Automated Exploitable Heap Layout Generation for Heap Overflows Through Manipulation Distance-Guided Fuzzing

Bin Zhang, Jiongyi Chen✉, Runhao Li, Chao Feng, Ruilin Li, and Chaojing Tang
National University of Defense Technology
{b.zhang, chenjiongyi, lirunhao, chaofeng, liruilin, chaojingtang}@nudt.edu.cn

Abstract

Generating exploitable heap layouts is a fundamental step to produce working exploits for heap overflows. For this purpose, the heap primitives identified from the target program, serving as functional units to manipulate the heap layout, are strategically leveraged to construct exploitable states. To flexibly use primitives, prior efforts only focus on particular program types or programs with dispatcher-loop structures. Beyond that, automatically generating exploitable heap layouts is hard for general-purpose programs due to the difficulties in explicitly and flexibly using primitives.

This paper presents SCATTER, enabling the generation of exploitable heap layouts for heap overflows in general-purpose programs in a primitive-free manner. At the center of SCATTER is a fuzzer that is guided by a new manipulation distance which measures the distance to the corruption of a victim object in the heap layout space. To make the fuzzing-based approach practical, SCATTER leverages a set of techniques to improve the efficiency and handle the side effects introduced by the heap manager's sophisticated behaviors in the real-world environment. Our evaluation demonstrates that SCATTER can successfully generate a total of 126 exploitable heap layouts for 18 out of 27 heap overflows in 10 general-purpose programs.

1 Introduction

Heap-based buffer overflows (*heap overflows*) are becoming one of the most prevailing threats to software [10]. Especially after the wide adoption of fuzzers such as AFL [24], Honggfuzz [17], and OSS-Fuzz [30], thousands of heap overflows are reported every year to NVD [11] and other platforms [1,2]. However, with the number of uncovered vulnerabilities continuing to grow, it still remains a manual and laborious task to assess whether those vulnerabilities can be exploited by attackers.

By transforming a crashing input into a working exploit, automated exploit generation (*AEG*) is a technique to assess the exploitability of vulnerabilities. After a decade of advancement, existing solutions work well on stack-based buffer overflows [4, 7, 22, 29]. Nevertheless, AEG is still unsatisfactory for heap overflows. To achieve control flow hijacking using a heap overflow, an attacker typically needs to provide a delicately-crafted input to place a victim object that contains a function pointer into the vulnerable object's overflowed region. Such placement requires strict control of heap allocations, which is challenging under the constraints of program execution logic and the heap manager's complicated mechanisms [18].

Existing approaches to constructing exploitable heap layouts can be categorized into two types: modeling-based approaches and fuzzing-based approaches. On one hand, Maze [32] leverages static analysis to extract heap primitives in interpreters and the programs with dispatcher-loop structures, analyzes their layout manipulation capabilities as well as dependency on each other, and set up linear Diophantine equations to model the layout manipulation problem. On the other hand, SHRIKE [20] analyzes the PHP statements to extract heap primitives for PHP interpreters, and then randomly inserts such heap primitives into a manually-written exploit template to generate exploitable layouts. Gollum [21] improves the performance by searching for test cases which have shorter distance between the address of the vulnerable object and the address of the victim object. However, existing modeling-based approaches and fuzzing-based approaches primarily rely on *explicit*, *powerful* and *easy-to-trigger* heap primitives that are easy to be utilized in specific programs such as interpreters or network service programs with dispatcher-loops (A comparison is given in Table 1). For example, PHP statement pair `$var = str_repeat("STR", x)` and `$var = 0` trigger a powerful primitive pair (`p=malloc(x)` and `free(p)`) [5], which can be used to implement almost *any layouts* that a working exploit needs. Still, for general-purpose programs which have no above properties of interpreters and network service programs, in the presence of complex execution logic that hinders the explicit and flexible use of heap primitives, together with the side effect introduced by the

heap manager's sophisticated behaviors in the real-world environment, there lacks a principled approach to automatically manipulate the heap layout to an exploitable state.

In this paper, we present SCATTER, a new fuzzing-based approach to automatically generate exploitable heap layouts of heap overflows for general-purpose programs. It works in a primitive-free way by directly mutating the crashing inputs without explicitly utilizing heap primitives. Specifically, SCATTER first discovers potential exploitable heap layouts leveraging static analysis and dynamic instrumentation to identify victim objects that can be placed into the vulnerable object's overflowed region. After that, it pinpoints heap operations whose arguments as well as execution times can be mutated from the program input, and maps those mutable operations to the input bytes. A highlight of this research is that we define a new manipulation distance which models the side effects introduced by the complicated behaviors of the heap manager, to measure the distance from the current heap layout to an exploitable layout in the heap layout space. Unlike bytes that measure the memory space, the heap layout space is organized by freed or occupied heap chunks. On this basis, we introduce a manipulation distance-guided fuzzer, whose goal is to decrease the manipulation distance until it becomes zero. To improve the fuzzing efficiency, SCATTER directly mutates the identified critical input bytes and selects interesting seeds that can increase the chance of successful manipulation.

We implemented a prototype of SCATTER on the `glibc` heap manager and evaluated it on 27 real-world heap overflows in 10 sample general-purpose programs. The evaluation results show that SCATTER can successfully generate exploitable heap layouts for 18 out of 27 vulnerabilities. On average, SCATTER generates 7 exploitable heap layouts for each vulnerability, and the average time to generate an exploitable heap layout is around one hour. The contributions of this research are summarized below.

- *New Problem.* This research aims to address the problem of exploitable heap layout generation for heap overflows of general-purpose programs, in cases where heap primitives are difficult to be explicitly utilized.
- *New Techniques.* We present a principled fuzzing-based approach that is guided by a new manipulation distance. A set of new techniques are proposed to make it efficient and practical.
- *Implementation and Evaluation.* We implemented a prototype and evaluated it on 27 real-world heap overflows in 10 sample general-purpose programs. It turns out that the tool can generate a total of 126 exploitable heap layouts for 18 out of 27 heap overflows. In the spirit of open source, we release this tool for continuous research[1].

[1] https://github.com/Epeius/Scatter

Table 1: Comparison with State-of-the-Art

Tools	Methods	Target Programs	Primitive-Free?	Modeling Side Effects?
SHRIKE [20]	Fuzzing	Interpreters	✗	*N.A.*
Gollum [21]	Fuzzing	Interpreters	✗	*N.A.*
Maze [32]	Modeling	Interpreters/CTFs	✗	✗
SCATTER	Fuzzing	General Programs	✓	✓

2 Background

2.1 Concepts in Heap-based Exploitation

To exploit a heap overflow, a common approach is to corrupt some critical data structures (e.g., an object that contains function pointers) in the heap using the overflowed object, which can lead to control flow hijacking or other read/write damages when de-referencing the corrupted pointers. The object used for the corruption is called a *vulnerable object*. The data structure that is corrupted or overwritten by the vulnerable object is called a *victim object*.

