MemSpate: Memory Usage Protocol Guided Fuzzing

Zhiyuan Fu[1], Jiacheng Jiang[1], Cheng Wen[2](✉), Zhiwu Xu[1](✉), and Shengchao Qin[2]

[1] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
xuzhiwu@szu.edu.cn

[2] Guangzhou Institute of Technology, Xidian University, Xi'An, China
wencheng@xidian.edu.cn

Abstract. Memory safety vulnerabilities are high-risk and common vulnerabilities in software testing, often leading to a series of system errors. Fuzz testing is widely recognized as one of the most effective methods for detecting vulnerabilities, including memory safety ones. However, current fuzzing solutions typically only partially address memory usage, limiting their ability to detect memory safety vulnerabilities. In this paper, we introduce MemSpate, a dedicated fuzzer designed to detect memory safety vulnerabilities. Utilizing a more comprehensive memory usage protocol, MemSpate identifies the memory operation sequences that may violate the protocol and estimates the overall memory consumption to exceed an acceptable limit. It then monitors the coverage of these operation sequences and tracks the maximum memory consumption, both of which are used as a new feedback mechanism to guide the fuzzing process. We evaluated MemSpate on 12 real-world open-source programs and compared its performance with 5 state-of-the-art fuzzers. The results demonstrate that MemSpate surpasses all other fuzzers in terms of discovering memory safety vulnerabilities. Furthermore, our experiments have led to the discovery of 4 previously unknown vulnerabilities.

Keywords: Fuzz Testing · Memory Safety Vulnerability · Memory Usage Protocol · Software Testing

1 Introduction

Memory safety vulnerabilities are high-risk and common vulnerabilities in software testing, often leading to a series of system errors. Generally, memory safety vulnerability exists in two different forms: spatial [1,19] or temporal [5,22]. The former happens when a memory allocated with an invalid size, or program accesses the memory exceeds its spatial threshold (*e.g.*, stack overflow, memory allocation failure). The latter is due to data being used out of its life span (*e.g.*, use-after-free, double-free).

K. Ogata et al. (Eds.): ICFEM 2024, LNCS 15394, pp. 237–256, 2024.
https://doi.org/10.1007/978-981-96-0617-7_14

Detecting memory safety vulnerabilities is challenging, as they are influenced by numerous factors such as the frequency of heap operations, the specific execution order, and code coverage. Fuzz testing [6,9] is widely recognized as one of the most effective methods for detecting vulnerabilities, including memory safety ones. However, current fuzzing solutions typically only address memory usage partially. For example, MemLock [17] and TortoiseFuzz [16] focus on memory spatial vulnerabilities, while UAFL [15] and HTFuzz [20] concentrate on memory temporal vulnerabilities. The lack of comprehensive memory usage limits their ability to detect all types of memory safety vulnerabilities.

To address the limitations of current memory-related fuzzers, this paper proposes a new fuzzer named MemSpate. MemSpate is guided by a comprehensive memory usage protocol that addresses both memory temporal vulnerabilities and memory spatial vulnerabilities. Utilizing this protocol, MemSpate identifies the memory operation sequences that may violate the protocol and estimates the overall memory consumption to exceed an acceptable limit. It then monitors the coverage of these operation sequences and tracks the maximum memory consumption, both of which are used as a new feedback mechanism to guide the fuzzing process. By doing so, MemSpate can effectively detect more memory safety vulnerabilities, including both temporal and spatial ones.

We implemented a prototype of MemSpate based on AFL++ [6] version 4.09a, and evaluated 12 widely used real-world open-source programs. We compared MemSpate with five state-of-the-art memory-related fuzzers: AFL++ [6], MemLock [17], UAFL [15], HTFuzz [20] and TortoiseFuzz [16]. The results demonstrate that MemSpate outperforms the other fuzzers in terms of discovering memory safety vulnerabilities. Specifically, MemSpate is able to detect 33.33%, 140.00%, 30.43%, 46.34% and 76.47% more vulnerabilities than AFL++, MemLock, UAFL, HTFuzz and TortoiseFuzz, respectively. Furthermore, MemSpate discovered 4 new previously unknown that had not been reported by any other studies.

In summary, this paper makes the following contributions.

1. We proposed a comprehensive memory usage protocol that addresses both memory temporal vulnerabilities and memory spatial vulnerabilities.
2. We designed and developed MemSpate, a grey-box fuzzer that utilizes a more comprehensive memory usage protocol to efficiently detect memory safety vulnerabilities.
3. We evaluated MemSpate on 12 real-world programs and compared it with five state-of-the-art memory-related fuzzers. The results demonstrate that MemSpate outperforms the other fuzzers in terms of discovering memory safety vulnerabilities. Furthermore, our experiments have led to the discovery of 4 previously unknown vulnerabilities.

2 Motivation

Listing 1 demonstrates a null pointer dereference vulnerability that was discovered by MemSpate. This vulnerability is present in the program *yasm*, due to the

Listing 1. A null-pointer-dereference in *yasm*

```c
static MMacro *mmacros[NHASH];

static int expand_mmacro(Token *tline) {
  Token **params, *t, *tt;
  MMacro *m;
  Line *l, *ll;
  // ...
  t = tline;
  // ...
  m = is_mmacro(t, &params); // get macro from defined table
  // ...
  for (l = m->expansion; l; l = l->next) {
    // ...
    for (t = l->first; t; t = t->next) {
      Token *x = t;
      if (t->type == TOK_PREPROC_ID && t->text[1] == ’0’ && t->text[2] == ’0’
    ) // crash
      // ...
    }
  }
}

static MMacro *is_mmacro(Token *tline, Token ***params_arr) {
  // ...
  head = mmacros[hash(tline->text)];

  for (m = head; m; m = m->next)
    if (!mstrcmp(m->name, tline->text, m->casesense))
      break;
  if (!m)
    return NULL;
  // ...
  while (m) {
    // ...
    return m; // without checking every line is null
    // ...
  }
  // ...
}
```

absence of validation for the pointer t before it is accessed. Specifically, within the file nasm-preproc.c, the function `expand_smacro()` utilizes a Token pointer t to process the text stored in MMacro m. However, m is initialized by the function `is_macro()` without validating the line during expansion before returning m. The function `expand_smacro()` only checks the type of token t, leading to a crash when attempting to access the line text of macro m. During our experiments, current fuzzing tools such as AFL++, MemLock, and TortoiseFuzz were unable to detect this vulnerability within 24 h. This is because they did not take the memory temporal information into account.

