CtxFuzz: Discovering Heap-Based Memory Vulnerabilities Through Context Heap Operation Sequence Guided Fuzzing

Jiacheng Jiang[1], Cheng Wen[2(✉)], and Shengchao Qin[2(✉)]

[1] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
[2] Guangzhou Institute of Technology, Xidian University, Xi'an, China
wencheng@xidian.edu.cn, shengchao.qin@gmail.com

Abstract. Heap-based memory vulnerabilities are significant contributors to software security and reliability. The presence of these vulnerabilities is influenced by factors such as code coverage, the frequency of heap operations, and the specific execution order. Current fuzzing solutions aim to efficiently detect these vulnerabilities by utilizing static analysis or incorporating feedback on the sequence of heap operations. However, these solutions have limited practical applicability and do not comprehensively address the temporal and spatial aspects of heap operations. In this paper, we propose a dedicated fuzzing technique called CtxFuzz to efficiently discover heap-based temporal and spatial memory vulnerabilities without requiring any domain knowledge. CtxFuzz utilizes context heap operation sequences (the sequences of heap operations such as allocation, deallocation, read, and write that are associated with corresponding heap memory addresses) as a new feedback mechanism to guide the fuzzing process. By doing so, CtxFuzz can explore more heap states and trigger more heap-based memory vulnerabilities, both temporal and spatial. We evaluate CtxFuzz on 9 real-world open-source programs and compare their performance with 5 state-of-the-art fuzzers. The results demonstrate that CtxFuzz outperforms most fuzzers in terms of discovering heap-based memory vulnerabilities. Moreover, Our experiments led to the identification of 10 zero-day vulnerabilities (10 CVEs).

1 Introduction

Heap-based temporal vulnerabilities (*e.g.*, null-pointer dereference, use-after-free and double-free) and spatial memory vulnerabilities (*e.g.,* buffer-overflow and allocation failure) pose serious threats to software security and reliability [6,20,31]. These vulnerabilities occur when a program performs incorrect or unsafe operations on heap memory, such as using freed memory, accessing invalid memory addresses, or consuming a large amount of memory. Such operations can result in memory corruption, which can cause the program to crash, behave unpredictably, or execute arbitrary code injected by an attacker.

W.-N. Chin and Z. Xu (Eds.): TASE 2024, LNCS 14777, pp. 205–223, 2024.
https://doi.org/10.1007/978-3-031-64626-3_12

Problem and Challenges. Detecting heap-based memory vulnerabilities is challenging, as they depend on various factors such as the frequency of heap operations, the specific execution order, and code coverage [14,16,24]. Moreover, these vulnerabilities can manifest themselves in different ways, such as temporal errors [4,11,22] and spatial errors [5,26,27]. Hence, it is essential to have a testing technique that can efficiently and comprehensively detect these vulnerabilities without necessitating prior knowledge of the program or the heap structure.

In recent years, fuzzing has emerged as a popular and effective testing technique for detecting and mitigating memory vulnerabilities. This technique involves supplying random or semi-random inputs to a program and observing its behavior for any irregularities. However, most fuzzing techniques do not adequately address heap-based memory vulnerabilities, particularly those that are temporal in nature. These vulnerabilities are not solely determined by code coverage, but also depend on the specific order of heap operations. Instead of being solely influenced by the amount of code exercised, these vulnerabilities arise from the sequence of heap operations. Consequently, fuzzing techniques that solely rely on code coverage as feedback may overlook a significant number of temporal errors. On the other hand, certain fuzzing techniques have been proposed to tackle heap-based memory vulnerabilities by utilizing various types of feedback, such as operation sequence coverage [28], typestate analysis [23], taint inference [30], memory usage [26], or user-defined properties [12,15]. However, these techniques often have limitations, such as focusing on a specific type of heap operation (*e.g.*, allocation or deallocation) [28], requiring expert knowledge [12], relying on imprecise static analysis [23], and thus limiting their practical applicability and effectiveness. Furthermore, these techniques frequently disregard the holistic nature of memory vulnerabilities, impeding a comprehensive approach to identifying and mitigating both temporal and spatial issues simultaneously.

Approach. We propose a novel fuzzing technique called CtxFuzz that aims to overcome the limitations of existing fuzzing solutions and efficiently discover both heap-based temporal and spatial memory vulnerabilities. CtxFuzz is a greybox fuzzing technique that does not require any domain knowledge. Instead, it utilizes Context Heap Operation Sequences (CHOS) as a feedback mechanism to guide the fuzzing process and input generation. CHOS refers to the sequences of heap operations (*e.g.*, allocation, deallocation, read, and write) associated with the corresponding heap memory addresses. CtxFuzz monitors the program's heap state and heap memory addresses accessed through instrumentation. Modifying the seed selection strategy by counting the number of generated CHOS can facilitate the generation of inputs that trigger more potential vulnerabilities. CtxFuzz is capable of detecting both temporal and spatial errors by manipulating the heap memory addresses and CHOS in various ways.

Evaluation. We evaluated the effectiveness and efficiency of CtxFuzz on 9 widely used real-world open-source programs that have undergone extensive testing by other fuzzers. In our comparison, we assessed CtxFuzz against 5 state-of-the-art fuzzers, namely AFL [29], AFL++ [7], HTFuzz [28], Memlock [26], and TortoiseFuzz [25]. The results demonstrate that CtxFuzz outperforms most

fuzzers in terms of discovering heap-based memory vulnerabilities. Specifically, CtxFuzz found 1.3x, 1.3x, 1.4x, 4.1x, and 1.7x more vulnerabilities than AFL, AFL++, HTFuzz, Memlock, and TortoiseFuzz, respectively. Furthermore, CtxFuzz identified 10 new zero-day vulnerabilities (10 CVEs) that had not been reported by any other studies. We promptly reported these vulnerabilities to the respective developers and vendors, receiving positive feedback and acknowledgments.

In summary, this paper makes the following contributions.