A ***Proof-of-Concept* (*PoC*)**, is a particular input that is manually crafted or produced by fuzzers and can cause the program to crash. An ***exploitable heap layout***, is a special heap layout which has at least one victim object locating in the vulnerable object's overflowed region. An ***exploitable PoC* (*ePoC*)**, is a special PoC that not only triggers the heap overflow but also constructs an exploitable heap layout when the heap overflow occurs. Even though ePoCs are not working exploits that hijack the function pointers for example, construction of the exploitable heap layout is crucial to develop working exploits as attackers can easily gain write-what-where or control flow hijacking capabilities based on ePoCs.

The concrete procedures to exploit a heap overflow typically involve the use of *heap primitives*. A heap primitive is a code snippet that invokes one or more heap operations (such as `malloc` and `free`), and it is often invoked multiple times with a certain program input. Attackers usually identify heap primitives and infer a specific sequence of primitives to construct the exploitable layout.

2.2 Heap Manager Internals

The heap manager manages the use of heap memory by providing (i.e., allocating) or collecting (i.e., freeing) pieces of memory regions and provides convenient interfaces for programs. For example, the mainstream heap manager *ptmalloc* implements four main interface functions, namely `malloc()`, `free()`, `calloc()`, and `realloc()`. If a program needs 20 bytes of heap memory, it invokes `malloc(20)` which returns a pointer pointing to the allocated memory chunk.

Memory chunk is the basic unit for a heap manager to organize the heap memory. It contains necessary metadata and a block of heap memory for programs. It is either in an *occupied* status or in a *freed* status. An occupied chunk is

being used by the program, and a free chunk is a ready-to-use chunk that is released by the program.

All free chunks in memory are linked to free lists. In order to efficiently locates a suitable free chunk for allocation, free chunks with the same size x are usually linked to the same free list $\mathbb{L}_x$. For example, when a program executes `malloc(0x200)` (whose chunk size is 0x210 bytes on 64 bit platforms), the heap manager will check the free list of 0x210 and return a free chunk in this list if the list is not empty. The in & out order of free chunks on the list can be either first-in-first-out (FIFO) or first-in-last-out (FILO).

For a free list $\mathbb{L}_x$, not only the heap operations that operate chunks with size x can affect the organization of $\mathbb{L}_x$, the side effect caused by the *split-merge mechanism* also dynamically changes the organization of chunks in $\mathbb{L}_x$. In particular, this mechanism enables two features:

- *Splitting chunks.* When there is no free chunk in $\mathbb{L}_x$ and an allocation of a chunk with size x occurs, the heap manager will split a larger free chunk with y bytes into two pieces. One is a piece of memory with x bytes. The rest is linked to a proper free list based on its size (i.e., $y-x$ bytes). For example, `malloc(0x200)` will cut 0x210 bytes from a free chunk of 0x230 bytes if there is no chunks in $\mathbb{L}_{0x210}$, and the remained 0x20 bytes of memory is linked to free list $\mathbb{L}_{0x20}$.
- *Merging chunks.* To organize the fragmented chunks in memory, the heap manager merges the adjacent free chunks to form a larger one. As illustrated in Figure 1, chunk c_b is a free chunk which is linked to $\mathbb{L}_{0x300}$. After executing `free(`c_a`)`, the heap manager merges c_a and c_b to form a free chunk with size 0x500 bytes. On the side of free lists, c_b is removed from $\mathbb{L}_{0x300}$ and a new free chunk c_a whose size is 0x500 bytes is linked to $\mathbb{L}_{0x500}$.

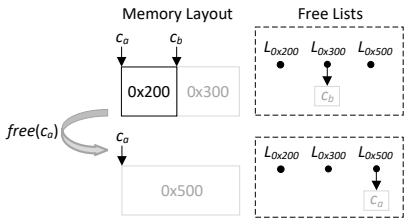

Figure 1: Chunk merging happens after executing `free(`c_a`)`.

Unlike interpreters which have powerful heap primitives (such as `malloc(x)` where `x` can be any given values), the heap operations in general-purpose programs are less attacker-controllable and always trigger the split-merge mechanism. For example, the PoC of the heap overflow vulnerability CVE-2017-11339 in `exiv2` triggers a total of 634 heap operations and 75% of them raise the split-merge mechanism for the chunks. The side effect on general-purpose programs makes it difficult to model the behaviors of heap memory allocation and the construction of exploitable heap layouts.

This paper focuses on the `glibc` heap manager, but our method is applicable to other heap managers if they obey the following rules:

- A heap operation sequence constructs the same memory layout in different runs.
- An allocation operation prefer to reuse the freed chunks with same size.
- A free list behaves either FIFO or FILO.
- Splitting should comply with the in & out order of the free list where chunk splitting happens.
- The size of free chunk used for splitting must be larger than the allocation size.
- Merging only happens between adjacent free chunks.

3 Motivation

3.1 A Running Example

The code in Listing 1 shows an example program whose input is the file specified by `argv[1]`. The program processes all home entries if the file contains entry information (line 26-27). It duplicates each entry name into `all_homes` by invoking `strdup()` (line 11), and then releases them at line 18 when the processing is finished. Then it continues to process the optional file information when `file_magic` is `FI` (line 32). It allocates a `FileInfo` object `c_file` at line 34, and fetches file names into `c_file->names` which are all heap buffers (line 35-40). After processing all optional blocks, the program allocates a buffer with size 0xF0 bytes which is used to store the pure data (line 44). However, it will trigger a heap overflow if the length of pure data is larger than 0xF0 bytes.

```
1  typedef struct FileInfo { // size is 0x20
2    char *names[4];
3  } FI;
4
5  int process_home_entries(char *data,uint32_t* home_num) {
6    char *all_homes[3]; int read = 0;
7    for (int i = 0; i < 3; ++i) { // loop l1
8      char *user_home_raw = data + read;
9      int home_len = strlen(user_home_raw);
10     if (home_len == 0xF0 || home_len == 0x20) {
11       all_homes[i] = strdup(user_home_raw); // m1
12       *home_num += 1;
13     }
14     ... // update read
15   }
16   ... // process all home entries
17   for (int i = 0; i < 3; ++i)  // loop l2
18     if (all_home[i]) free(all_home[i]); // f1
19   return read;
20 }
21
22 void main(int argc, char *argv[]) {
23   char *data = read_file(argv[1]);
24   uint32_t offset, home_num = 0;
25   // process optional entry information block if needed
26   if (data[0] == 'E' && data[1] == 'N')
27     offset += process_home_entries(data+2,&home_num);
28   char *file_magic = (char*)malloc(0x2); // m2
29   memcpy(file_magic, data+offset, 2);
30   FI *c_file = NULL;
31   // process optional file information block if needed
32   if (file_magic[0] == 'F' && file_magic[1] == 'I') {
33     offset += 2;
34     c_file = (FI*)malloc(sizeof(FI)); // m3
35     for (int i = 0; i < home_num; ++i) { // loop l3
```

```
36      uint8_t name_size = *(uint8_t*) (data + offset);
37      c_file->name[i] = (char*)malloc(name_size); // m4
38      memcpy(c_file->name[i], data+offset+1, name_size);
39      offset += (name_size+1);
40    }
41  } else
42    free(x); // f2
43  uint32_t data_len  = *(uint32_t*)(data+offset);
44  char *pure_data = (char*)malloc(0xF0); // m5
45  memcpy(pure_data, data+offset+4, data_len);  //overflow
46  ...
47 }
```