Another example is a heap buffer overflow vulnerability in the program *binutils*, as illustrated in Listing 2. Specifically, the array *shndx_pool*, with size *shndx_pool_size*, is initialized within the function `prealloc_cu_tu_list()`. However, when the function `add_shndx_to_cu_tu_entry()` attempts to write data to the array within the function `process_cu_tu_index()`, no bound checking is performed on the array, resulting in a heap buffer overflow. Similarly, existing

Listing 2. A heap-buffer-overflow in *binutils*

```
  static void add_shndx_to_cu_tu_entry(unsigned int shndx) {
  shndx_pool[shndx_pool_used++] = shndx; // out of bounds
}

static void prealloc_cu_tu_list(unsigned int nshndx) {
  if (shndx_pool == NULL) {
    shndx_pool_size = nshndx;
    shndx_pool_used = 0;
    shndx_pool = (unsigned int *)xcmalloc(shndx_pool_size, sizeof(unsigned int));
  } else {
    shndx_pool_size = shndx_pool_used + nshndx;
    shndx_pool = (unsigned int *)xcrealloc(shndx_pool, shndx_pool_size, sizeof(unsigned int));
  }
}

static int process_cu_tu_index(struct dwarf_section *section, int do_display) {
  // ...
  unsigned char *shndx_list;
  unsigned int shndx;
  // ...
  if (!do_display) {
    prealloc_cu_tu_list((limit - ppool) / 4);
    for (shndx_list = ppool + 4; shndx_list <= limit - 4; shndx_list += 4) {
      shndx = byte_get(shndx_list, 4);
      add_shndx_to_cu_tu_entry(shndx); // entry
    }
    end_cu_tu_entry();
  }
  // ...
}
```

fuzzing tools such as AFL++, UAFL, and HTFuzz, which disregard memory spatial information, were incapable of detecting this vulnerability within 24 h.

Above all, existing fuzzing tools are unable to detect both of the aforementioned vulnerabilities. This is primarily because existing fuzzing solutions either overlook memory usage information or address it partially. As a result, there is a clear need for an enhanced fuzzer capable of identifying a broader spectrum of memory safety vulnerabilities in order to address this limitation.

3 Our Approach

3.1 Overview of MemSpate

The workflow of MemSpate is shown in Fig. 1, which consists of two main components: *static analysis* and *fuzzing loop*. MemSpate follows the general workflow of grey-box fuzzers but integrates improvements in both components guided by a memory usage protocol. In particular, the static analysis takes the program source code as the input, and generates the information related to the memory usage protocol, including control flow graph, call graph, memory operations, and memory sequences. Similar to the general grey-box fuzzers, the control flow graph information is utilized to gather the branch coverage, and the call graph

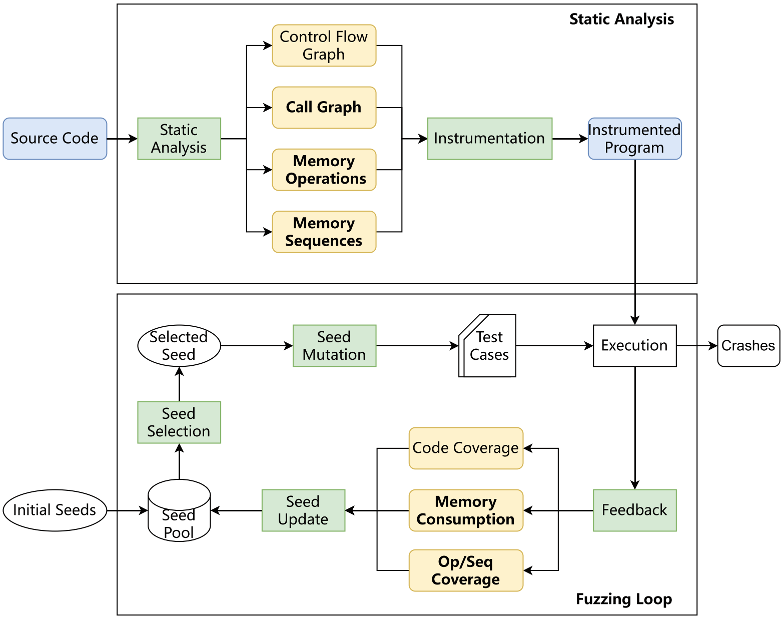


Fig. 1. Workflow of MemSpate

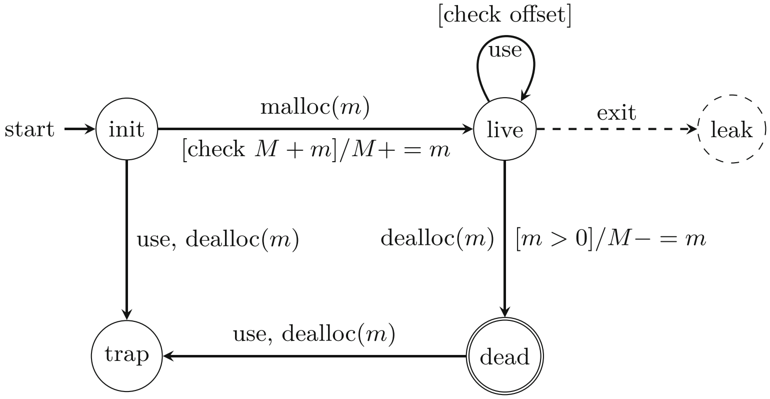
information is used to instrument the function call entries and returns. With guidance from the memory usage protocol, MemSpate identifies the locations of memory operations, calculates the (maximum) memory consumption, and analyzes the operation sequences that may violate the memory usage protocol. This information helps to determine where to instrument and what instrumentation is needed. Once the program is instrumented, MemSpate feeds it into the fuzzing loop for detecting memory safety vulnerabilities.