- ***Originality.*** We propose a dedicated fuzzing solution CtxFuzz to efficiently discover both heap-based temporal and spatial memory vulnerabilities without requiring any domain knowledge.
- ***Technique.*** Our approach leverages context heap operation sequences (CHOS) as a feedback mechanism to guide the fuzzing process and input generation. CtxFuzz monitors the program's heap state and heap memory addresses accessed through instrumentation. Modifying the seed selection strategy by counting the number of generated CHOS can facilitate the generation of inputs that trigger more potential vulnerabilities.
- ***Evaluation.*** We have implemented CtxFuzz and evaluated its effectiveness on 9 real-world open-source programs. We demonstrate that CtxFuzz outperforms 5 state-of-the-art fuzzers in terms of discovering heap-based memory vulnerabilities.
- ***Practical Impact.*** We discover 10 zero-day vulnerabilities (10 CVEs) that have not been previously reported, and report them to the corresponding developers and vendors.

2 Motivation

This section highlights the limitations of existing grey-box fuzzing techniques in detecting heap-based memory vulnerabilities. We also provide an informal description of how CtxFuzz works, exemplified by a zero-day vulnerability CVE-2023-49554 that was discovered by CtxFuzz.

Listing 1.1 shows a use-after-free (UAF) vulnerability in the real-world code processing tool *Yasm*. The vulnerability is triggered by a specific order of code execution in the function `do_directive`, which is repeatedly called by the function `nasm_parser_parse` (line 33) along with the function `pp_getline` (line 27). The function `do_directive` has a switch statement that can execute two cases: `PP_REP` (line 7) and `PP_ENDREP` (line 9). In the first case, it creates a heap memory object for a variable named `defining` (line 8) and stores its address in another variable named `istk→mstk` (line 10). In the second case, it uses the same address to access the memory object (line 10). However, before going back to `do_directive`, the function `pp_getline` may free the memory object (line 19). This leads to a UAF when `do_directive` tries to use the freed memory object again (line 3).

Existing mainstream grey-box fuzzers, such as AFL [29] mainly use code coverage as feedback to guide the fuzzing. In this case, the UAF can be triggered

```
1  // from modules/preprocs/nasm/nasm-pp.c
2  static int do_directive(Token *tline){
3      if (... && !istk->mstk->in_progress){ // Crash
4          return NO_DIRECTIVE_FOUND;
5      }
6      switch (i) {
7          case PP_REP:
8              defining = nasm_malloc(sizeof(MMacro)); // Memory allocation
9          case PP_ENDREP:
10             istk->mstk = defining; // Memory access
11     }
12 }
13
14 static char *pp_getline(void){
15     while (...){
16         ...
17         MMacro *m = istk->mstk; // Memory access
18         if (...) {
19             free_mmacro(m); // Memory deallocation
20         }
21     }
22     if (do_directive(tline) == DIRECTIVE_FOUND) {...}
23 }
24
25 // modules/preprocs/nasm/nasm-preproc.c
26 static char *nasm_preproc_get_line(yasm_preproc *preproc){
27     line = nasmpp.getline(); // a function pointer of pp_getline
28 }
29
30 // modules/parsers/nasm/nasm-parse.c
31 void nasm_parser_parse(yasm_parser_nasm *parser_nasm){
32     unsigned char *line;
33     while ((line=(unsigned char*)yasm_preproc_get_line(parser_nasm->preproc))!=NULL){
34         ...
35     }
36 }
37
38 void main(int argc, char *argv[]){
39     ...
40     nasm_parser_parse(...);
41 }
```

Listing 1.1. A UAF example simplified from CVE-2023-49554

if lines 8→10→17→19→3 are executed temporally. However, code coverage is not enough to capture the order and amount of heap operation, which is essential for triggering such heap-based memory vulnerabilities. Some fuzzing techniques also use heap operation sequences to detect heap-based memory vulnerabilities. For example, UAFL [23] uses static analysis to identify potential heap operation sequences and then directs fuzzing to cover them gradually. However, static analysis can miss some sequences, such as the aliasing between `defining` (line 8) and `istk→mstk` (line 10), or the pointees of the function pointer `getline` (line 27). HTFuzz [28] improves the diversity of heap operation sequences but does not take the corresponding memory objects into account. For example, both memory allocation (line 8) and deallocation (line 19) impact the variable `istk→mstk`. Additionally, HTFuzz overlooks memory read and write operations, exemplified by lines 10 and 17, that are critical in this context, thereby hindering efficient bug detection. Note that in our experiment, CtxFuzz was able to detect this bug in 10 h. HTFuzz and AFL took at least an additional 10 h compared to CtxFuzz. However, TortoiseFuzz [25] and MemLock [26] were unable to detect this vulnerability.

Our key insight is that tracking how heap operations manipulate different memory objects is crucial for delving into the intricate effects of memory management in the program and identifying vulnerabilities related to heap-based mem-

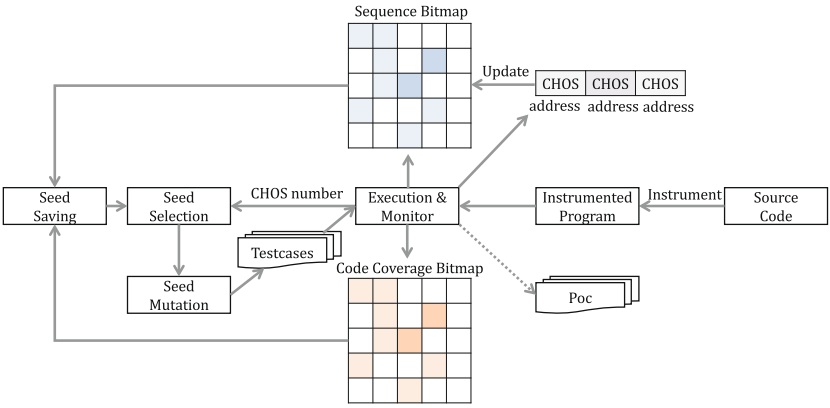

Fig. 1. The workflow of CtxFuzz

ory. Therefore, triggering various sequences of heap operations can be deemed as interesting behaviors similar to code coverage. CtxFuzz adheres to this concept by focusing on distinct heap memory addresses and the corresponding heap operations. Let us now explore the motivating example mentioned above. Assuming that we tracked each heap operation using a two-digit binary number representation: `00` for allocation, `01` for reading, `10` for writing, and `11` for deallocation. A set of heap operations can be represented as a sequence, and each sequence may be viewed as a distinct behavior of the programs. For instance, if the `do_directive` function allocates the heap memory address `0x1234` using `malloc` at line 8, we utilize the lower 16 bits of the address (`0x1234`) as the array index and record the operation `00` in the corresponding element. Subsequently, if a read operation to the address `0x1234` occurs, we append `01` to the respective element, resulting in `0001`. Finally, upon the function releasing the address `0x1234` with `free`, we add `11` to the allocated index, resulting in `000111`. This representation helps in identifying new behaviors in the sequence, similar to tracking code coverage. Our approach aids in maintaining a record of heap operation sequences for each memory address.