Listing 1: Code snippet that contains a heap OOB-write overflow.

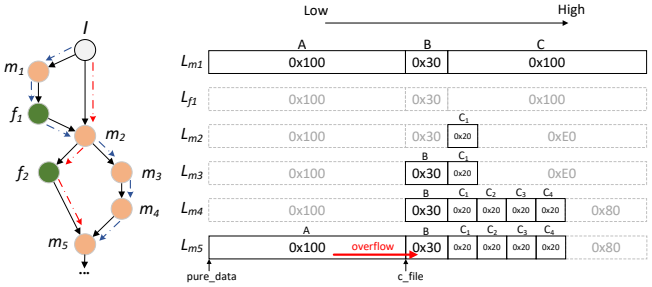

Figure 2: An example exploitable layout (in glibc 2.23) for the vulnerability in Listing 1. Node *I* is a fake initial root node to mark the startup of the program. The red and blue dashed arrows represent the execution traces of PoCs and ePoCs , respectively.

A PoC of this vulnerability is `\x00\x01\x00\x00T···T`, whose heap operation trace is the red dashed arrows in Figure 2. It has no optional blocks but only the pure data with `length` 0x100 bytes. Therefore, this PoC is not exploitable since it cannot allocate any victim objects.

An ePoC of this vulnerability is `ENa···a\x00b···b\x00c···c\x00FI\x02dd\x02dd\x02dd\x00\x01\x00\x00T···T`. This ePoC contains an entry block `EN` (the home entries' lengths are 0xF0 bytes, 0x20 bytes and 0xF0 bytes) as well as a file information block `FI\x02dd\x02dd\x02dd`. Its heap operation trace is shown as the blue dashed arrows in Figure 2. The layouts after executing m_1, f_1, m_2, m_3, m_4, and m_5 are labeled as L_{m_1}, L_{f_1} and so on. After processing the entry information, we get three free chunks as shown in L_{f_1}. Then `file_magic` at line 28 is allocated to C_1 by splitting free chunk C since there is no free chunks with size 0x20. As a result, m_3 will place `c_file` into chunk B. Since m_4 allocates buffers for `c_file->names`, the `name_size` at line 36 must be carefully selected (such as `\x02`) to avoid splitting free chunk A in L_{m_4}. The final exploitable layout is shown as L_{m_5}, where the vulnerable object `pure_data` is allocated to chunk A. And the overflowed data corrupts `names` array of victim object `c_file` which raises damages when de-referencing the `name` pointers in it.

We notice that it is hard to explicitly leverage heap primitives to construct the exploitable layout. For example, the number of times that m_4 executes depends on the logic in `process_home_entries`, which is difficult to model. Nevertheless, the exploitable layout can be constructed by directly transforming the initial PoC and exploring alternative program paths: if the first two bytes in the initial PoC are mutated to `EN` and the subsequent bytes meet the path constraint at line 10, we can trigger m_1 and f_1, which creates layout L_{f_1}. The `file_magic` at line 28 occupies chunk C_1 which leads to layout L_{m_2}. The victim object `c_file` at line 34 can be allocated to chunk B if we randomly insert some `FI` magics in the PoC which in turn creates layout L_{m_3}. Similarly, L_{m_4} and L_{m_5} can be progressively constructed by providing certain inputs that are mutated from the initial PoC.

3.2 Technical Challenges and Insights

It turns out that there is the potential to generate exploitable layouts by bridging the gap between mutating the program input and manipulating the heap layout, without explicitly utilizing heap primitives. However, achieving this goal needs to address the following challenges:

- **What is a desired exploitable layout?** Existing works like Gollum and Maze can flexibly specify a victim object or a desired layout as their inputs. However, it is difficult to specify desired layouts for general-purpose programs without using powerful primitives, because the creation of victim objects is highly dependent on the program's execution logic (e.g., `c_file` at line 34 in Listing 1 is created only when the input contains `FI` block). To tackle this challenge, we leverage static analysis and dynamic instrumentation to automatically identify a set of potential victim objects/feasible layouts based on the target program's concrete executions.
- **How to improve the efficiency of the fuzzing-based approach?** The mutation in the program input is to eventually control the heap operations. However, the metrics used in prior fuzzing-based approaches are coarse-grained, as they measure the manipulation objective using the distance in the memory space. A shorter distance in the memory space does not always map to less heap operations. To solve this challenge, given that the change on the distance in the memory space is controlled by the heap manager's allocations and frees, we propose a more accurate metric–manipulation distance–that is defined in the heap layout space. Additionally, we focus on the critical bytes in the input that can affect heap operations and leverage a multi-level scheduling strategies to increase the chance of successful manipulation.
- **How to model the side effects brought by complex heap behaviors so as to precisely control the manipulation?** The way the heap manager works is complicated. Not only does the normal allocation mechanism need to be modeled, but also the split-merge mechanism (discussed in Section 2) and the early occupation problem (discussed in Section 4.3.2) should be handled. To deal with those side effects, we model the diverse behaviors

by updating the distance measure at runtime and use an overload factor to mitigate the early occupation problem.

