (AFL-S, AFL-SM, AFL-SP) of HTFuzz.

The `0day` row in Table 4 shows the number of new CVE vulnerabilities found by each fuzzer. In total, these fuzzers found 37 CVE vulnerabilities, as shown in Table 3, whereas HTFuzz found all of them. For the target vulnerability types, i.e., HT-Vuls, HTFuzz find 32 new HT-Vuls, which is 1.45x more than the best of all baseline fuzzers (i.e., AFL and Ankou which find 22 new HT-Vuls).

5.3 RQ2. Ablation Studies of HTFuzz

As we have to set the ring buffer length and the sequence bitmap size, we first perform a study of different settings by comparing the fuzzing performance. Then, as we modify three components of AFL, we perform an ablation study to detail the contributions of different modifications in HTFuzz.

5.3.1 Hyperparameter settings of HTFuzz. As described in the design section, the ring buffer length L and the sequence bitmap size 2^K are configurable, and we find the proper values of L and K by fuzzing with different settings. Although our goal of testing is more heap operation sequences which means a high bitmap density, the distinguishing ability of the bitmap will decrease quickly because of the hash collision problem when the map density crosses a warning line [12], and the fuzzer could even become dumb when the bitmap is filled fully. So, we consider two metrics of fuzzing performance here, i.e., the bitmap density and the vulnerability number.

We have two options to mitigate the quick increase of bitmap: enlarging the bitmap size [6] or decreasing the sequence length L. The baseline of AFL is K=16 and L=0. We first test the enlargement of bitmap size and set L as fixed 16. We can see from Table 5, five programs (i.e., yasm, mjs, binutils) have reached nearly 100% bitmap density when K=16 and L=16, and larger bitmap size values cannot solve the filled full problem (the results are omitted for space). Moreover, the execution overhead will increase quickly with a larger bitmap size than 16, which is also evaluated in CollAFL [12]. Following another direction, we set the bitmap size fixed as 2^{16} (i.e., K=16) and decrease the sequence length L. Considering the potential hash collision, we set the warning line of the bitmap density as 50% (AFL reminds the warning line of 70%), and get the best results when L=3 and K=16, which is used as default settings for HTFuzz.

5.3.2 Ablation study. As the complete solution of HTFuzz contains three modifications, we want to figure out if they all contribute to the final results. We set three options to control each of them and run an experiment to compare the results when opening one respectively, i.e., AFL-S, AFL-SP and AFL-SM.

The results are shown in Table 6 including the HT-Vuls findings, the code coverage and the heap operation sequence amount. We also calculate the intersection set of different options in Figure 5. The data in Table 6 and Figure 5 shows that all of the three modifications contribute to the final results. Specifically, the sequence amount is increased from AFL (i.e., 28608) to AFL-S (i.e., 39045), and to AFL-SM (i.e., 52901). And the HT-Vuls amount increases from AFL (i.e., 48) to the complete HTFuzz (i.e., 74) with seed saving (i.e., -S), seed prioritization (i.e., -P) and seed mutation (i.e., -M) added gradually. Though the winner of code coverage is the AFL-SM(26.41%), AFL-SM's ability to find HT-Vuls is lower than HTFuzz, revealing that code coverage is not the sole criterion for HT-Vuls discovery. Thus, the whole solution of HTFuzz should contain all three modifications.

[6]Designing new hash collision function will not help to reduce the bitmap density for the quickly increasing information.

Table 6: The number of HT-Vuls, the code coverage (L-Cov) and the heap operation sequences number (HSeq) of three strategies (AFL-S, AFL-SP, AFL-SM). AFL-S stands for the fuzzer based on the base AFL with seed saving strategy of heap operation sequence feedback. AFL-SP stands for the fuzzer based on AFL-S added with the seed selection strategy of pointers accessed. AFL-SM stands for the fuzzer utilizing the coordinated mutation strategy of MOPT based on the AFL-S. In the columns, U means the total amount of HT-Vuls in the total runs. In the Sum/Avg/Avg line, we present the summation of U in the test benchmark as well as the average percentage of L-Cov and HSeq.