During the fuzzing loop, in addition to collecting the code coverage information, MemSpate also gathers information on memory consumption, memory operation coverage, and memory sequence coverage. MemSpate preserves the test cases that actively contribute to covering new code branches, consuming more memory, or covering new memory operation sequences, thus enhancing the likelihood of uncovering memory safety vulnerabilities.

3.2 Memory Usage Protocol

To detect a broader range of memory safety vulnerabilities, MemSpate employs a more comprehensive protocol to guide the fuzzing process.

The memory usage protocol used in MemSpate is represented as an automaton, which is illustrated in Fig. 2, where the conditions enclosed in square brackets denote the guards that the operation should satisfy, and if so, the corresponding action is performed. The protocol contains three kinds of memory operations, that is, memory allocation (e.g., malloc, new), memory deallocation (e.g., free,



M: Memory consumption
m: Allocated/deallocated memory size

Fig. 2. Memory Usage Protocol of MemSpate

delete) and memory use (e.g., read, write). And for both memory allocation and deallocation, a parameter m, representing the size of allocated/deallocated memory, is associated. To describe the effects of the memory operations, MemSpate uses four states, that is, *init*, *live*, *dead trap*, and *leak*. MemSpate also uses a global variable M to represent the current memory consumption. When a request for memory allocation of size m arises, MemSpate checks whether the sum of the current memory consumption M and the required memory size m exceeds a preset memory threshold (*i.e.*, the total system memory size). If this limit is exceeded, MemSpate may report a warning regarding potential *memory overflow* or *allocation failure*. Conversely, if there is sufficient available memory, the current memory consumption M is updated as $M + m$, and a new memory block is allocated with its state marked as *live* (transitioning from *init*). This allocated memory block can be used as long as the accessed points (offsets) fall within its defined bounds. And it remains in *live* until either it is deallocated or the program exits. In the latter case, a potential *memory leak* would be reported, as indicated by the dotted state in Fig. 2). While in the first case, this memory block becomes *dead* and then the current memory consumption is reduced to $M - m$. However, if any use or deallocation is performed on an already-deallocated memory block, then it goes into *trap* state, which indicates the detection of a *use-after-free* vulnerability or a *double-free* vulnerability. Similarly, if any use or deallocation is performed on an uninitialized memory address (in *init* state), it indicates the detection of a *null-pointer-dereference* vulnerability or an *invalid-free* vulnerability.

Function Call Stack. In addition to addressing heap memory overflow vulnerabilities, MemSpate also considers stack overflow vulnerabilities. Similarly, MemSpate utilizes a global variable D to represent the current stack depth.

Algorithm 1: Memory Operation Sequences Analysis

```
   Input: Original Program $P$
   Output: Memory Operation Sequences Set $Seq$
1  $Seq \leftarrow \emptyset$;
2  $S_M \leftarrow findMalloc(P)$;
3  $S_U \leftarrow findUsage(P)$;
4  $S_F \leftarrow findDealloc(P)$;
5  foreach $(s_m, m) \in S_M$ do
6      $A \leftarrow getAlias(m)$;
7      $A_F \leftarrow findDealloc(A, P)$;
8      $A_U \leftarrow findUsage(A, P)$;
9      $Seq \leftarrow Seq \cup reachAnalysis(s_m, A_F, A_U)$;
10     $Seq \leftarrow Seq \cup reachAnalysis(s_m, A_F, A_F)$;
11     $Seq \leftarrow Seq \cup checkOffset(s_m, A_U)$;
12     $S_U \leftarrow S_U - A_U$;
13     $S_F \leftarrow S_D - A_F$;
14 $Seq \leftarrow Seq \cup S_U$ // null-pointer-dereference
15 $Seq \leftarrow Seq \cup S_F$ // invalid-free
16 return $Seq$;
```

Upon function invocation, the variable D is incremented by 1; upon function return, D is decremented by 1. It is worth noting that recursive functions may be invoked excessively, gradually consuming the stack until it overflows.

3.3 Static Analysis

According to the memory usage protocol, any memory operation may introduce vulnerabilities if performed improperly (*i.e.*, violating the temporal rule of the protocol) or if cumulative memory consumption exceeds an acceptable limit (*i.e.*, violating the spatial rule of the protocol). Therefore, MemSpate will identify all potential memory operations within a program, which can be done during instrumentation. Furthermore, certain vulnerabilities arise from special memory operation sequences, such as use-after-free and double-free. For that, MemSpate will gather these memory operation sequences, particularly those with a length greater than 1, that result in the *trap* state.

Memory Operation Sequences. Inspired by UAFL [15], MemSpate conducts static analysis to identify the memory operation sequences that violate the temporal rule of the protocol. Algorithm 1 illustrates the basic idea, which takes a program P as input and outputs a set of memory operation sequences S.

To begin with, MemSpate gathers all memory allocation operations along with their corresponding memory objects in S_M (line 2). It also collects all memory use operations in S_U (line 3) and all memory deallocation operations in S_F (line 4). For each memory operation s_m and its associated object m, MemSpate employs pointer analysis to identify their potential aliasing pointers (line 6). Subsequently, MemSpate identifies all memory deallocation operations

Algorithm 2: Instrumentation

```
   Input: Original Program $P$
   Output: Instructioned Program $P'$
1  foreach function $f \in P$ do
2      foreach basic block $bb \in CFG_f$ do
3          if isEntryBB(bb) then
4              $stack\_depth \leftarrow stack\_depth + 1$;
5              $max\_stack\_depth \leftarrow max(max\_stack\_depth, stack\_depth)$;
6          foreach instruction $i \in bb$ do
7              if isReturnInst(i) then
8                  $stack\_depth \leftarrow stack\_depth - 1$;
9              if isAllocInst(i) then
10                 $size \leftarrow calculate\_size(i)$;
11                 $alloc\_size \leftarrow alloc\_size + size$;
12                 $max\_alloc\_size \leftarrow max(max\_alloc\_size, alloc\_size)$;
13             if isDeallocCond(i) then
14                 if isDeallocInst(i) then
15                     $size \leftarrow lookup\_size(i)$;
16                     $alloc\_size \leftarrow alloc\_size - size$;
17                 foreach sequence $seq \in$ getDeallocSequences(bb) do
18                     if $isLast(seq, bb)$ then
19                         $op\_seqs[seq \oplus bb] \leftarrow op\_seqs[seq \oplus bb] + 1$;
20                     else
21                         $pos \leftarrow findDeallocPos(bb, seq)$;
22                         $op\_seqs[pos \oplus seq] \leftarrow op\_seqs[pos \oplus seq] + 1$;
23             if isUseCond(i) $\vee$ isUseInst(i) then
24                 foreach sequence $seq \in$ getUseSequences(bb) do
25                     $op\_seqs[seq \oplus bb] \leftarrow op\_seqs[seq \oplus bb] + 1$;
26         $code\_cov[bb_{pre} \oplus bb] \leftarrow code\_cov[bb_{pre} \oplus bb] + 1$;
```