4 Design

A high-level design of SCATTER is shown in Figure 3. The inputs of SCATTER is the source code of the target general-purpose program as well as the PoC that triggers a heap overflow vulnerability, and the output are ePoCs that construct exploitable heap layouts. SCATTER firstly identifies potential victim objects that can be used for corruption through static analysis and dynamic instrumentation (Section 4.1). It then leverages a layout dependence graph to identify mutable operations, in order to further map those operations to some critical input bytes (Section 4.2). After that, it handles the side effects and leverages the manipulation distance metric to guide the fuzzer (Section 4.3). In the end, SCATTER focuses on mutating the critical input bytes and leverages a multi-level scheduling strategy to schedule the seeds and assign mutation energy (Section 4.4).

4.1 Identifying Victim Objects

Locating victim objects is key to determine the desired exploitable layout. SCATTER first leverages static analysis to identify sensitive structures and uses dynamic instrumentation to identify potential victim objects at runtime. Particularly, we focus on the following three types of sensitive structures:

- *A structure that contains pointers.* Hijacking pointers is a prevalent way to achieve control flow hijacking or other special capabilities in exploitation. For example, if an attacker controls a function pointer, then he/she is able to execute code at an attacker-specified address.
- *A structure that has no pointers but contains a member that can affect a buffer's access.* Take the code in Listing 2 as an example. Even though structure `IInfo` has no pointers, its member `w` is used as an array index for reading and writing buffer `raw_data`. If `w` is corrupted by a heap overflow, attacker can gain memory read and write capabilities at any offsets from `raw_data`. We leverage static value-flow analysis [31] to track whether the member would flow to an array index or other variables that control buffer access (e.g., the `n` parameter of function `memcpy()`).
- *A `union` structure that contains previous two types of structures and is accessed as its structure type.* A `union` is a piece of memory that is accessed using different implicit pointer casts. SCATTER identifies a sensitive `union` structure by statically tracking whether it is cast to a structure that contains a pointer or a member that can affect a buffer's access.

After recognizing the sensitive structures, SCATTER locates all victim objects allocation points by identifying heap operations that allocate or free memory for the sensitive structures (e.g., the `malloc()` at line 6 and the `free()` at line 15 in Listing 2). Since the variables' type information is lost after compilation, we need to analyze the type for the return of allocation functions. Again, SCATTER performs static value-flow analysis on LLVM IR to identify victim objects' allocations. It firstly collects the returned pointers from allocation functions like `malloc()`. Then it tracks whether the returned pointers are cast to pointers with sensitive structure types. For instance, the `malloc()` at line 6 in Listing 2 are compiled to `%3 = call i8* @malloc(i64 8); %4 = bitcast i8* %3 to %struct.ImageInfo*` in LLVM IR. Since `%3` is then cast to `struct.ImageInfo*`, we regard the `malloc` at line 6 as a victim object's allocation point.

```
1  typedef struct ImageInfo {
2    uint32_t w;
3    uint32_t h;
4  } IInfo;
5  void distort_image(char *raw_data) {
6    IInfo *info = (IInfo*)malloc(sizeof(IInfo));
7    ... // initialize w and h
8    for (int i = 0; i < info->h; ++i) {
9      char a = raw_data[(info->h-i)*info->w];
10     char b = raw_data[i*info->w];
11     raw_data[(info->h-i)*info->w] = b;
12     raw_data[i*info->w] = a;
13   }
14   ...
15   free(info);
16 }
```

Listing 2: Structure `IInfo` has no pointer members but the member `w` can affect memory access.

For each identified victim object allocation point, SCATTER marks its allocated chunk c as a victim object $o_s = \langle index, type\_id, addr \rangle$, where *index* is the object's allocation index in the execution trace, *type_id* is its static sensitive structure ID, and *addr* is the address of this object. Then SCATTER hooks the `free()` functions and determines whether its parameter points to a victim object o_s by checking all o_s's addresses at runtime. In doing so, we can dynamically collect the victim object set during execution.

4.2 Pinpointing Critical Input Bytes

To improve the fuzzer's efficiency by focusing on critical input bytes, in this step we identify heap operations whose parameters as well as execution times are mutable from the input and map the mutable operations to input bytes.

Identifying Mutable Operations. This step constructs and leverages the *Layout Dependence Graph* (*LDG*) to identify mutable operations. A LDG is a graph whose nodes are heap operations and edges are control flows. An example of LDG for the code in Listing 1 is shown in Figure 4, where node I is a fake root node to mark the startup of the program, node m_* and f_* represent the allocation and free operations respectively. The blue edges e_1, e_2 and $e3$ denote the `for`

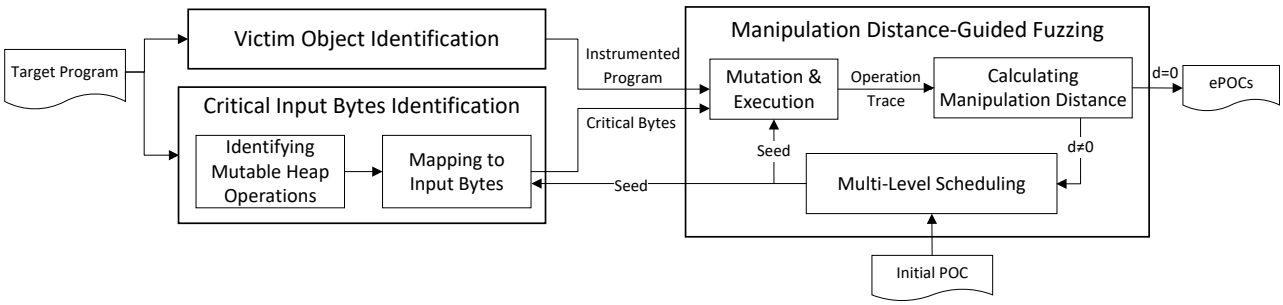

Figure 3: The system overview of SCATTER.

loops l_1, l_2 and l_3 respectively, while other edges describe the normal control flow.

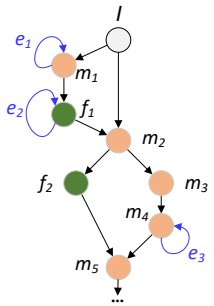

Figure 4: LDG of the example code in Listing 1.

Each vertex in LDG is represent as a two-tuple $v = \langle o, s_c \rangle$ where o represents the operation type (`malloc`, `free`, and so on) and s_c denotes the call stack of this heap operation which is generated at runtime. The purpose of involving call stack is to uniquely determine a heap operation. This is because different encapsulated heap operation functions could be regarded as the same heap operation, if only the last heap operation function is considered (e.g., in the program `sudo` [26], `sudo_reallocarray()` encapsulates `realloc()`).