3 Methodology

3.1 Overview of CtxFuzz

The workflow of CtxFuzz is illustrated in Fig. 1. CtxFuzz follows the general workflow of grey-box fuzzers but introduces improvements in three areas, namely *feedback mechanism*, *instrumentation*, and *seed selection*. Specifically, CtxFuzz leverages the new context heap operation sequence (CHOS) feedback by recording a set of recently accessed heap memory addresses and their corresponding last heap operation sequences (*i.e.*, a new sequence bitmap in Fig. 1) at the entry of each basic block. To keep track of the CHOS information, CtxFuzz performs an

Algorithm 1: Tracking CHOS Feedback with Instrumentation

```
   Input: A program P
   Output: An instrumented program P'
1  Function SeqUpdate(SeqArray, RecentObj, Address, Code):
2      SeqArray[Address] = SeqArray[Address] ≪ 2 | Code // Update operation sequence
3      RecentObjUpdate(RecentObj, Address, K) // Update the last K accessed addresses
4  Function SeqFeedback(SeqCov, cur_loc, prev_loc, SeqArray, RecentObj):
5      Hash = 0
6      for each Address in RecentObj do
7          Hash = Hash ⊕ (SeqArray[Address]&(1 ≪ (L × 2)) − 1))
8      end
9      SeqCov[cur_loc ⊕ prev_loc ⊕ Hash] + + // non-zero integer
10 Initialize(BranchCov, SeqCov, SeqCount, SeqArray, RecentObj, LastOperation)
11 HeapInstMap = getHeapRelatedInstMap(P)
12 for each BB in each Func do
13     cur_loc = RandomInt() // Get a basic block ID as AFL++
14     Insert(BranchCov[cur_loc ⊕ prev_loc] + +) // Coverage feedback as AFL++
15     for each Inst in each BB do
16         if Inst in HeapInstMap and HeapInstMap[Inst] ≠ -1 then
17             LastOperationUpdate(LastOperation, Inst, HeapInstMap[Inst])
18     end
19     for each Inst, Code in LastOperation do
20         Insert(SeqUpdate(SeqArray, RecentObj, Inst, Code))
21         Insert(SeqCount + +)
22     end
23     Insert(SeqFeedback(SeqCov, cur_loc, prev_loc, SeqArray, RecentObj)) // CHOS feedback
24     prev_loc = cur_loc ≫ 1
25 end
```

LLVM-instrumentation pass to collect the heap-related operations and their corresponding addresses. Additionally, CtxFuzz preserves testcases that actively contribute to either code coverage or context heap operation sequence coverage. And the seed selection strategy is refined to prioritize seeds based on the quantity of context heap operation sequences generated after their execution, thus enhancing the likelihood of uncovering potential vulnerabilities.

3.2 Context Heap Operation Sequence Feedback

Context Heap Operation Sequence (CHOS) feedback is a novel feedback mechanism that we propose for fuzzing. CHOS tracks the sequences of heap operations (*e.g.*, allocation, deallocation, read, and write) associated with corresponding heap memory addresses, and uses them to guide the generation and selection of testcases. CHOS differs from other feedback mechanisms, such as code coverage, in that it captures the dynamic behavior and state of the heap memory, which is often the source of memory vulnerabilities. CHOS also addresses the challenge of sequence record explosion, which occurs when the number of heap operation sequences grows exponentially with the number of heap memory addresses and operations. Sequence record explosion can cause significant memory and time overhead for fuzzing, and reduce the diversity and quality of testcases.

Generally, a simply way to track CHOS is to utilize all the heap operation sequences for all the heap memory addresses. However, this would be inefficient and redundant. Instead, we only utilize the last K accessed heap memory addresses, and their corresponding sequences formed by the last L heap operations. This is based on the principle of program locality, which states that programs tend to access the same or nearby memory locations repeatedly. By

Algorithm 2: Heap-related Instruction Analysis

```
   Input: A program P
   Output: Heap-Related Instruction Map HeapInstMap
 1 Function getHeapRelatedInstMap(P):
 2     HeapInstMap ← {}
 3     for each Func in each Module do
 4         for each Arg in Func do
 5             if isPointerType(Arg) then
 6                 HeapInstMap[Arg] = −1
 7         end
 8         for each BB in Func do
 9             for each Inst in each BB do
10                 if isCallInst(Inst) then
11                     if isAllocFunc(Inst → getCalledFunction()) then
12                         HeapInstMap[Inst] = ALLOC
13                     else if isDeallocFunc(Inst → getCalledFunction()) then
14                         HeapInstMap[Inst] = DEALLOC
15                 else if isLoadInst(Inst) and Inst → getOperand(0) in HeapInstMap then
16                     HeapInstMap[Inst] = LOAD
17                 else if isStoreInst(Inst) then
18                     if Inst → getOperand(0) in HeapInstMap then
19                         HeapInstMap[Inst → getOperand(1)] = −1
20                         HeapInstMap[Inst] = STORE
21                     else if Inst → getOperand(1) in HeapInstMap then
22                         HeapInstMap[Inst] = STORE
23                 else if isGetElementPtrInst(Inst) and Inst → getOperand(0) in
                     HeapInstMap then
24                     HeapInstMap[Inst] = −1
25                 else if isBitCastInst(Inst) and Inst → getOperand(0) in HeapInstMap
                     then
26                     HeapInstMap[Inst] = −1
27             end
28         end
29     end
30     return HeapInstMap
```

doing so, we can reduce the memory and time overhead of CHOS feedback, and focus on the most relevant and interesting heap operation sequences for fuzzing. This is similar to how AFL, a popular fuzzing system, saves edges to the code coverage bitmap at the entry of each basic block.