that deallocate the memory object m via an aliasing pointer, yielding a set A_F (line 7). Similarly, MemSpate finds all memory use operations related to any aliasing pointer of m, yielding a set A_U (line 8). With the assistance of reachability analysis, MemSpate adds into Seq the paths of the operation sequence $[s_m, s_f, s_u]$ if s_m can reach s_f and s_f can reach s_u. As well as the paths of the operation sequence $[s_m, s_f, s'_f]$ if s_m can reach s_f and s_f can reach s'_f, where s_u is a use operation from S_U while s_f and s'_f are two different deallocation operations from S_F (lines 9–10). Additionally, if feasible, it also checks the offset is appropriate for use operations (line 11). After that, MemSpate respectively remove the operations in A_F and A_U from S_F and S_U (lines 12–13). Finally, MemSpate appends the remaining deallocation and usage operations into Seq (lines 14–15) and returns Seq (line 16).

Instrumentation. In order to monitor memory usage information and adjust the fuzzing strategy in the fuzzing loop accordingly, we perform instrumentation to collect both code coverage information and memory-related information. Algorithm 2 illustrates the process of program instrumentation.

MemSpate follows the standard workflow of AFL [9] and AFL++ [6] for handling code coverage: MemSpate instruments every basic block in the program using *code_cov* (line 26).

In terms of stack consumption, MemSpate instruments the entry (line 3) and return instructions (line 7) of each function. It respectively increments (line 4) and decrements (line 8) the stack depth by 1 correspondingly. Furthermore, MemSpate keeps track of the maximum stack depth (line 5), which is beneficial for seed selection during the fuzzing loop. Concerning heap consumption, MemSpate instruments each allocation operation (line 9) and each deallocation operation (line 13).

MemSpate calculates the memory size *size* and increases the memory consumption *alloc_szie* by *size* for each allocation operation (lines 10–11); while MemSpate looks up the *size* and decreases the memory consumption *alloc_szie* by *size* for each deallocation (lines 14–15). Similarly, MemSpate keeps track of the maximum memory consumption (line 12), which is beneficial for seed selection during the fuzzing loop.

Finally, MemSpate utilizes a bitmap, *op_seqs*, to track the memory operation sequence coverage at the basic block level, as generated by Algorithm 1. As previously discussed, the operations in sequences may necessitate certain path conditions (*i.e.*, the branch basic blocks required by the paths). If an instruction is deallocated with its path conditions holding (line 14), MemSpate retrieves all the sequences associated with the deallocation (line 17). For each sequence *seq*, if the deallocation appears last in *seq* (*i.e.*, a double-free sequence), MemSpate identifies and marks the sequence as covered (lines 18–19); otherwise (*i.e.*, a subsequence of a use-after-free or double-free sequence), MemSpate locates the position of the deallocation in *seq* and marks that position as covered (lines 21–22). The use operation follows similar procedures (lines 23–25).

3.4 Fuzzing Loop

The fuzzing loop of MemSpate is outlined in Algorithm 3. It takes the instrumented program P' and a set of initial seeds T as inputs and returns a set of test cases that trigger crashes S and a set of test cases that trigger memory safety vulnerabilities S_{mem}.

MemSpate initializes the seed pool *Queue* as the initial seeds T (line 3). Then MemSpate performs the following process until timeout: it selects a seed *seed* from the seed pool *Queue* (line 5) and assigns the seed an energy value *testcase_num* (line 6), which determines the number of children (*i.e.*, testcases) to be generated from that seed (line 8), following the same heuristics as AFL++ [6]. After that, for each mutated testcase, MemSpate monitors the execution of the instrumented program P' and collects the information on code

Algorithm 3: Fuzzing Loop

```
    Input: Instrumented Program $P'$, Initial Seed $T$
    Output: Crashes Set $S$, Memory Safety Vulnerabilities Set $S_{mem}$
 1  $S \leftarrow \emptyset$;
 2  $S_{mem} \leftarrow \emptyset$;
 3  $Queue \leftarrow T$;
 4  while time ¡ timeout do
 5      $seed \leftarrow selectSeed(Queue)$;
 6      $testcase\_num \leftarrow assignEnergy(seed)$;
 7      for $i \leftarrow 1$ to $testcase\_num$ do
 8          $testcase_i \leftarrow mutate(seed)$;
 9          $code\_cov_i, op\_seqs_i, max\_stack\_depth_i, max\_alloc\_size_i \leftarrow$
            $fuzzRun(testcase_i, P')$;
10          if triggerCrash($testcase_i$) then
11              $S \leftarrow S \cup testcase_i$;
12              if triggerMemCrash($testcase_i$) then
13                  $S_{mem} \leftarrow S_{mem} \cup testcase_i$;
14          if hasNewCov($op\_seqs_i$) then
15              $Queue \leftarrow Queue \cup testcase_i$;
16          else if hasNewCov($code\_cov_i$) then
17              $Queue \leftarrow Queue \cup testcase_i$;
18          else if isLarger($max\_alloc\_size_i$, $max\_stack\_depth_i$) then
19              $Queue \leftarrow Queue \cup testcase_i$;
```

coverage, memory operation sequence coverage, maximum stack depth, and maximum memory consumption (line 9). If the program crashes, then the tasecase is added to the crash set S (lines 10–11). Moreover, if the crash is classified as a memory-related vulnerability by sanitizers, then the testcase is added into set S_{mem} as well (lines 12–13). Otherwise, if the testcase is considered to be interesting, meaning that it achieves new code branch coverage (line 14), introduces new memory operation sequence coverage (line 16), or results in larger stack/heap memory consumption (line 18), then the testcase is added into the seed pool for further testing (lines 15, 17, 19).