To build a comprehensive LDG, SCATTER firstly utilizes a heap-operation-guided fuzzer to explore different execution paths, and then extracts heap operations and control flow information on each path. This fuzzer is designed to search for new heap behaviors by selecting test cases that trigger new heap operations or new heap function parameters as interesting seeds. For each round of seed selection, the heap operations of the new seeds are used to update the LDG, as those seeds could hit some new heap operations or trigger new cycles. At last, SCATTER merges all the extracted information on the paths to construct the LDG for the target program.

After building the LDG for target program, SCATTER identifies mutable heap operations by identifying whether the parameters or execution times of the operations change when giving different program inputs. To this aim, SCATTER leverages dynamic taint analysis technique, namely S2E's symbolic execution without forking symbolic states [9], to check whether an operation's parameters can be affected by the input bytes. To determine whether an operation's amount of execution is mutable, SCATTER analyzes the LDG to locate *cycles*. A cycle in the LDG means the heap operations within the cycle can be triggered multiple times. During the construction of the exploitable heap layout, this cycle can be leveraged by attackers and be repeatedly triggered. Given that the amount of execution of the cycles may not be explicitly data-flow dependent with the input, we use a statistical approach to identify the cycles whose amount of execution can be controlled from the input: first identify back edges from the directed LDG (such as e_1); Based on the identified back edges, SCATTER then transforms the raw graph to a DFS tree according to the control flows and uncovers the cycles[2]; Next, for each identified cycle, SCATTER marks the operations within the cycle as mutable if the cycle's amount of execution can vary with different inputs. For instance, the loop times of e_1 and e_2 in Figure 4 is not constant, indicating that m_1 and f_1 are both mutable operations.

Mapping to Input Bytes. SCATTER leverages a lightweight *"mutate-check"* strategy to locate input bytes that can affect mutable heap operations. At first, by feeding the program with a seed input, SCATTER collects the arguments and the occurrences of all mutable heap operations from the execution trace. Then for each byte in the input, SCATTER mutates it for several times and checks whether there are any mutable operations' occurrences or parameters are changed. If so, SCATTER collects the corresponding offsets in the input. For example, when mutating the `home entries` in the input for the example code in Listing 1, the occurrences and parameters of m_1 and f_1 are also changed, then the offsets of `home entries` are recognized as critical input bytes.

4.3 Modeling Fuzzing-Based Manipulation

The high-level workflow of manipulation is to firstly identify a victim object o_s, the vulnerable object o_v, and a free chunk in

[2]In graph theory, a back edge is an edge from a vertex to one of its ancestors. A graph G has a cycle if and only if it has a back edge with respect to a DFS tree.

o_v's overflowed region; then place the victim object o_s into the free chunk by implicitly invoking the statements or system calls that trigger heap operations. Such a process is often achieved by feeding the target program with certain inputs provided by an attacker. For general-purpose programs, in most cases the vulnerable object o_v is created after the victim objects, and the program triggers the vulnerability right after creating o_v. Therefore, SCATTER's strategy is to collect a set of victim objects, determine their locations in the heap, and adjust o_v's location according to the victim objects' locations through fuzzing, based on a manipulation distance metric that we defined in Section 4.3.1. More specifically, through dynamic instrumentation, we obtain all victim objects that are alive in heap when the vulnerability is triggered. For each victim object o_s, we traverse the trace $\vec{R}$ of heap operations to locate all suitable free chunks for placing the vulnerable object o_v, and calculate the manipulation distances. In the execution trace, a freed chunk is a potential location for the vulnerable object o_v to corrupt a victim object o_s only if the freed chunk meets the following requirements: (1) it is freed before allocating o_v and has the same size with o_v; (2) o_v can overflow into o_s when o_v is placed into this free chunk.

4.3.1 Defining Basic Manipulation Distance

We notice that it is inappropriate to guide a fuzzer by gradually shortening the difference between the object's address and the destination address [21]. A more suitable way is to define the distance in the heap operation space. To occupy a potential free chunk c with o_v, we need to make sure the heap operations $\vec{R}_o$ between c's free and o_v's allocation can remove all free chunks in c's free list whose allocation order are *earlier* than c. Therefore, we have the following equation:

$$n_A - \zeta_{\{0,1\}} \cdot n_F + 1 = \delta \tag{1}$$

where n_A and n_F denote the number of allocation and the number of free operations with the same size of c in $\vec{R}_o$, respectively. $\zeta_{\{0,1\}} = 0$ if c's free list is FIFO, otherwise, $\zeta_{\{0,1\}} = 1$ for FILO. δ represents the position index (start from 1) of c in the free list according to the allocation order. On this basis, we define the ***basic manipulation distance*** to corrupt a victim object o_s by placing the vulnerable object o_v into a free chunk c as follows:

$$\tau_d = |\delta + \zeta_{\{0,1\}} \cdot n_F - n_A - 1| \tag{2}$$

4.3.2 Handling Early Occupation Problem

Early Occupation Problem. Consecutive allocations in the operation trace occupy the target free chunk c which may not be released by the following free operations before allocating the vulnerable object o_v, which causes designated occupation to fail. We regard this case as *Early Occupation* problem. Take $\vec{R}_o = \langle f_2, m_1, m_2, f_3 \rangle$ in Figure 5 as an example. Assume

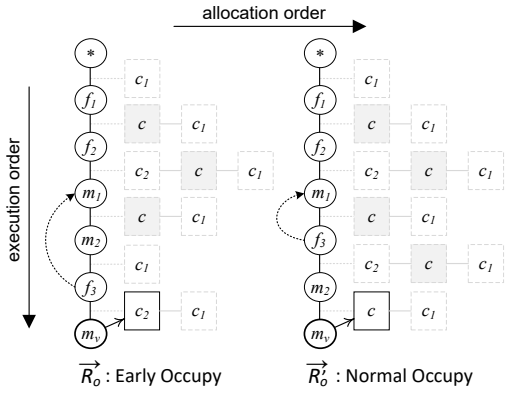

Figure 5: An example to illustrate the early occupation problem, where chunk c is early occupied by m_2 in $\vec{R}_o$ in FILO free list.

chunk c freed by f_1 is the target chunk where we want to place o_v into. According to (2), we have $\tau_d = |1 + 1 \cdot 2 - 2 - 1|$ equals 0, meaning that m_v is supposed to place o_v into chunk c. However, since c has been *early occupied* by m_2 and f_3 releases c_2 that is allocated by m_1, o_v will be placed into c_2 instead of c. On the contrary, $\vec{R}'_o = \langle f_2, m_1, f_3, m_2 \rangle$ in Figure 5 that has the same set of operations but no consecutive allocations can successfully place o_v into c.