To use the heap operation sequences for fuzzing, we need a way to measure the difference or novelty of the sequences generated by different testcases. To do this, we represent each heap operation as a two-digit binary (*e.g.*, 00 for allocation, 01 for reading, 10 for writing, and 11 for deallocation) and use a simple but effective operation $\oplus$ (*a.k.a.*, XOR) to concatenate sequences for different addresses. Specifically, we select K heap memory addresses that are accessed by the testcase, and XOR the elements of the sequence array corresponding to those addresses (line 7 of function `SeqFeedback` in Algorithm 1). The result number can be considered as a representation for the heap operation sequences of the testcase. We then use this number as the index for updating the new sequence bitmap at the beginning of each basic block (line 9 of function `SeqFeedback`). The new sequence bitmap records whether a testcase has generated a new or different heap operation sequence compared to the previous testcases. If a testcase has generated a new sequence, we save the testcase as a seed for further fuzzing. Otherwise, we discard the testcase.

3.3 Instrumentation

To keep track of the CHOS information, we perform an instrumentation to collect the heap-related operations and their corresponding addresses. Algorithm 1 presents the details of our instrumentation. Our approach follows the general workflow of AFL++. Given that AFL++ already includes an LLVM-instrumentation pass [18], we have made direct modifications to it to instrument the target application while preserving the original instrumentation implemented by AFL++.

Code coverage information is collected by instrumenting the entry of each basic block using a bitmap named *BranchCov* (line 14). Prior to the instrumentation for the CHOS feedback, we perform a heap-related instruction analysis to collect the heap operations (line 11). In order to optimize fuzzing efficiency, we strategically preserve the last operation for each pointer within each basic block (lines 15–18) and instrument the preserved operations correspondingly (line 20). Additionally, we also instrument the target program to track the number of generated CHOS (line 21), which influences seed selection during fuzzing. Finally, we instrument the code to update the generated CHOS into an additional bitmap named *SeqCov* (line 23).

Our heap-related instruction analysis is presented in Algorithm 2, which takes a program as input and returns a map recording the heap-related instructions and their corresponding codes. To collect as many heap operations as possible, we record the pointer arguments in the map *HeapInstMap* with an uninteresting code –1 for each function (lines 4–7). The analysis proceeds instruction by instruction. If the instruction is involved a memory allocation function, such as `malloc()` and `realloc()`, it is recorded with the interesting code *ALLOC* (lines 11–12). Similarly, if the instruction is involved a memory deallocation function, such as `free`, it is recorded with the interesting code *DEALLOC* (lines 13–14). If the instruction is a *load* one (*i.e.*, memory reading operation) and the corresponding address is in the map (*i.e.*, interesting address), it is recorded with the interesting code *LOAD* (lines 15–16). The *store* instruction (*i.e.*, memory writing operations) is similar, but with two cases (lines 17–22). Finally, to avoid an excessive number of distinct memory addresses, we consider operations on arrays as operations on their corresponding base addresses (lines 23–24) and ignore the cast operations (lines 25–26). Note that, this analysis can be merged into the LLVM-instrumentation pass to optimize instrumentation efficiency. For better understanding, we keep them separately.

3.4 Fuzzing Loop and Seed Selection

The fuzzing loop of CtxFuzz is presented in Algorithm 3, which takes the instrumented program P' (obtained by Algorithm 1) and a set of initial seeds S as inputs and returns a set of crashes *crashSet*. The algorithm initialize the seed pool *Queue* as the initial seeds S (line 2). Then the algorithm performs the following process until interruption: it selects a seed from the seed pool *Queue* as the input (line 4) and assigns the input a *energy* value (line 5), which

Algorithm 3: Fuzzing Loop of CtxFuzz

```
   Input: Instrumented program P', Initial seed inputs S
   Output: Set of crashes crashSet
1  crashSet ← ∅
2  Queue ← S
3  while not interruption do
4      input ← SelectSeed(Queue)
5      energy ← AssignEnergy(input)
6      for i = 0 to energy do
7          testcase ← Mutate(input)
8          BranchCov, SeqCov, SeqCount ← Execute(P', testcase)
9          if testcase triggers crash then
10             crashSet ← crashSet ⋃ {testcase }
11         else if isInteresting(BranchCov, SeqCov) then
12             Queue ← Queue ⋃ {testcase } // Interesting seed
13             update_bitmap_score(testcase, SeqCount) // Adjustment of seed order
14     end
15 end
16 return crashSet
```

determines the number of children (*i.e.*, testcases) to be generated from that input, following the same heuristics as AFL++ [7]. After that, for each mutated testcase, it monitors the execution of the instrumented program P' (line 8). If the program crashes or triggers alarms set by sanitizers (line 9), the testcase is recorded as a Proof of Concept (PoC) (line 10). Otherwise, if the testcase is interesting (*e.g.*, new code coverage or CHOS coverage) (line 11), the testcase is added into the seed pool for further testing (line 12) and the score, used to guide the seed selection, is updated correspondingly (line 13).

As mentioned in Sect. 3.1, CtxFuzz follows the workflow of AFL++, but introduces new seed selection strategies (lines 12–13). First, a testcase is considered interesting if and only if it introduces either new code coverage or new CHOS coverage, as shown in Eq. (1).

$$isInteresting = \begin{cases} True, & hasNew(BranchCov) \vee hasNew(SeqCov) \\ False, & otherwise \end{cases} \tag{1}$$

Moreover, we also improve the score-updating API *update_bitmap_score()* of AFL++ (line 13), which allows for the adjustment of the seed queue's order and the designation of certain seeds as favored. These favored seeds are then given priority for mutation in subsequent iterations (line 4). Initially, AFL++ prioritizes seeds based on shorter execution times and smaller file sizes. However, we enhance this strategy by also favoring seeds that exhibit a greater number of CHOS during testing. Formally, a favored testcase is defined by Eq. (2).

$$isFavored = \begin{cases} True, & isMin(Time \times FileSize) \vee isMax(SeqCount) \\ False, & otherwise \end{cases} \tag{2}$$

4 Evaluation

We have implemented a prototype of CtxFuzz based on AFL++ [7] version 4.08c. Our main focus is on modifying the influencing factors in the instrumentation, feedback mechanism, and seed selection. By making these modifications

Table 1. Objective real-world programs evaluated in our experiment.