Program	HTFuzz			AFL			AFL-S			AFL-SP			AFL-SM		
	U	L-Cov	HSeq	U	L-Cov	Hseq	U	L-Cov	Hseq	U	L-Cov	Hseq	U	L-Cov	Hseq
binutils	6	23.40%	53,590	3	19.27%	13,996	2	20.43%	20,360	4	20.77%	18,171	6	23.50%	50,174
boolector	1	12.40%	11,902	1	12.40%	7,394	1	12.44%	9,334	1	12.43%	11,179	1	12.40%	10,469
gpac	5	3.26%	73,081	4	2.70%	25,512	3	2.63%	32,612	4	2.82%	33,275	3	4.75%	76,368
gpac_latest	3	3.43%	85,686	3	3.00%	37,393	3	2.90%	58,307	3	2.88%	44,543	3	3.45%	72,177
GMagick	2	10.45%	17,331	2	9.55%	12,940	2	9.75%	12,609	2	9.17%	10,233	2	10.70%	19,573
IMagick	1	8.30%	27,639	1	9.80%	22,209	1	8.95%	23,906	1	7.81%	12,103	1	8.20%	30,608
jasper	4	17.06%	10,244	4	19.43%	6,983	3	21.33%	7,529	4	20.47%	6,285	4	21.90%	12,158
jpegoptim	1	46.71%	292	1	47.85%	368	1	49.51%	301	1	49.19%	364	1	49.40%	643
lrzip	10	20.68%	6,206	2	20.40%	3,745	8	20.40%	4,408	8	20.40%	5,377	9	20.75%	6,640
mjs	9	62.31%	75,214	9	64.42%	34,911	9	62.58%	61,757	10	62.22%	61,031	6	62.85%	74,038
yasm	12	37.84%	152,866	12	37.92%	104,471	12	38.04%	130,918	10	37.24%	120,388	11	39.45%	160,202
nasm	10	43.95%	132,747	3	42.60%	94,523	4	43.74%	124,250	5	43.69%	122,148	11	44.30%	148,838
mxml	2	27.54%	22,529	1	27.48%	11,182	1	27.44%	19,416	2	27.50%	19,361	2	27.55%	22,928
yara	8	40.44%	60,766	2	40.00%	24,896	1	39.65%	40,921	6	40.89%	54,315	2	40.55%	55,795
Sum/Avg/Avg	74	25.56%	52,149	48	25.49%	28,608	51	25.70%	39,045	61	25.53%	37,055	62	26.41%	52,901

Table 7: The reproducibility results of HTFuzz for the HT-Vuls in the two datasets[28, 40].

Bug ID	Dataset	Program	Type	Reproduce
yasm-issue-91	UAFuzz	yasm(6caf151)	UAF	Yes
CVE-2021-33461	UAFuzz	yasm(6caf151)	UAF	New
CVE-2021-33462	UAFuzz	yasm(6caf151)	UAF	New
CVE-2021-33467	UAFuzz	yasm(6caf151)	UAF	New
CVE-2021-33468	UAFuzz	yasm(6caf151)	UAF	New
CVE-2018-11416	UAFuzz,UAFL	jpegoptim(d23abf2)	DF	Yes
mjs-issue-78	UAFuzz,UAFL	mjs(9eae0e6)	UAF	Yes
mjs-issue-73	UAFuzz,UAFL	mjs(e4ea33a)	UAF	Yes
CVE-2018-11496	UAFuzz,UAFL	lzrip(ed51e14)	UAF	Yes
CVE-2018-10685	UAFuzz	lzrip(9de7ccb)	UAF	Yes
CVE-2021-33453	UAFuzz,UAFL	lzrip(9de7ccb)	UAF	New
CVE-2019-6455	UAFuzz	rec2csv(97d20cc)	DF	Yes
CVE-2017-10686	UAFuzz	nasm(7a81ead)	UAF	Yes
CVE-2017-6966	UAFL	readelf(2.28)	UAF	Yes
CVE-2018-20592	UAFL	Mini XML(2.12)	UAF	Yes
cflow-issue	UAFL	GNU cflow(1.6)	UAF	Yes
boolector-issue	UAFL	Boolector(3.0.1)	UAF	Yes
ImageMagick-issue	UAFL	ImageMagick(7.0.8)	UAF	Yes
CVE-2018-19216	UAFL	nasm(2.14)	UAF	Yes
CVE-2018-20535	UAFL	nasm(2.14)	UAF	Yes
CVE-2017-17813	UAFL	nasm(2.14)	UAF	New
CVE-2017-17817	UAFL	nasm(2.14)	UAF	New