4 Evaluation

We have implemented a prototype of our memory usage protocol guided fuzzer MemSpate based on AFL++ [6] version 4.09a, wherein SVF [14], a static value-flow tool, is used to implement the static analysis. Our main focus is on modifying the influencing factors in the instrumentation and feedback mechanism. By making these modifications without changing other components, we have successfully improved the overall performance of memory-related vulnerability detection.

We conducted comprehensive experiments to evaluate MemSpate using a set of real-world programs, and compared MemSpate with state-of-the-art fuzzers. In the experiments, we aim to answer the following research questions:

RQ1. How effective is MemSpate in detecting memory safety vulnerabilities in real-world programs?
RQ2. How does MemSpate compare to other state-of-the-art fuzzers?
RQ3. Can the protocol of MemSpate assist in detecting memory safety vulnerabilities more comprehensively?

4.1 Experiment Setup

Benchmark Programs. We curated a collection of 12 benchmark applications from fuzzing papers focusing on memory safety vulnerabilities, as shown in Table 1.

Table 1. Real-world programs evaluated in our experiment

No.	Program	Version	LoC	Input Format	Test Instruction
1	bento4-640	1.6.0-640	106K	mp4	mp42hls @@
2	bento4-639	1.6.0-639	105K	mp4	mp42hls @@
3	binutils	2.40	4984K	elf	readelf -w @@
4	cflow-1.7	1.7	91K	c	cflow @@
5	cflow-1.6	1.6	80K	c	cflow @@
6	cxxfilt	2.40	4984K	text	cxxfilt -t
7	giflib	5.2.1	17K	gif	gif2rgb @@
8	mjs	9eae0e6	49K	js	mjs -f @@
9	openh264	8684722	141K	text	h264dec @@ ./tmp
10	yara	3.5.0	63K	text	yara @@ strings
11	yaml-cpp	0.6.2	122K	text	parse @@
12	yasm	9defefa	176K	asm	yasm @@

Baseline Fuzzers. We evaluated MemSpate by comparing it against five state-of-the-art fuzzers: AFL++ [6], MemLock [17], UAFL [15], HTFuzz [20] and TortoiseFuzz [16]. These fuzzers were selected based on several factors. Firstly, AFL++ is an improved version of AFL [9] and has established itself as one of the most widely used baselines in recent research papers. Both Memlock and TortoiseFuzz have proposed strategies for memory operations to discover memory spatial vulnerabilities, while UAFL and HTFuzz focus on detecting memory temporal vulnerabilities via feedback on memory operation sequences. Since UAFL is not publicly available, we made efforts to replicate it in our environment and refer to our replication of UAFL as UAFL†.

Evaluation Metrics. Since most baseline fuzzers and MemSpate focus on detecting memory safety vulnerabilities, we evaluate the performance of the fuzzers in terms of the number of memory safety vulnerabilities, instead of the number of unique crashes as traditional coverage-guided grey-box fuzzers do. Similar to TortoiseFuzz [16] and HTFuzz [20], we utilized AddressSanitizer [13] to analyze the crashes and to identify memory safety vulnerabilities. Additionally, we manually reviewed the crash reports to identify unique vulnerabilities and compared them with existing CVEs and GitHub issues to discover new vulnerabilities.

Experiment Configuration. Each experiment ran for 24 h, and the command options for the benchmark programs are listed in the last column of Table 1. According to Klees's suggestions [8], each experiment was conducted 10 times to minimize the influence of randomness.

Experiment Infrastructure. All the experiments were conducted using the same setup: a docker container configured with 1 CPU core of Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz and the 64-bit Ubuntu 18.04 LTS.

Table 2. Number of memory safety vulnerabilities found by different fuzzers

Program	MemSpate		AFL++		MemLock		UAFL†		TortoiseFuzz		HTFuzz	
	Uniq	Avg	Uniq	Avg	Uniq	Avg	Uniq	Avg	Uniq	Avg	Uniq	Avg
bento4-640	**1**	**0.10**	**1**	**0.10**	0	0.00	0	0.00	0	0.00	**1**	**0.10**
bento4-639	**4**	**4.00**	**4**	**4.00**	**4**	2.50	**4**	3.80	**4**	2.60	**4**	3.80
binutils	**1**	**0.30**	0	0.00	0	0.00	**1**	0.10	0	0.00	**1**	0.10
cflow-1.7	**4**	**1.60**	1	0.80	3	1.10	1	0.80	1	0.30	1	0.50
cflow-1.6	4	**2.80**	4	1.90	3	1.80	**5**	2.20	3	1.30	2	1.10
cxxfilt	**1**	**0.20**	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
giflib	**1**	**0.50**	**1**	0.30	**1**	0.10	0	0.00	**1**	0.40	**1**	0.10
mjs	**12**	**5.50**	7	4.90	3	2.10	9	**5.50**	7	4.90	7	5.20
openh264	**2**	**1.10**	1	1.00	0	0.00	1	1.00	1	0.90	1	1.00
yaml-cpp	5	2.70	5	2.40	**6**	**3.00**	5	2.60	5	2.30	0	0.00
yara	**9**	4.00	6	3.50	3	1.90	5	3.40	5	3.10	6	**5.30**
yasm	**16**	**10.90**	15	8.00	2	0.70	15	9.70	14	9.70	10	7.70
total	**60**	**33.70**	45	26.90	25	13.20	46	29.10	41	25.50	34	24.90

4.2 Memory-Related Vulnerability Detection Capability (RQ1)

The "MemSpate" column in Table 2 presents the results of MemSpate in detecting memory safety vulnerabilities, where the term "Uniq" denotes the total number of unique vulnerabilities found during the 10 runs while "Avg" denotes the average number of unique vulnerabilities among the 10 runs. As depicted in

Table 2, MemSpate successfully found a total of 60 memory safety vulnerabilities and an average of 33.70 ones across the 12 benchmark programs. Notably, MemSpate was able to identify more than 10 memory safety vulnerabilities in mjs and yasm.