Program	Version	SLoC	Input Format	Test instruction
Bento4	v1.6.0-639	104K	mp4	mp42hls @@
cflow	1.6	80K	c	cflow @@
cxxfilt	2.41	5.03M	text	cxxfilt -t
Exiv2	v0.26	386K	jpg	exiv2 -pX @@
giflib	5.2.1	14K	gif	gif2rgb @@
mJS	2.20.0	43K	js	mjs -f @@
OpenH264	8684722	141K	text	h264dec @@ ./tmp
YARA	v3.5.0	62K	text	yara @@ strings
Yasm	9defefa	173K	asm	yasm @@

without changing other components, we have successfully improved the overall performance of heap-based memory vulnerability detection. To foster further research, more information (*e.g.*, benchmark dataset, initial seed) is available at https://sites.google.com/view/ctxfuzz.

We conducted comprehensive experiments to evaluate CtxFuzz using a set of real-world programs, and compare CtxFuzz with state-of-the-art fuzzers, following Klees's suggestions [8]. In the experiments, we aim to answer the following questions:

RQ1. How effective is CtxFuzz in discovering heap-based memory vulnerabilities in real-world programs?
RQ2. How does CtxFuzz compare to other state-of-the-art fuzzers?
RQ3. Does the improvements made by CtxFuzz contribute to the efficiency of fuzzing towards heap-based memory vulnerabilities?

4.1 Experiment Setup

Benchmark Programs. We curated a collection of benchmark applications from fuzzing papers that focus on heap-based memory vulnerabilities, as shown in Table 1. The selection process took various factors into consideration, such as popularity, frequency of testing, and the presence of memory vulnerabilities. In total, we used 9 widely used real-world programs for our evaluation, all of which contain memory vulnerabilities. These programs include renowned code analysis tools (*e.g.*, cxxfilt, cflow), code processing tools (*e.g.*, mJS, Yasm), graphics processing libraries (*e.g.*, Exiv2, giflib), video processing tools (*e.g.*, Bento4, OpenH264), and a data processing library (*e.g.*, YARA), among others.

Baseline Fuzzers. We evaluated CtxFuzz by comparing it against 5 state-of-the-art fuzzers: AFL, AFL++, HTFuzz, Memlock, and TortoiseFuzz. These fuzzers were selected based on several considerations. AFL++ [7] is an enhanced version of AFL, and our work is built upon it. Memlock [26] and TortoiseFuzz [25]

Table 2. The number of heap-based memory vulnerabilities found in 48 h.

Program	CtxFuzz		AFL		AFL++		HTFuzz		MemLock		TortoiseFuzz	
	Unique	Average	Unique	Average	Unique	Average	Unique	Average	Unique	Average	Unique	Average
Bento4	3	3.00	2	1.88	3	3.00	3	2.88	2	2.00	1	1.00
cflow	2	2.00	2	1.12	3	2.12	3	1.75	1	0.62	1	0.38
cxxfilt	1	0.12	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
Exiv2	2	2.00	1	1.00	1	1.00	2	2.00	1	1.00	1	1.00
giflib	1	0.88	1	0.12	1	0.62	1	0.38	1	0.12	1	0.25
mJS	11	3.25	10	5.25	4	1.50	4	1.75	0	0.00	4	3.38
OpenH264	5	2.38	2	1.00	3	2.25	2	2.00	0	0.00	1	0.62
YARA	5	3.88	4	2.88	5	3.88	5	4.12	3	1.00	5	3.25
Yasm	11	6.88	9	7.88	11	8.50	9	8.50	2	0.88	9	7.00
Sum/Sum	41	24.39	31	21.13	31	22.87	29	23.38	10	5.62	23	16.88

specifically focus on memory operations, including memory allocation, deallocation, and usage. HTFuzz [28] targets memory vulnerabilities through candidate heap operation sequences. All of these fuzzers are partially related to CtxFuzz, and we ran them with their default parameters.

Performance Metrics. Instead of considering unique crash numbers reported by the fuzzers, We evaluate the performance of the fuzzers based on the number of heap-based memory vulnerabilities. To compare the vulnerability findings of different fuzzers, we also conduct the Mann-Whitney U-test for statistical evaluation.

Configuration. To ensure accurate results, we implemented two measures to mitigate performance jitter during the fuzzing process. Firstly, we conducted extensive testing by running each program for a duration of 48 h, allowing the fuzzer to reach a relatively stable state. Secondly, we repeated each experiment 8 times to minimize random noises that may occur during fuzzing. The applications were compiled using clang and AddressSanitizer [19,32], and we executed them in various fuzzers using the command options outlined in Table 1.

Experiment Infrastructure. All our experiments are performed on machines with an Intel(R) Xeon(R) Gold 6132 CPU @ 2.60 GHz and 252 GB of RAM under 64-bit Ubuntu LTS 20.04.

4.2 RQ1. Vulnerability Detection Capability

Table 2 presents the statistical results of CtxFuzz in detecting heap-based memory vulnerabilities. As shown in the column labeled CtxFuzz, CtxFuzz successfully detected 41 heap-based memory vulnerabilities in our eight 48-hour experiments, out of a total of 46 vulnerabilities. We manually reviewed these 41 bugs to assess their authenticity and severity. These vulnerabilities, classified by bug type, include stack buffer overflow, heap buffer overflow, use-after-free, and double-free vulnerabilities, many of which are critical and can have severe consequences. Among the vulnerable programs, mJS and Yasm stand out with

Table 3. The zero-day vulnerabilities found by CtxFuzz.

Bug ID	Program	Version	Type
CVE-2023-49554	Yasm	9defefa	use-after-free
CVE-2023-49551	mJS	2.20.0	null-pointer dereference
CVE-2023-49555	Yasm	9defefa	null-pointer dereference
CVE-2023-49557	Yasm	9defefa	null-pointer dereference
CVE-2023-49558	Yasm	9defefa	null-pointer dereference
CVE-2023-49549	mJS	2.20.0	null-pointer dereference
CVE-2023-49550	mJS	2.20.0	null-pointer dereference
CVE-2023-49553	mJS	2.20.0	null-pointer dereference
CVE-2023-49556	Yasm	9defefa	heap-buffer-overflow
CVE-2023-49552	mJS	2.20.0	stack-overflow

11 and 11 identified bugs, respectively. We promptly reported these previously unknown bugs to the respective maintainers.