5.4 RQ3. Compare with Other Fuzzers

5.4.1 Compare with UAFL and UAFuzz. We compare with two fuzzers (i.e., UAFL and UAFuzz) designed for UAF vulnerabilities as they are typical HT-Vuls. Specifically, we consider the vulnerabilities in their benchmark applications found by UAFL or UAFuzz as the ground truth, and run HTFuzz on the applications with the same version and the same commands to see if we can find these vulnerabilities. The results in Table 7 show that HTFuzz can find all the reported UAF and DF (Double-Free) vulnerabilities in the datasets. In addition, we find 7 previously unknown UAF vulnerabilities which could be considered as missed vulnerabilities of the previous fuzzers. 5 of them are assigned with new CVEs, and the other 2 have been reported by others.

We take a further root cause analysis for the newly found vulnerabilities to see why they are missed. CVE-2017-17813 and CVE-2017-17817 in the application NASM are caused by the function pointer $preproc\text{->}getline$ in $assemble\_file()$ which points to $free()$ and generates a dangling pointer. And CVE-2021-33462 is related to the callback function which finally calls to $free()$. These three vulnerabilities are missed because of the difficulty of precise static analysis. CVE-2021-33453 in the application LRZIP is missed because it is a multi-thread UAF vulnerability, which is also hard for static analysis used by UAFuzz and UAFL. As HTFuzz is sensitive to heap operations and thus could catch HT-Vuls with a higher probability. When a specific thread interleaving is met by chance, it thereby catches the missed vulnerability. Further studies on active thread interleaving are required to better discover this type of vulnerabilities and we leave it as future work.

5.4.2 Compare with other greybox fuzzers. We compare HTFuzz to the open-source greybox fuzzers by re-running them on our dataset. The results of found HT-Vuls are shown in Table 8 and the heap operation sequence amount results are shown in Table 9. From the view of vulnerability number, HTFuzz is more capable of discovering HT-Vuls than other greybox fuzzers. Although HTFuzz is only a little better than the second winner MOPT (i.e., 48.88 vs. 41) and the third winner Ankou (i.e., 48.88 vs. 40.88) when calculating the average number, HTFuzz is much better than them according to the total results (i.e., 74 vs. 51 and 74 vs. 59). Based on the p-value of the Mann-Whitney U-test with the basis of HTFuzz, HTFuzz outperforms all the nine compared fuzzers [7]. It has to mention that two afl-sensitive tools utilize the bitmap to record execution changes as HTFuzz but don't balance the information sensitivity, leading to bitmap full quickly and bad results (i.e., 74 vs. 21 for afl-sensitive-ma and 74 vs. 18 for afl-sensitive-mw).

From Table 9, we can see that HTFuzz triggers much more heap operation sequences than other fuzzers. Specifically, HTFuzz finds

[7]It means statistically significant when p-value is smaller than 0.05.

Table 8: The number of HT-Vuls found by different fuzzers. In the columns, `U` and `Avg` mean the total and the average amount of HT-Vuls in the total runs. The symbol - means the corresponding fuzzer could not compile the binary.