The early occupation problem is very common in the exploitation of general-purpose programs. For example, a total number of 1926 heap operations are invoked before triggering the program `vim`'s heap overflow (vim-issue-2466). Many potential free chunks that are suitable to place the vulnerable object to corrupt victim objects are early occupied by the heap operations before allocating the vulnerable object.

Overload Factor. To deal with the early occupation problem, we introduce *overload factor* to describe the overall changes on the heap layout in the process of occupying the target free chunk c. Smaller overload factor indicates the change of target free list brought from operations before allocating o_v is more smooth, and the early occupation problem is less likely to occur. When the overload factor reduces to 0, the early occupation problem is eliminated.

Suppose our target is to place o_v into free chunk c, given a heap operation sequence $\vec{R}_o = \langle \sigma_1, \sigma_2, \cdots, \sigma_N \rangle$, its overload factor $\Delta_{\vec{R}_o}$ is given by:

$$\Delta_{\vec{R}_o} = \max_{1 \le \theta \le N} (\max(1 - \delta - \sum_{i=1}^{\theta} e_i, 0)) \tag{3}$$

where N refers to the number of heap operations in $\vec{R}_o$, δ denotes the position index of chunk c in its free list, θ is the operation index, and e_i denotes σ_i's affect to chunk c's free list which is defined below:

$$e_i = \begin{cases} \zeta & , \sigma_i \in \vec{R}_o \text{ frees a chunk from the target list} \\ -1 & , \sigma_i \in \vec{R}_o \text{ allocates a chunk from the target list} \\ 0 & , \text{else} \end{cases}$$

In this definition, $\sum_{i=1}^{\theta} e_i$ represents the accumulated changes of chunk c's free list raised by operations until σ_θ. Then after executing σ_θ, the allocation order of chunk c is updated to $\delta + \sum_{i=1}^{\theta} e_i - 1$. If this value is positive, it means that chunk c is still in the free list. Otherwise, the chunk may already be occupied by an allocation operation that occurs before σ_θ. Therefore, we get all negative values of $\delta + \sum_{i=1}^{\theta} e_i - 1$ where $1 \leq \theta \leq N$ by inverting and applying $max(1 - \delta - \sum_{i=1}^{\theta} e_i, 0)$ on it.

Given the fact that the lower this negative value is, the more efforts (like adding free operations or removing allocation operations) we need to deal with the early occupation problem. As such, we choose the maximum value of $max(1 - \delta - \sum_{i=1}^{\theta} e_i, 0)$ where $1 \leq \theta \leq N$ as the final overload factor of $\vec{R}_o$.

Table 2 lists the overload factors of $\vec{R}_o$ and $\vec{R}'_o$ in Figure 5. We can see that even though $\vec{R}_o$'s basic distance $\tau_d = 0$, it still cannot place the vulnerable object o_v into free chunk c since its overload factor $\Delta_{\vec{R}_o} = 1$. This overload factor means chunk c is already occupied by the operation that happens 1 allocation earlier before allocating o_v (i.e., m_2). On the contrary, $\vec{R}'_o$ can construct the desired layout because its $\Delta_{\vec{R}'_o}$ and τ_d are both reduced to 0.

Table 2: The Overload Factor for Record $\vec{R}_o$ and $\vec{R}'_o$ in Figure 5

θ	$\vec{R}_o(\tau_d = 0)$			$\vec{R}'_o(\tau_d = 0)$		
	σ_i	e_i	$\sum_{i=1}^{\theta} e_i$	σ_i	e_i	$\sum_{i=1}^{\theta} e_i$
1	f_2	+1	+1	f_2	+1	+1
2	m_1	-1	0	m_1	-1	0
3	m_2	-1	-1	f_3	+1	+1
4	f_3	+1	0	m_2	-1	0
Δ			1			0

δ for chunk c is 1, θ denotes the operation index, and Δ is the overload factor.

Based on the basic distance τ_d and the overload factor $\Delta_{\vec{R}_o}$, we have the ***extended manipulation distance*** d given below, to measure the distance in the heap operation space to corrupt a victim object:

$$d = \tau_d + \Delta_{\vec{R}_o} \tag{4}$$

4.3.3 Handling Split-Merge Mechanism

As mentioned in Section 2, the side effect caused by the split-merge mechanism makes it difficult to model the construction of the exploitable heap layout, which mainly affects the accurate calculation and update of manipulation distance. We model the split-merge mechanism based on concrete executions by analyzing the behaviors of heap allocation at runtime:

- If the returned chunk of an allocation operation comes from the free list whose chunk size is larger than the operation actually needs, chunk splitting occurs;
- If the chunk released by a free operation is not chained to its target free list, chunk merging occurs.

Suppose our target free chunk c's size is x and its free list is $\mathbb{L}_x$. To deal with the split-merge mechanism, we update n_A, n_F, and e_i in (4) to make a more accurate manipulation distance based on the allocation behaviors during concrete executions.

When Chunk Splitting Occurs. Case ①: an allocation operation whose size is less than x but returns a chunk from $\mathbb{L}_x$ is treated as an interesting allocation operation since it reduces the length of $\mathbb{L}_x$. In this case, we update (2) by adding 1 to the n_A; Case ②: an allocation operation whose size is x' but returns a chunk from $\mathbb{L}_{x+x'}$ is treated as an interesting free operation. Because it creates a remained free chunk with size x which will be linked to $\mathbb{L}_x$. In this case, we update (2) by adding 1 to the n_F.

When Chunk Merging Occurs. Chunk merging is more complicated than splitting. For a free operation `free`(c') where c''s size is y, the manipulation distance is updated in the following cases:

- $y \neq x$, and this free operation merges with one chunk in $\mathbb{L}_x$. If the merged chunk's allocation order is before the target free chunk c's, this operation is treated as interesting and n_A in (2) is increased by 1.
- $y \neq x$, and this free operation merges with another chunk and the result chunk's size is x. In this case, a new free chunk is constructed and chained to $\mathbb{L}_x$. This operation is regarded as an interesting free operation by increasing n_F in (2) by 1.
- $y \neq x$, and this free operation merges with another chunk and the result chunk's size is not x. This has no effect on both n_A and n_F in (2).
- $y = x$, and this free operation merges with one chunk in $\mathbb{L}_x$. In this case, no free chunks are chained but one free chunk removed from $\mathbb{L}_x$. So this operation is removed from interesting free operations by decreasing n_F in (2) by 1. Besides, if the merged chunk's allocation order is before chunk c's, we increase n_A in (2) by 1.
- $y = x$, and this free operation merges with chunk in another free list. In this case, we only decrease n_F in (2) by 1.