Security Impact of Newly Found Vulnerabilities. CtxFuzz showcases its effectiveness and efficiency in discovering heap-based vulnerabilities by employing guided fuzzing of CHOS. At the time of writing, 10 zero-day vulnerabilities have already been identified, resulting in the assignment of 10 CVEs by Mitre, as shown in Table 3. Out of these 10 CVEs, 9 are heap-based memory vulnerabilities, while one is a stack overflow. These heap-based memory vulnerabilities discovered by CtxFuzz pose serious threats to the security and reliability of the affected applications. These include 7 null-pointer dereferences, one use-after-free vulnerability, and one heap-buffer-overflow vulnerability, which may result in memory corruption, denial-of-service, or arbitrary code execution [10,21]. Furthermore, some of these vulnerabilities are challenging to detect and exploit, with 5 of them requiring more than 48 h for successful exploitation, particularly if they involve complex heap operations or meta-operations. Consequently, it is crucial for software developers and security researchers to identify and address these vulnerabilities promptly.

4.3 RQ2. Compare with Other SOTA Fuzzers

Table 2 also presents the number of heap-based memory vulnerabilities detected by each baseline fuzzer (*e.g.*, AFL, AFL++, HTFuzz, MemLock, TortoiseFuzz) during a period of 48 h. CtxFuzz identified a total of 41 unique vulnerabilities, surpassing the second-best performers AFL++ which only uncovered 31 unique

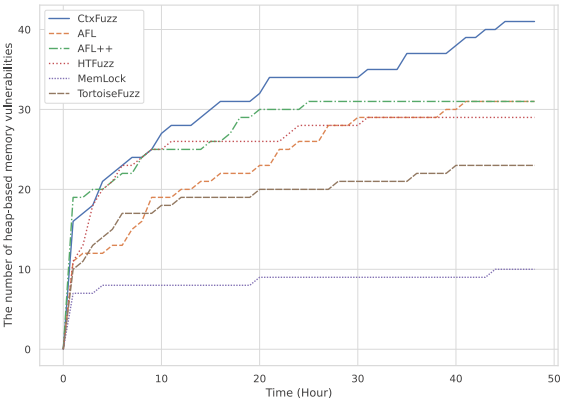

Fig. 2. Unique heap-based memory vulnerabilities accumulated over time

vulnerabilities. This represents a significant improvement, as CtxFuzz demonstrates a 32.26% enhancement over AFL++ in terms of detecting heap-based memory vulnerabilities. It is worth noting that, in some programs with frequent heap memory operations like OpenH264, CtxFuzz has a particularly significant impact, discovering over 100% more vulnerabilities than most baseline fuzzers. In terms of the average number of detected vulnerabilities, CtxFuzz slightly surpasses AFL++ and HTFuzz, while still significantly outperforming MemLock and TortoiseFuzz. In addition, although CtxFuzz detect a lower average number of vulnerabilities for some programs (*e.g.*, Yasm) than some baseline fuzzers (*e.g.*, AFL++ and HTFuzz), CtxFuzz is able to detect more or not less than the number of unique vulnerabilities. This could be partly attributed to the fact that CtxFuzz explores diverse CHOS across different runs, which highlights the necessity of exploring CHOS.

Discovered Vulnerabilities Over Time. To facilitate the comparison of different fuzzers, we also use a plot to illustrate the number of identified vulnerabilities in all 9 evaluated programs. This representation is depicted in Fig. 2. Within the initial 10-hour timeframe, CtxFuzz promptly identified nearly 27 vulnerabilities, showcasing its superior efficiency in vulnerability detection compared to other competitors. However, as the duration exceeds 30 h, the performance of the competing fuzzers gradually stagnates, with new vulnerabilities being uncovered at a slow pace. In contrast, CtxFuzz continues to discover new vulnerabilities. In summary, the plot clearly shows a consistent and robust growth trend in the discovery of vulnerabilities by CtxFuzz, surpassing other fuzzers and maintaining a leading position.

Statistical Test. According to the p-value of the Mann-Whitney U-test using CtxFuzz as the basis in Table 4, CtxFuzz outperforms the 5 compared fuzzers in 23 out of the 45 comparisons with a significant difference (*i.e.*, p-value is smaller than 0.05 [2]).

Table 4. P-values of Table 2. P-values below 0.05 are in bold.

Program	AFL	AFL++	HTFuzz	MemLock	TortoiseFuzz
Bento4	**1.03e-04**	1.00e+00	1.91e-01	**6.88e-05**	**6.88e-05**
cflow	**3.97e-04**	7.56e-01	8.50e-02	**1.55e-04**	**1.55e-04**
cxxfilt	1.91e-01	1.91e-01	1.91e-01	1.91e-01	1.91e-01
Exiv2	**6.88e-05**	**6.88e-05**	1.00e+00	**6.88e-05**	**6.88e-05**
giflib	**2.23e-03**	1.47e-01	**2.63e-02**	**2.23e-03**	**8.69e-03**
mJS	9.98e-01	**3.29e-03**	**1.01e-02**	**1.87e-04**	6.10e-01
OpenH264	**4.45e-03**	4.72e-01	8.55e-02	**1.35e-04**	**2.68e-04**
YARA	**2.29e-03**	5.00e-01	9.28e-01	**2.44e-04**	6.43e-02
Yasm	9.42e-01	9.86e-01	9.87e-01	**2.60e-04**	6.09e-01

Table 5. Results of tests with different hyperparameter settings.