Program	HTFuzz		AFL		Memlock		AFL-sen-ma		AFL-sen-mw		PathAFL		Tofuzz		MOPT		Angora		Ankou	
	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg
binutils	6	4.75	3	2.25	2	2	2	2.00	2	2.00	2	2.00	3	2.13	4	3.00	5	3.25	5	4.00
boolector	1	1.00	1	1.00	1	1	0	0.00	1	1.00	1	1.00	1	1.00	1	1.00	-	-	1	1.00
gpac	5	3.00	4	2.88	0	0	0	0.00	0	0.00	3	3.00	3	2.88	3	2.25	7	3.25	2	1.63
gpac_latest	3	3.00	3	2.13	3	2.25	3	2.00	2	2.00	2	2.00	2	1.50	3	3.00	8	5.50	3	3.00
GMagick	2	1.13	2	1.13	0	0	0	0.00	0	0.00	1	1.00	0	0.00	2	2.00	0	0.00	1	0.25
IMagick	1	0.13	1	0.13	0	0	1	0.50	1	0.25	1	1.00	0	0.00	0	0.00	-	-	0	0.00
jasper	4	3.13	4	3.25	3	2	0	0.00	0	0.00	2	2.00	3	2.63	3	3.00	6	3.00	3	2.63
jpegoptim	1	1.00	1	1.00	1	0.75	0	0.00	0	0.00	1	1.00	1	1.00	1	1.00	0	0.00	1	0.63
lrzip	10	6.75	2	2.00	5	3.75	9	5.00	3	2.25	7	5.33	4	2.25	5	4.00	8	3.50	10	6.75
mjs	9	4.75	9	4.50	2	1.5	0	0.00	0	0.00	10	6.67	6	3.63	6	4.50	0	0.00	7	4.88
yasm	12	8.38	12	7.38	3	2.25	4	2.50	7	3.25	7	5.83	8	5.38	9	8.75	2	0.50	11	7.75
nasm	10	5.38	3	2.13	4	1.75	1	0.75	1	1.00	2	2.00	6	2.88	10	5.00	-	-	11	5.63
mxml	2	1.38	1	1.00	1	0.25	1	0.25	1	0.25	1	1.00	1	1.00	1	1.00	0	0.00	2	1.25
yara	8	5.13	2	1.13	2	1.5	0	0.00	0	0.00	1	1.00	0	0.00	3	2.50	1	0.88	2	1.50
Sum/Sum	**74**	**48.88**	48	31.88	27	19	21	13.00	18	12.00	41	34.83	38	26.25	51	41.00	37	19.88	59	40.88
P-value			2.39E-06		9.03E-07		1.52E-08		3.15E-09		5.76E-06		4.04E-06		5.09E-04		7.05E-08		3.77E-02	

Table 9: The amount of heap operation sequences found by different fuzzers.

Program	HTFuzz	AFL	AFL-sen-ma	AFL-sen-mw	Memlock	PathAFL	Tofuzz	MOPT	Angora	Ankou
binutils	53,590	13,996	10,522	10,588	15,342	11,132	10,897	24,066	21,802	16,001
boolector	11,902	7,394	2,145	2,991	6,272	6,804	6,225	7,587	-	4,036
gpac	73,081	25,512	26,105	27,770	37,570	22,219	25,576	37,453	46,878	40,458
gpac_latest	85,686	37,393	38,792	36,228	53,380	38,098	28,114	51,886	49,637	54,895
GMagick	17,331	12,940	8,833	7,129	3,711	9,502	9,989	18,494	7,569	9,621
IMagick	27,639	22,209	14,643	13,606	3,546	9,600	11,658	26,488	-	16,419
jasper	10,244	6,983	1,080	1,080	5,543	5,800	6,074	10,067	6,344	4,299
jpegoptim	292	368	266	359	297	286	338	533	101	173
lrzip	6,206	3,745	4,706	3,763	6,150	3,748	3,969	5,012	3,610	3,701
mjs	75,214	34,911	16,745	17,298	26,377	25,279	25,745	50,164	13,255	23,569
yasm	152,866	104,471	65,897	69,505	72,582	64,510	101,298	108,959	31,668	87,041
nasm	132,747	94,523	69,817	65,999	96,108	93,056	91,432	107,678	-	78,863
mxml	22,529	11,182	7,563	7,649	9,854	12,526	11,965	15,747	3,438	8,383
yara	60,766	24,896	11,596	10,569	24,606	27,885	20,366	33,059	8,160	21,294
Avg	**52,149**	28,608	19,908	19,609	25,810	23,603	25,260	35,514	17,497	26,339