Moreover, we manually reviewed the vulnerabilities and compared them with existing CVEs and GitHub issues. And we found that MemSpate is able to find 4 previously unknown memory safety vulnerabilities in yasm, which have been submitted in GitHub and are listed in Table 3.

Table 3. New memory safety vulnerabilities found by MemSpate

Program	Version	Report	Vulnerability Type
yasm	9defefa	Issue-273	null-pointer-dereference
		Issue-274	heap-use-after-free
		Issue-276	heap-buffer-overflow
		Issue-277	null-pointer-dereference

According to the above results, we conclude that MemSpate is capable of detecting memory safety vulnerabilities in real-world programs.

4.3 Comparison with Other Memory-Related SOTA Fuzzers (RQ2)

Table 2 also presents the number of memory safety vulnerabilities detected by each baseline fuzzer (e.g., AFL++, MemLock, UAFL†, TortoiseFuzz, and HTFuzz) on 12 benchmark programs over 24 h, where the numbers in bold indicate that the corresponding fuzzers achieve the best results. The results demonstrate that MemSpate achieves superior performance in terms of both total unique vulnerability number and average vulnerability number. Specifically, compared with AFL++, MemLock, UAFL†, TortoiseFuzz and HTFuzz, MemSpate can respectively identify approximately 33.33%, 140.00%, 30.43%, 46.34% and 76.47% more unique vulnerabilities in total, as well as respectively identifying 25.82%, 155.30%, 15.81%, 32.16% and 35.34% more vulnerabilities on average. Moreover, MemSpate demonstrates superior performance across most programs. In particular, on the program *mjs*, MemSpate identifies 12 memory safety vulnerabilities, which is significantly higher compared to any other fuzzers.

Figure 3 illustrates the trend of memory safety vulnerabilities found by each fuzzer over time. Within the initial 6-h period, all baseline fuzzers have reached a convergence point and successfully detected over 75.00% of the vulnerabilities within their detection scope. However, for MemSpate, 36.67% of detected vulnerabilities are explored in the later stage, resulting in a significantly higher number of detected vulnerabilities compared to the baseline fuzzers.

Based on the findings presented in Table 2, we have computed the p-value of the Mann-Whitney U-test between MemSpate and each baseline fuzzer. The

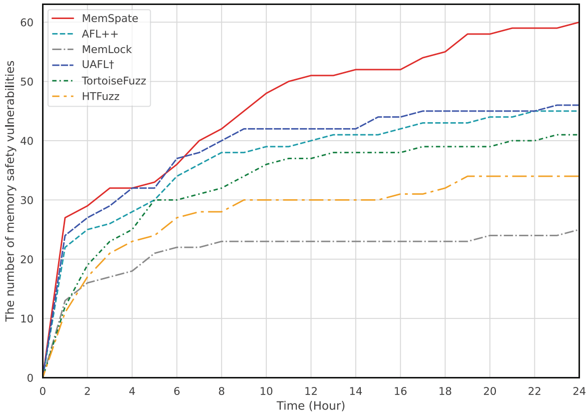


Fig. 3. Trend of memory safety vulnerabilities found by each fuzzer over time

Table 4. P-values of memory safety vulnerabilities found in 10 runs

programs	AFL++	MemLock	UAFL†	TortoiseFuzz	HTFuzz
bento4-640	5.29e-01	1.84e-01	1.84e-01	1.84e-01	5.29e-01
bento4-639	1.00e+00	**1.01e-04**	8.37e-02	**9.86e-05**	8.37e-02
binutils	**3.84e-02**	**3.84e-02**	1.50e-01	**3.84e-02**	1.50e-01
cflow-1.7	**1.81e-03**	**1.26e-02**	**1.81e-03**	**2.45e-04**	**6.77e-04**
cflow-1.6	**1.15e-03**	**8.56e-03**	**5.06e-03**	**8.92e-05**	**6.89e-05**
cxxfilt	8.37e-02	8.37e-02	8.37e-02	8.37e-02	8.37e-02
giflib	1.99e-01	**3.18e-02**	**6.83e-03**	3.47e-01	**3.18e-02**
mjs	1.15e-01	**3.95e-05**	6.22e-01	1.15e-01	2.64e-01
openh264	1.84e-01	**1.21e-05**	1.84e-01	9.58e-02	1.84e-01
yaml-cpp	1.02e-01	9.33e-01	3.03e-01	**4.46e-02**	**2.02e-05**
yara	8.09e-02	**1.63e-04**	6.63e-02	**8.60e-03**	9.99e-01
yasm	**7.42e-04**	**5.97e-05**	**4.00e-02**	**4.77e-02**	**5.97e-05**

outcomes are shown in Table 4, with values highlighted in bold indicating significance levels below 0.05. Our analysis reveals that MemSpate outperforms all the five compared fuzzers in 29 out of the 60 comparisons with a significant difference.

Overall, our experimental results demonstrate that MemSpate outperformed AFL++, MemLock, UAFL†, TortoiseFuzz, and HTFuzz in identifying memory safety vulnerabilities.

4.4 Effectiveness of Memory Usage Protocol (RQ3)

Figure 4 presents the number of memory safety vulnerabilities categorized by various types found by each fuzzer. The findings demonstrate that MemSpate is capable of detecting all 9 types of memory safety vulnerabilities, whereas other fuzzers fail to detect 2 or 3 types among the 12 benchmark programs. Furthermore, MemSpate has identified a total of 33 memory temporal vulnerabilities (including double-free, heap-use-after-free, invalid-free and null-pointer-dereference) and 27 memory spatial vulnerabilities (the others). In comparison, AFL++, MemLock, UAFL†, TortoiseFuzz and HTFuzz have found 27, 8, 28, 25 and 21 memory temporal vulnerabilities respectively. Additionally, they have identified 18, 17, 18, 16 and 13 memory spatial vulnerabilities. Notably, only MemSpate has the ability to uncover the invalid-free vulnerability.