Program	AFL++		K = 2, L = 3		K = 2, L = 8		K = 5, L = 3		K = 5, L = 8		CtxFuzz-wo	
	CHOS	Bug	CHOS	Bug	CHOS	Bug	CHOS	Bug	CHOS	Bug	CHOS	Bug
Bento4	19675	3	22235	3	21005	3	20883	3	19914	3	21586	3
cflow	19607	3	23129	2	23363	2	21619	2	21323	2	24062	2
cxxfilt	16091	0	20638	1	19702	0	19333	0	16795	0	21680	0
Exiv2	2170	1	3176	2	3300	2	2851	2	2576	2	3271	2
giflib	1719	1	3524	1	3698	1	3103	1	3088	1	3738	1
mJS	21381	4	27246	11	26256	8	25676	6	24749	2	26823	7
OpenH264	37423	3	40809	5	39707	3	38061	3	38855	2	42066	6
YARA	19611	5	22108	5	21493	5	20992	4	19940	5	23284	5
Yasm	45485	11	44088	11	41176	7	41917	9	37267	8	45665	9
Avg/Sum	20351	31	22995	41	22189	31	21604	30	20501	25	23575	35

4.4 RQ3. Ablation Studies

Hyperparameter Settings. As described in Sect. 3, the number K of the last accessed heap memory addresses and the number L of the last heap operations are configurable. To determine suitable values for K and L, we conducted experiments with different settings. As our testing goal is to explore more context heap operation sequences and discover more heap-based memory vulnerabilities, we naturally consider these two metrics. The results of tests conducted with different hyperparameter settings can be found in Table 5, where"CHOS" denotes the number of different CHOS found during runtime. We collect CHOS with afl-showmap [29] and map the CHOS found by other settings into the ones found by the setting of $K = 2$ and $L = 3$ for better comparasion. According to Table 5, we set the default configuration for CtxFuzz as $K = 2$ and $L = 3$ since it gives the best results.

Effectiveness of CHOS Feedback Mechanism. In the evaluation of CHOS feedback mechanism, we compared the performance of AFL++ and CtxFuzz without seed selection strategy (denoted as CtxFuzz-wo) in Table 5. The results show that CtxFuzz-wo explores over 15% more CHOS and discovering over 12% more vulnerabilities. In particular, it is able to detect more than nearly twice as many heap-based vulnerabilities as AFL++ on mJS and OpenH264. This indicates that the incorporation of CHOS feedback mechanism leads to more effective and targeted fuzzing, resulting in the discovery of additional heap-based vulnerabilities as compared to the baseline AFL++.

Effectiveness of Seed Selection. In evaluating the effectiveness of seed selection strategies, we compared the performance of CtxFuzz and CtxFuzz-wo in Table 5. The results indicate that CtxFuzz explores slightly fewer CHOS than CtxFuzz-wo. This difference can be attributed to the fact that CtxFuzz focuses on specific seeds that exhibit a greater number of CHOS to trigger potential vulnerabilities, which may sacrifice some diversity in CHOS generation. However, CtxFuzz is able to detect 6 more vulnerabilities. This suggests that the seed selection strategy contributes positively to the effectiveness of CtxFuzz in discovering heap-based vulnerabilities in real-world programs.

4.5 Discussion

Additional Experiments. The three aforementioned research questions illustrate the effectiveness and efficiency of CtxFuzz in detecting heap-based memory vulnerabilities. Additionally, we evaluated other secondary indicators, including code coverage, runtime overhead, and vulnerabilities beyond heap-based memory vulnerabilities. In the following, we provide a succinct and comprehensive explanation of the experimental findings. To estimate code coverage, we utilized `gcov` to obtain line coverage. The results showed that the coverage of CtxFuzz is comparable to that of baseline fuzzers. With respect to runtime overhead, we compared the total number of each baseline fuzzer's executions. CtxFuzz is slower in 32 out of 45 comparisons, ranging from 1.12 to 4.34 times. Although CtxFuzz does introduce some runtime overhead, we consider it worthwhile as it has been successful in discovering more vulnerabilities, even in resource-intensive scenarios. Furthermore, CtxFuzz primarily emphasizes the analysis of the heap operation sequence, which may limit its ability to detect other types of vulnerabilities (*e.g.*, the stack-overflow in Table 3 is a stack-based memory vulnerability). For readers interested in more detailed information, additional experimental results can be found on CtxFuzz's website.

Threat to Validity. We discuss the potential threats to the validity and generalizability of our study, as well as the measures we have taken to mitigate or control them. One potential threat is the selection bias that may arise from using only 9 open-source programs, which could limit the diversity of our dataset. To address this concern, we made sure to select diverse programs from various domains and with different characteristics. We are continuously working on improving and

evaluating CtxFuzz. Another potential concern entails the sampling error that may arise when utilizing a restricted number of seeds and inputs for each program and fuzzer. This limitation has the potential to impact the comprehensive nature of our testing process and introduce factors that create noise or variance in our findings. To address this, we employed an identical set of seeds for each fuzzer and conducted a 48-hour runtime. We repeated each experiment 8 times and reported the average and standard deviation to account for any potential variations. A third threat to consider is the possibility of statistical errors. In order to compare CtxFuzz with other testers, we utilized statistical tests and significance levels. However, it is important to acknowledge that these tests have certain assumptions and limitations. To address this concern, we thoroughly checked the assumptions and conditions before applying the tests, ensuring that we used the appropriate test for each specific scenario.

5 Related Work

Existing fuzzing solutions that aim to detect heap-based memory vulnerabilities can be broadly categorized into two groups: temporal memory vulnerabilities and spatial memory vulnerabilities. Use-after-free is a common temporal memory vulnerability, and there exist dedicated fuzzing techniques designed to detect Use-after-free vulnerabilities. UAFL [23] employs typestate analysis to detect operation sequences that may violate typestate properties. It utilizes operation sequence coverage as feedback to direct test generation and gradually cover the operation sequences. UAFuzz [15] relies on user-defined UAF sites to guide the fuzzer during exploration. However, these approaches rely on expert knowledge or imprecise static analysis, which presents challenges for their widespread adoption and effectiveness in practical scenarios. There are also some fuzzing techniques that can test for a wider range of temporal memory vulnerabilities. For example, HTFuzz [28] introduces heap operation sequences as new feedback, in addition to code coverage, to increase the diversity of heap operation sequences. However, these tools primarily concentrate on memory allocation and deallocation, neglecting the significant aspects of memory read and write operations. Many techniques primarily focus on identifying spatial memory vulnerabilities. Memlock [26] identifies statements and operations that are relevant to memory consumption and is guided by memory usage. Concolic execution-based smart fuzzing [13] only focuses on detecting heap-based buffer overflows as spatial memory vulnerabilities. However, it is important to note that CtxFuzz is capable of efficiently discovering both spatial and temporal memory vulnerabilities.