The update of e_i in (3) is similar to n_A and n_F's update as discussed above. With the defined manipulation distance metric, we calculate the distances in all potential freed chunks that could be used to place o_v in order to overwrite a victim object o_s, and choose the shortest distance value as the fuzzer's objective.

4.4 Manipulation Distance-Guided Fuzzing

For the PoC that triggers the vulnerability, SCATTER generates new PoCs by mutating the critical input bytes. During fuzzing, we extract the heap operation trace, the vulnerable object, and all alive victim objects when the overflow occurs. For each

victim object, SCATTER calculates the distance to corrupt it according to (4). By iteratively reducing the distance to 0, SCATTER generates the final ePoCs.

For accuracy and efficiency, there are two challenges that need to be addressed: how to determine a mutated PoC triggers the same vulnerability as the initial PoC does? which PoCs deserve higher priorities to fuzz and how much mutation energy should be assigned?

Validation of Mutated PoCs. In practical, ASAN's crash context report is a useful information to determine whether different PoCs trigger the same heap overflow. For two PoCs, if their vulnerable objects' allocation points and overwriting points from the ASAN's reports are same, we regard them as triggering the same heap overflow.

However, since ASAN leverages a customized heap manager to detect memory errors, the memory layouts generated from the same PoC in ASAN-enabled targets and ASAN-disabled targets are largely different. This would lead to inaccurate manipulation distance calculation. To tackle this problem, SCATTER disables ASAN and implements the following three instrumentation functions, to determine whether the PoCs trigger the same bug:

- `add_maybe_VO(addr, size)`. This function is invoked right after the creation of the vulnerable object. It collects the return pointer `addr` as well as its `size`, and sets a special tag "`MARK`" right after the allocated memory.
- `check_OOB_before()` & `check_OOB_after()`. These two functions are invoked before & after executing the overwrite statement to check whether the tag "`MARK`" inserted after the vulnerable object is changed or not[3].

Since the tag "`MARK`" could be corrupted by another earlier overwrite statement that differs from the one exposed in the initial PoC, we regard that a new PoC triggers the same vulnerability as the initial PoC when the tag stays unchanged before the statement but changes after executing the statement.

Scheduling Strategy. Unlike traditional fuzzers that typically aim to uncover vulnerabilities by maximizing the code coverage, SCATTER aims to increase the probability of producing exploitable layouts, by identifying more heap allocations whose sizes are controllable and diverse heap operation sequences. SCATTER is based on genetic algorithms and selects the seeds according to the following criteria:

- *Shorter distance.* The distance to corrupt a victim object is shortened. Shortening the manipulation distance to zero is the ultimate goal of manipulation.
- *More victim objects.* New victim objects are identified before the overflow occurs. This increases the chance of producing more exploitable layouts by adding more victim objects.
- *More free chunks.* More potential chunks are freed before allocating the vulnerable object. It means more optional positions for locating the vulnerable object.
- *Diverse heap operation sequences.* Before the overflow occurs, a new heap operation sequence is triggered. More combinations of heap operations increase the manipulation capabilities.

After collecting the interesting PoCs as seeds, SCATTER leverages a *multi-level scheduling strategy* to determine which seeds are prioritized and how much mutation energy should be assigned to them. The priorities for the criteria are (from high to low): shorter distance, more victim objects, more free chunks and diverse heap operation sequences.

The scheduling strategy is greedy: whenever it needs to pick up a seed from the queue, it firstly chooses to select the test cases with the shortest distance to corrupt a victim object. If all the test cases that shorten the distance are fuzzed, SCATTER chooses to fuzz the test cases that trigger the largest number of victim objects. Then, SCATTER focus on the test cases with the largest number of potential free chunks. After that, test cases that uncover new heap operation sequences are scheduled. In the end, the test cases that expand the code coverage are considered.

For each scheduled test case, SCATTER generates an expansion factor ε to adjust the mutation energy obtained by code coverage-based fuzzing (by multiplying the expansion factor with the original energy obtained from AFL's Havoc). The expansion factor ε is defined as follows:

$$\varepsilon = a \cdot \frac{1}{d} + b \cdot \frac{n_s}{N_s} + c \cdot \frac{n_c}{N_c} \tag{5}$$

where a, b, and c are all constant factors ($c < b < a$, meaning that more weights are assigned to the distance factor), d is the minimal manipulation distance[4], which is negatively correlated to the expansion factor. n_s denotes the number of victim objects uncovered in this PoC and N_s is the maximum number of victim objects exhibited by all PoCs. n_c represents the number of potential free chunks in this PoC and N_c is the maximum number of potential free chunks released by all PoCs. n_s/N_s and n_s/N_s are positively correlated to the expansion factor.

5 Evaluation

We implemented a prototype of SCATTER based on AFL fuzzing engine, S2E symbolic execution engine [9] and SVF static value-flow analysis engine [31]. We chose AFL instead of AFL++ [13] due to familiarity to AFL's codebase. We added over 5K lines of C/C++ code into AFL to facilitate the analysis of heap manager's behaviors, the calculation of the fine-grained manipulation distance. Besides, we implemented an S2E plugin which runs the target program with symbolic data but does not fork states, in order to perform accurate dynamic taint analysis to construct LDG. To support victim

[3] This only supports to detect the overflow that raised by continuous memory overwrite starting from the vulnerable object.

[4] A seed could contain multiple victim objects, corresponding to multiple manipulation distances.

object identification, we also implemented an SVF plugin based on its value flow graph. We evaluated SCATTER to answer the following research questions:

- **RQ1:** Can SCATTER generate ePoCs for heap overflows in real-world general-purpose programs?
- **RQ2:** Can SCATTER find more ePoCs than the state-of-the-art solutions?
- **RQ3:** Can SCATTER outperform state-of-the-art solutions regarding time consumption?

To answer the research questions above, we select 27 heap overflow vulnerabilities in 10 real-world general-purpose programs as our benchmark. Among these 27 vulnerabilities, only for CVE-2021-3156 of `sudo` exists a public exploit [28], while the others still remain unexploited publicly. The input types of our benchmark include executable files, command line arguments, images and raw text files. The vulnerabilities are selected based on the following rules: 1) The vulnerabilities cause heap OOB-write overflows and their PoCs are public; 2) The programs are open sourced general-purpose programs; 3) The programs do not implement their customized heap managers. All the experiments were conducted on a Ubuntu 18.04 LTS server (with default Glibc version 2.27) running with 128G RAM and Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz*70.