more sequences of 1.82x, 2.62x, 2.66x, 2.02x, 2.21x, 2.06x, 1.47x, 2.98x and 1.98x compared to AFL, AFL-sensitive-ma, AFL-sensitive-mw, Memlock, PathAFL, TortoiseFuzz, MOPT, Angora and Ankou, respectively. Although the sequence amount is more reasonable for us as we are targeting sequences, we also test the basic code coverage of the fuzzers. Generally, HTFuzz achieves a similar number as baseline fuzzers and details are presented as additional artifacts.

5.5 Case Study

5.5.1 Realloc-caused vulnerabilities. We found several HT-Vuls caused by the incorrect use of the realloc function. Realloc is used to resize the memory region a pointer points to and it has two parameters, i.e., ptr and size. However, when the function fails to extend the original memory block, it allocates a new larger block elsewhere. As the developer is expected to update the previous pointer as invalid, it may have the risk of UAF vulnerabilities when forgetting to update. There is another corner case when the size is 0. In the case of CVE-2016-8693 (a DF vulnerability) in the application Jasper, although the developers remember updating the original memory buffer $m\text{->}buf$ to the return value of the $jas\_realloc2()$, the code execution exits before the updating action in the un-patched version, causing a double-free at another release point of $m\text{->}buf$. HTFuzz focuses on heap operation sequences including operation orders and access operations, thus is able to find the vulnerabilities.

5.5.2 Incomplete patch. It has to mention that we found a new UAF vulnerability (CVE-2020-35980) resulting from an incomplete fix of the previous one (CVE-2019-20164). Both vulnerabilities are related to the function pointer of $free()$ and hard to be statically analyzed. In this case, the crash happens at the $gf\_isom\_box\_del()$ function when reusing the heap structure vul_a. The developer previously patched the bug by adding an if condition to invoke error handling codes and avoiding the abnormal use of vul_a. However, we found a new UAF bug in the patched function again, which reveals that finding HT-Vuls is essential because they are complicated to fix.

6 THREAT TO VALIDITY

One threat to validity is that, because of the well-known large amount of heap operation sequence information, HTFuzz has to set parameters of recording map size K and the latest heap operation sequence length L configurable, and coarse-grained instrument the sensitive heap operations. Although we did experiments based on AFL [52] to study the values, we cannot ensure the default setting (i.e., K=16 and L=3) is the best, and we also suffer from hash collision problem of bitmap. A better choice is to adjust the size K and length

L based on the available computing resources adaptively, and we leave it as future work.

Another threat to validity is that it is hard to evaluate the capabilities of fuzzers, including the testing randomness and benchmark bias. To address this issue, we collected the benchmarks from newly published datasets [28, 40, 43] with functional diversity and compared tools by their published results or re-running. In addition, each experiment was extended to 72 hours and conducted 8 times to mitigate the fuzzing randomness factor. We did the Mann-Whitney U-test for the statistic evaluation of HTFuzz's result of 8 experiments compared with other tools. According to the different compared metrics, we believe that the results of HTFuzz are representative.

7 RELATED WORK

In this paper, we propose HTFuzz, a fuzzing solution for HT-Vuls. So we introduce the fuzzing solutions for HT-Vuls and other greybox fuzzing improvements.