While there are other fuzzing techniques that target temporal and spatial vulnerabilities, they mainly focus on algorithm complexity vulnerability rather than memory vulnerability [1,3,9,17]. For instance, HotFuzz [3] is a framework that utilizes a genetic algorithm to generate inputs that lead to the worst-case performance of Java methods. In contrast, CtxFuzz specifically focuses on memory safety vulnerabilities, distinguishing it from HotFuzz's primary focus on algorithmic complexity vulnerabilities.

6 Conclusion

We proposed CtxFuzz, a fuzzing technique that leverages context heap operation sequences as a new feedback mechanism to efficiently discover heap-based temporal and spatial memory vulnerabilities. We evaluated CtxFuzz on 9 real-world programs and showed that it outperformed 5 state-of-the-art fuzzers in terms of discovering heap-based memory vulnerabilities. We also reported 10 new zero-day vulnerabilities (10 CVEs) to the corresponding vendors. CtxFuzz demonstrates its utility for testing real-world programs that are susceptible to heap-based memory vulnerabilities.

Acknowledgements. The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Natural Science Foundation of China (Nos. 62372304, 62302375, 62192734), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Fundamental Research Funds for the Central Universities.

References

1. Alsaeed, Z., Young, M.: Finding short slow inputs faster with grammar-based search. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1068–1079 (2023)
2. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 1–10 (2011)
3. Blair, W., et al.: Hotfuzz: discovering temporal and spatial denial-of-service vulnerabilities through guided micro-fuzzing. ACM Trans. Priv. Secur. **25**(4), 1–35 (2022)
4. Chen, Z., Liu, D., Xiao, J., Wang, H.: All use-after-free vulnerabilities are not created equal: an empirical study on their characteristics and detectability. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 623–638 (2023)
5. Du, C., Cui, Z., Guo, Y., Xu, G., Wang, Z.: Memconfuzz: memory consumption guided fuzzing with data flow analysis. Mathematics **11**(5), 1222 (2023)
6. Farkhani, R.M., Ahmadi, M., Lu, L.: {PTAuth}: temporal memory safety via robust points-to authentication. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1037–1054 (2021)
7. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, August 2020
8. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138 (2018)
9. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 254–265 (2018)
10. Liu, J., An, H., Li, J., Liang, H.: Detecting exploit primitives automatically for heap vulnerabilities on binary programs. arXiv preprint arXiv:2212.13990 (2022)

11. Lu, F., Tang, M., Bao, Y., Wang, X.: A survey of detection methods for software use-after-free vulnerability. In: Wang, Y., Zhu, G., Han, Q., Zhang, L., Song, X., Lu, Z. (eds.) Data Science. ICPCSEE 2022. CCIS, vol. 1629, pp. 272–297. Springer, Singapore (2022). https://doi.org/10.1007/978-981-19-5209-8_19
12. Meng, R., Dong, Z., Li, J., Beschastnikh, I., Roychoudhury, A.: Linear-time temporal logic guided greybox fuzzing. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1343–1355. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022)
13. Mouzarani, M., Sadeghiyan, B., Zolfaghari, M.: A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In: 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 42–49. IEEE (2015)
14. Mouzarani, M., Sadeghiyan, B., Zolfaghari, M.: A smart fuzzing method for detecting heap-based vulnerabilities in executable codes. Secur. Commun. Netw. **9**(18), 5098–5115 (2016)
15. Nguyen, M.D., Bardin, S., Bonichon, R., Groz, R., Lemerre, M.: Binary-level directed fuzzing for Use-After-Free vulnerabilities. In: 23rd International Symposium on Research in Attacks. Intrusions and Defenses (RAID 2020), pp. 47–62. USENIX Association, San Sebastian, October 2020
16. Novark, G., Berger, E.D.: Dieharder: securing the heap. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 573–584 (2010)
17. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2155–2168 (2017)
18. Sarda, S., Pandey, M.: LLVM Essentials. Packt Publishing Ltd., Birmingham (2015)
19. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: a fast address sanity checker. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 309–318 (2012)
20. Simpson, M.S., Barua, R.K.: Memsafe: ensuring the spatial and temporal memory safety of c at runtime. Softw. Pract. Exp. **43**(1), 93–128 (2013)
21. Tu, H.: Boosting symbolic execution for heap-based vulnerability detection and exploit generation. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 218–220. IEEE (2023)
22. Van Der Kouwe, E., Nigade, V., Giuffrida, C.: Dangsan: scalable use-after-free detection. In: Proceedings of the Twelfth European Conference on Computer Systems, pp. 405–419 (2017)
23. Wang, H., et al.: Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 999–1010. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020)
24. Wang, W., Fan, M., Yu, A., Meng, D.: Towards heap-based memory corruption discovery. In: 2021 17th International Conference on Mobility, Sensing and Networking (MSN), pp. 502–511. IEEE (2021)
25. Wang, Y., et al.: Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: NDSS (2020)
26. Wen, C., et al.: Memlock: memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 765–777. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020)

27. Younan, Y., Joosen, W., Piessens, F.: Efficient protection against heap-based buffer overflows without resorting to magic. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 379–398. Springer, Heidelberg (2006). https://doi.org/10.1007/11935308_27
28. Yu, Y., et al.: HTFuzz: heap operation sequence sensitive fuzzing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22, Association for Computing Machinery, New York, NY, USA (2023)
29. Zalewski, M.: American fuzzy lop (afl) fuzzer (2013). http://lcamtuf.coredump.cx/afl/
30. Zhang, G., Wang, P.F., Yue, T., Kong, X.D., Zhou, X., Lu, K.: Ovaflow: detecting memory corruption bugs with fuzzing-based taint inference. J. Comput. Sci. Technol. **37**(2), 405–422 (2022)
31. Zhang, T., Lee, D., Jung, C.: Bogo: buy spatial memory safety, get temporal memory safety (almost) free. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 631–644 (2019)
32. Zhang, Y., Pang, C., Portokalidis, G., Triandopoulos, N., Xu, J.: Debloating address sanitizer. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 4345–4363 (2022)