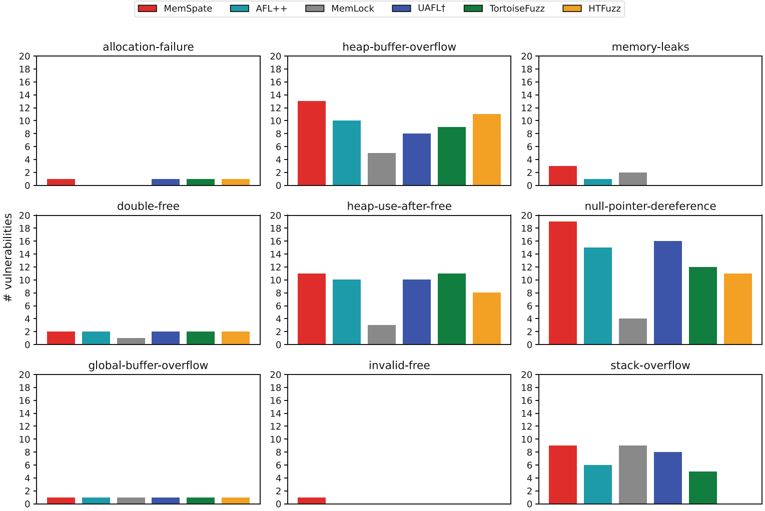


Fig. 4. Vulnerability numbers categorized by various types

Figure 5 presents upset plots illustrating the collective number of vulnerabilities discovered by different sets of fuzzers. The figure indicates that there exists a total of 12 vulnerabilities that can be found by all the fuzzers. Furthermore, MemSpate demonstrates the capability to exclusively identify 8 vulnerabilities, while both UAFL† and TortoiseFuzz can each uniquely identify one vulnerability. Among all the vulnerabilities, MemSpate only missed 6 vulnerabilities, whereas AFL++, MemLock, UAFL†, TortoiseFuzz, and HTFuzz missed a greater number at 21, 41, 20, 25, and 32, respectively.

Based on the above findings, we can deduce that the memory usage protocol of MemSpate contributes to a more comprehensive detection of memory safety vulnerabilities, encompassing both a greater number and variety of types.

4.5 Discussion

Overhead of Instrumentation. This paper presents our proposed fuzzer, MemSpate, which effectively detects memory safety vulnerabilities. However, due to the high expressiveness of preliminary static analysis, the instrumentation component introduces more overhead to the target program compared to other fuzzers. We calculated the average time (in seconds) spent on 10 runs of instrumentation, and the results are shown in Table 5.

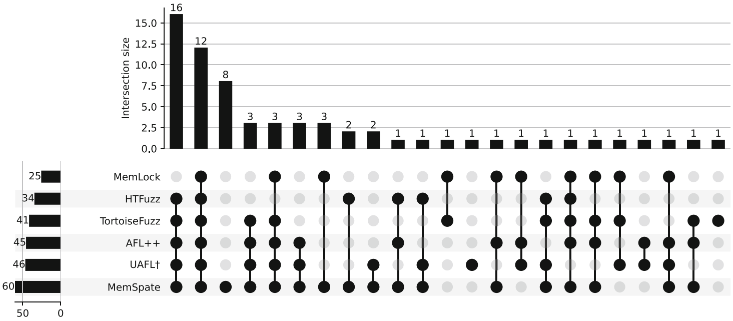


Fig. 5. UpSet Plot for MemSpate and five baseline fuzzers

It is evident that MemSpate consumes an average of 517.32 s to instrument the target programs, which is significantly higher than its base fuzzer AFL++ (3.74 ×). Considering our replication UAFL† of UAFL on MemSpate, we find that its average cost of instrumentation (3.70 ×compared to AFL++) is slightly lower than that of MemSpate. MemLock, TortoiseFuzz, and HTFuzz are built on top of AFL and may have a lower instrumentation overhead than AFL++. While MemLock's instrumentation is even slightly lower than AFL++ (0.90 ×), TortoiseFuzz and HTFuzz still have a higher instrumentation overhead than AFL++ (1.96 ×and 2.04 ×) due to the complex memory safety protocol they propose; however, both are far lower than MemSpate and UAFL†.

Upon thorough analysis, we assert that the high instrumentation overhead of MemSpate, along with the requirement for static analysis and additional preprocessing before conducting static analysis, also contributes to a considerable expense that cannot be ignored.

Moreover, an experiment was conducted to investigate the impact of our instrumentation on the fuzzing loop phase. The speed of the fuzzing loop for each fuzzer on each benchmark program was measured in terms of executions per second. Figure 6 illustrates the results from 10 runs. The findings indicate that the executions per second of MemSpate are not significantly lower than those of the other base fuzzers and even show significant improvement compared to the other base fuzzers on program *yara*.

In conclusion, the instrumentation cost of MemSpate in the static analysis stage results in a significant time investment. However, the time overhead of these static analyses (approximately 10 min) falls within acceptable limits when compared to the duration of the fuzzing loop test (24 h as we have set). Furthermore, the impact of instrumentation on the efficiency of fuzzing loop execution is negligible.

Table 5. Instrumentation overhead of fuzzers

programs	MemSpate	AFL++	MemLock	UAFL†	TortoiseFuzz	HTFuzz
bento4-639	466.90	182.10	140.40	436.00	174.40	152.00
bento4-640	461.10	193.10	133.90	454.10	185.40	150.10
binutils	1275.10	452.10	407.10	1285.10	907.40	867.30
cflow-1.6	243.70	42.10	44.10	245.50	65.20	58.80
cflow-1.7	266.10	47.80	48.80	285.50	74.30	64.20
cxxfilt	1865.60	449.50	432.00	1888.70	892.10	831.30
giflib	90.60	33.50	30.40	90.70	35.10	31.90
mjs	36.00	5.20	4.80	32.80	12.30	10.90
openh264	134.60	39.10	37.20	117.30	145.40	114.40
yaml-cpp	387.40	133.70	139.50	342.00	607.10	966.70
yara	457.10	33.70	32.90	470.10	58.20	51.70
yasm	523.60	47.50	41.60	490.60	93.30	81.90
Average	517.32	138.28	124.39	511.53	270.85	281.77