7.1 Greybox Fuzzing for HT-Vuls

Heap-based temporal vulnerability is one of the most common types of memory-corruption vulnerabilities [37, 39], and discovering them is still an open question and has been a research hotspot [1, 28, 40]. Researchers propose several solutions such as static analysis [35, 47] and runtime detection [4, 17, 38, 49]. Fuzzing, especially greybox fuzzing, has made a big success in finding bugs [24], such as OSS-fuzzing platform [32], LibFuzzer [31], and the AFL family [3, 5, 6, 8, 9, 11, 12, 18, 19, 22, 25, 30, 43, 44, 48, 52, 54–56]. Thus recently, lots of fuzzing solutions have been proposed to find various types of bugs [2, 3, 6, 44, 46], including finding HT-Vuls [28, 40].

Fuzzing solutions such as UAFL [40], UAFuzz [28] and LTL-Fuzzer [26] pre-define or pre-analyze the candidate vulnerable operations statically and guide the fuzzer to cover the operations in sequence to find HT-Vuls. However, their discovery ability is limited by expert knowledge and imprecise static analysis. This mainly motivates us to take another technique route, i.e., introducing heap operation sequence feedback to fuzzing without prior knowledge.

The original code coverage-based greybox fuzzing is insufficient for HT-Vuls as it is unaware of the heap operation sequence which is critical. There are some general fuzzing solutions to obtain more subtle code execution by enriching the coverage feedback sensitivity. For instance, TortoiseFuzz [43], MemFuzz [10], and AFL-sensitive [41] enhance the code coverage feedback with memory-related operations. However, they face the same limitation for finding HT-Vuls that they cannot catch sequence information. Although there are some attempts such as PathAFL [48] which tries to record the whole execution trace, PathAFL finally gives up the order information because of the record explosion problem. It is also the first challenge to design HTFuzz and we propose a lightweight tracking mechanism for heap operation sequence feedback. Also, previous solutions don't adapt the seed selection based on accessed pointers which is another critical condition to find HT-Vuls, and the seed mutation to handle some conflicts of code coverage and heap operation sequence coverage.

7.2 Greybox Fuzzing Improvements

In HTFuzz, we modify three main components of greybox fuzzing based on a new heap operation sequence feedback towards HT-Vuls. And there are more improvements for the three components and other parts in greybox fuzzing solutions.

As we mentioned in the evaluation, several works are proposed to improve the feedback sensitivity, from basic block level [29] to edge level [52], and to the path level [48]. And seed saving based on new feedbacks is generally proposed for specific targets, such as alias instruction pair coverage of different threads for concurrency vulnerabilities in KRACE [46], memory usage peak value for memory consuming bugs in Memlock [44]. Seed selection is used to prioritize the valuable seeds to mutate with a higher probability, and usually, it is modified with seed saving. For example, CollAFL [12] and PathAFL [48] propose to prioritize seeds with neighbour branches. Moreover, some works take advanced algorithms for power schedule, such as Multi-Armed Bandit [42, 50, 53], Ant Colony Optimization [57], etc. Seed mutation optimizations are also useful, e.g., MOPT [22] and EMS [23].

Apart from modifying the above parts, there are some other techniques that can improve fuzzing, such as increasing the fuzzing speed, e.g., REDQUEEN [2], Full speed fuzzing [27], etc. Driller [36] and Qsym [51] combine symbolic execution technique to deal with magic numbers. And machine learning is utilized for fuzzing [7, 34, 45]. As they are proposed to enhance fuzzing from different angles, they can be combined with other solutions including HTFuzz.

8 CONCLUSION

In this paper, we propose a heap operation sequence sensitive fuzzer named HTFuzz for heap-based temporal vulnerabilities. As current greybox fuzzers have limitations to integrating heap operation sequence feedback, such as relying on expert knowledge or imprecise static analysis, we propose to increase the diversity of heap operation sequences and tune the fuzzer based on heap operation sequence feedback. We evaluated HTFuzz with 11 fuzzers on 14 real-world applications. The results showed that, HTFuzz outperformed all 11 baselines in terms of heap operation sequences and 0day heap-based temporal vulnerabilities. HTFuzz found 37 new vulnerabilities with 37 CVEs assigned, including 32 new heap-based temporal vulnerabilities.

ACKNOWLEDGMENTS

We would like to thank the shepherd and all anonymous reviewers for their insightful comments and feedback. This project has received funding, in part, from the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (U1936211, 61972224, 62102406), the Strategic Priority Research Program of the Chinese Academy of Sciences (XDC02020300), Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006 and Outstanding Science and Technology Talent Program of ISCAS. All opinions expressed in this paper are solely those of the authors.

REFERENCES

[1] [n. d.]. 2021 cwe top25. http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the Network and Distributed System Security Symposium.*

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[4] J. Caballero, G. Grieco, M. Marron, and A. Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. *ACM* (2012).

[5] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *ACM Conference on Computer and Communications Security.*

[6] Peng Chen and Hao Chen. 2018. Angora: efficient fuzzing by principled search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy.* IEEE Computer Society.

[7] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. *arXiv preprint arXiv:2002.08568* (2020).

[8] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 1580–1596.

[9] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. 515–530. https://doi.org/10.1145/3319535.3354249

[10] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. 2019. Memfuzz: Using memory accesses to guide fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST).* IEEE, 48–58.

[11] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association.

[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy.* IEEE.

[13] gcov. [n. d.]. a test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov.

[14] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE).* 323–333. https://doi.org/10.1109/ISSRE.2019.00040

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

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[17] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS.*

[18] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM.

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

[20] LLVM. [n.d.]. The Often Misunderstood GEP Instruction. https://llvm.org/docs/GetElementPtr.html.

[21] American Fuzzy Lop. [n.d.]. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt.

[22] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Security Symposium.* USENIX Association.

[23] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. [n.d.]. EMS: History-Driven Mutation for Coverage-based Fuzzing. ([n. d.]).

[24] V. Manes, H. Han, C. Han, s. cha, M. Egele, E. J. Schwartz, and M. Woo. 5555. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 01 (oct 5555), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[25] V. J. M. Manès, S. Kim, and S. K. Cha. 2020. Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* 1024–1036.

[26] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2021. Linear-time Temporal Logic guided Greybox Fuzzing.

[27] Stefan Nagy and Matthew Hicks. 2018. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. *CoRR* abs/1812.11875 (2018). arXiv:1812.11875

[28] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. (2020).

[29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Network and Distributed System Security Symposium.* The Internet Society.

[30] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17).* USENIX Association.

[31] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev).* IEEE, 157–157.

[32] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. (2017).

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Usenix Conference on Technical Conference.*

[34] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2018. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. (2018).

[35] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. *ACM SIGPLAN Notices* 53, 4 (2018), 693–706.

[36] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23rd Network and Distributed Systems Security Symposium.* The Internet Society.

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 48–62.

[38] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. *Proceedings of the Twelfth European Conference on Computer Systems* (2017).

[39] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses.* Springer-Verlag.

[40] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *42nd International Conference on Software Engineering.* ACM.

[41] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019).* USENIX Association.

[42] Jinghan Wang, Chengyu Song, and Heng Yin. [n.d.]. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. ([n. d.]).

[43] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. (2020).

[44] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *42nd International Conference on Software Engineering.* ACM.

[45] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. 2022. Evaluating and Improving Neural Program-Smoothing-based Fuzzing. (2022).

[46] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 1643–1660.

[47] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. 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).*

[48] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. 2020. PathAFL: Path-Coverage Assisted Fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security.* 598–609.

[49] Yves Younan. 2015. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Network and Distributed System Security Symposium.*

[50] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. {EcoFuzz}: Adaptive {Energy-Saving} Greybox Fuzzing as a Variant of the Adversarial {Multi-Armed} Bandit. In *29th USENIX Security Symposium (USENIX Security 20).* 2307–2324.

[51] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium.* USENIX Association.

[52] Michal Zalewski. 2013. American fuzzy lop (AFL) fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.t.

[53] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kang Lu. 2022. MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing. (2022).
[54] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. 2018. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* (2018).
[55] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *NDSS*.
[56] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1099–1114.
[57] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2169–2182.