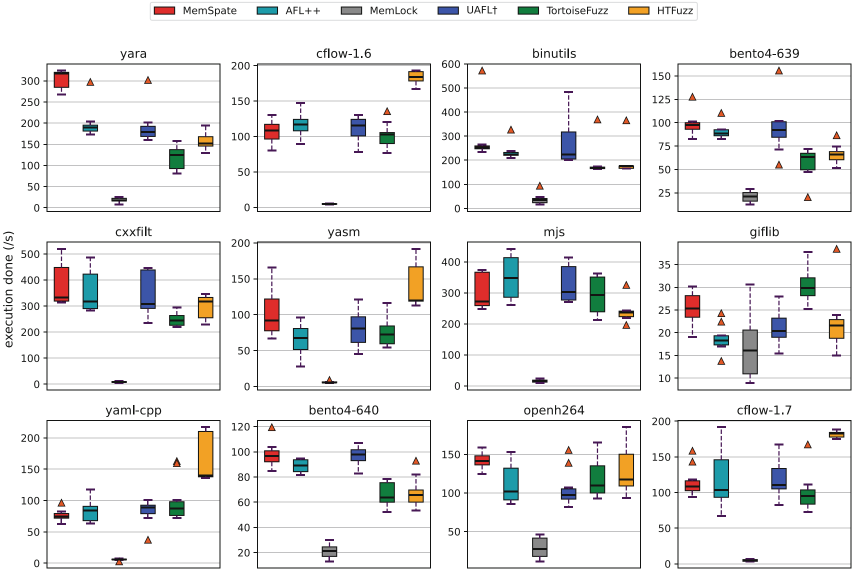


Fig. 6. Executions per second of each fuzzer in 10 runs

Threats 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 12 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 MemSpate. 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 24-h runtime. We repeated each experiment 10 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 MemSpate 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

There are various memory-related grey-box fuzzing solutions, whose target can be classified into two categories: memory spatial bugs and memory temporal bugs.

Fuzzing for Memory Spatial Bugs. Dowser [7], SAFAL [2] and a concolic execution-based smart fuzzing method [11] are specifically designed to detect buffer overflow vulnerabilities. MemFuzz [3] leverages information about memory accesses to guide the fuzzing process. MemLock [17] employs memory consumption information to guide the fuzzing process. MemConFuzz [4] extracts the locations of heap operations and data-dependent functions through static data flow analysis. ovAFLow [21] broadens the vulnerable targets to memory operation function arguments and memory access loop counts. TortoiseFuzz [16] proposes a new metric *coverage accounting* to evaluate coverage by security impacts (*i.e.*, memory operations), and introduces a new scheme to prioritize fuzzing inputs.

Fuzzing for Memory Temporal Bugs. FUZE [18] is a new framework to facilitate the process of kernel UAF exploitation. UAFuzz [12] relies on user-defined UAF sites to guide the fuzzer during exploration. UAFL [15] uses typestate automata to describe a memory temporal protocol of use-after-free vulnerability. HTFuzz [20] only focuses on the temporal memory vulnerability. LTL-Fuzzer [10] supports a linear-time temporal logic protocol, but requires expert knowledge to instrument program locations related to potential temporal violations. MDFuzz [23] identifies memory operation sequences as targets to guide the fuzzer without wasting resources exploring unrelated program components.

Note that existing fuzzing solutions typically only address memory usage partially, limiting their ability to detect all types of memory safety vulnerabilities. While MemSpate utilizes a more comprehensive memory usage, and thus is able to detect more memory safety vulnerabilities.

6 Conclusion

We have proposed MemSpate, a fuzzing technique that utilizes a more comprehensive memory usage protocol to efficiently detect memory safety vulnerabilities. We have evaluated MemSpate on 12 real-world programs, demonstrating its superior performance over 5 state-of-the-art fuzzers in terms of detecting memory safety vulnerabilities. Additionally, we have disclosed 4 previously unknown vulnerabilities to the respective vendors. This underscores MemSpate's efficacy in testing real-world programs that are prone to memory safety vulnerabilities.

Acknowledgements. This work was supported in part by the National Natural Science Foundation of China (Nos. 62472339, 62372304, 62302375, 62192734), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Fundamental Research Funds for the Central Universities.

References

1. Ba, J., Duck, G.J., Roychoudhury, A.: Efficient greybox fuzzing to detect memory errors. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–12 (2022)
2. Bhardwaj, M., Bawa, S.: Fuzz testing in stack-based buffer overflow. In: Advances in Computer Communication and Computational Sciences: Proceedings of IC4S 2017, vol. 1, pp. 23–36. Springer (2019)
3. Coppik, N., Schwahn, O., Suri, N.: Memfuzz: using memory accesses to guide fuzzing. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 48–58. IEEE (2019)
4. Du, C., Cui, Z., Guo, Y., Xu, G., Wang, Z.: Memconfuzz: memory consumption guided fuzzing with data flow analysis. Mathematics **11**(5), 1222 (2023)
5. 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)
6. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: {AFL++}: combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20) (2020)
7. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for {Overflows}: a guided Fuzzer to find buffer boundary violations. In: 22nd USENIX Security Symposium (USENIX Security 13), pp. 49–64 (2013)
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. LCAMTUF: American Fuzzy Loop (2017). https://lcamtuf.coredump.cx/afl/
10. 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 (2022)
11. 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)

12. 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 (2020)
13. 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)
14. Sui, Y., Xue, J.: SVF: interprocedural static value-flow analysis in LLVM. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 265–266. ACM (2016)
15. 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 (2020)
16. Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D., Su, P.: Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: NDSS (2020)
17. Wen, C., et al.: Memlock: memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 765–777 (2020)
18. Wu, W., et al.: {FUZE}: towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 781–797 (2018)
19. Ye, D., Su, Y., Sui, Y., Xue, J.: Wpbound: enforcing spatial memory safety efficiently at runtime with weakest preconditions. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, pp. 88–99. IEEE (2014)
20. Yu, Y., et al.: Htfuzz: heap operation sequence sensitive fuzzing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–13 (2022)
21. 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)
22. 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)
23. Zhang, Y., Wang, Z., Yu, W., Fang, B.: Multi-level directed fuzzing for detecting use-after-free vulnerabilities. In: 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 569–576. IEEE (2021)