5.1 ePOC Generation (RQ1)

The ePoC generation results are shown in Table 3. Each case in our benchmark is fuzzed for 10 times, and each fuzzing campaign lasts for 24 hours. Given that the goal of an ePoC is to corrupt victim objects, ePoCs which corrupt the same victim objects that have been corrupted by other ePoCs are regarded as redundant ones in our experiments. By removing all the redundant ones, we get the *unique ePoCs* in which one ePoC corrupts a unique victim object. SCATTER successfully generates 126 unique ePoCs for 18 vulnerabilities which are all manually verified. The column of *# of Sensitive Struct. /Identified Victims* in Table 3 gives the results of victim object identification. On average, each heap overflow involves 377 sensitive structures and 89 identified victim objects, indicating that they widely exist in the programs. The column of *# of Mutable Cycles /Ops.* gives the results of mutable heap operations identification from LDG. SCATTER not only mutates the file-based inputs, but also the environment variables and command arguments. On average, we identified 212 mutable cycles and 135 mutable operations on each vulnerability.

Failed Cases. SCATTER fails to generate ePoCs for 9 vulnerabilities in our benchmark. After a careful manual examination, the causes of failure are given below:

- *Limited number of victim objects.* For CVE-2022-0359 and CVE-2022-0392 of `vim`, the number of the identified victim objects is only 3. For CVE-2019-16352 of `ffjpeg`, the target program is quite simple which has only 3K LoC and 8 defined structures (5 of them contain sensitive pointers). Meanwhile, the allocation of victim objects mostly occur before invoking any `free` functions. SCATTER failed to adjust the location of the vulnerable object to generate ePoCs.
- *Limited heap operations.* CVE-2021-4019 of `vim` occurs near the entry point of the whole program, which overflows at the third heap operation after entering `main()` function. We found that there are no chunks in its overflowed region that are freed before the corruption happens. As a result, the chunks around the vulnerable object are less flexible and it is difficult for SCATTER to alter the heap layouts to generate ePoCs.
- *Limited explored paths.* For CVE-2017-6965 of `readelf`, although the free chunk that suits the vulnerable object to corrupt a victim object is created, the heap operations on the explored paths are unable to adjust the location of the vulnerable object. However, there are other paths with more heap operations but SCATTER fails to explore them due to complex path constraints.
- *Running Failure.* For CVE-2018-15209 and CVE-2018-16335 of `tiff2pdf`, the running time of their POCs are 102 and 131 seconds respectively and the memory consumed for both exceed 34G. This makes AFL fail to run. For CVE-2019-16347, ASAN reports this vulnerability as a wild pointer overflow. Since our OOB checker only supports to detect overflow raised by continuous writing, SCATTER failed to detect the overflow.

5.2 Comparison with State-of-the-Art (RQ2)

As far as we know, Gollum and Maze are the two state-of-the-art tools for the construction of exploitable layouts for userspace programs. Maze needs to extract and precisely model the capabilities of heap primitives, which is not suitable for general-purpose programs whose primitives are difficult to extract. Gollum is designed for interpreters (i.e., PHP and Python) and do not support general-purpose programs. We replace our distance metric with Gollum's distance (SCATTER$_G$), for the comparison. Besides, AFL supports a *crash explore* mode, which takes crashing test cases as the input, and tries to explore program paths while crashing the program. Given that this crash mode can help discover new PoCs, we use it to generate ePoCs for the comparison (AFL$_{crash}$).

We test the cases in Table 3 with AFL$_{crash}$, AFL$_{crit.}$ (we feed the AFL' mutation engine with the critical input bytes recognized by SCATTER), SCATTER$_G$, and SCATTER*[5]. The experimental configuration is the same with Section 5.1. The final results are shown in Table 4.

On the whole, SCATTER generated the highest number of ePoCs among all the four tools. Compared to the other three tools, the number of ePoCs generated by SCATTER is increased by 133.3%, 38.6%, 31.3% and 6.8% when compared with AFL$_{crash}$, AFL$_{crit.}$, SCATTER$_G$, and SCATTER*

[5] SCATTER with AFL's original code coverage-based scheduling strategy.

Table 3: Results of ePoC Generation

Program	Vulnerability ID	Length of Overflow	# of Overflowed Victims in PoC[2]	# of Victims in PoC[3]	# of Sensitive Struct. /Identified Victims	# of Mutable Cycles/Ops.	# of Unique ePoCs
readelf	CVE-2019-9077	16[1]	0	3	246 / 89	681 / 273	3
	CVE-2017-6965	16	0	1	273 / 88	652 / 264	0
sudo	CVE-2021-3156	unlimited	2	99	165 / 88	496 / 647	4
exiv2	Issue-456	16	0	13	474 / 129	83 / 90	15
	CVE-2018-17230	16	0	13	474 / 129	88 / 86	13
	CVE-2018-11531	151	0	120	486 / 145	82 / 80	14
	CVE-2017-12955	74	0	2	301 / 112	88 / 86	2
	CVE-2017-11339	16	0	2	302 / 112	85 / 84	2
	CVE-2017-1000127	16	0	34	301 / 112	90 / 81	7
	CVE-2018-17229	16	0	5	481 / 115	88 / 86	4
opj_decompress (openjpeg suite)	CVE-2020-6851	16	0	104	72 / 89	88 / 86	3
	CVE-2014-7903	45	0	27	58 / 28	94 / 76	3
opj_compress (openjpeg suite)	CVE-2017-14039	16	1	27	71 / 84	79 / 83	10
	CVE-2017-14164	16	0	25	71 / 84	71 / 37	10
vim	CVE-2021-4019	1023	0	13	285 / 20	397 / 106	0
	Issue-2466	12092	0	15	272 / 27	433 / 138	5
	CVE-2022-0359	800	0	1	355 / 18	413 / 118	0
	CVE-2022-0392	16	0	1	273 / 23	422 / 126	0
gpac	Issue-1703	138	0	52	1243 / 191	253 / 128	2
	Issue-1317	16	0	11	1243 / 234	237 / 116	22
	CVE-2019-20162	16	0	5	1150 / 234	253 / 128	2
	CVE-2022-26967	16	0	57	1470 / 191	261 / 122	5
ffjpeg	CVE-2019-16352	16	0	2	5 / 7	10 / 5	0
tiff2pdf (libtiff suite)	CVE-2018-15209	16	0	3	50 / 18	130 / 290	✗[4]
	CVE-2018-16335	16	0	1	50 / 18	130 / 290	✗
ngiflib	CVE-2019-16346	16	0	1	5 / 12	6 / 4	0
	CVE-2019-16347	48	0	1	5 / 12	6 / 4	✗

[1] For overflow length that is too short like 16, we regard the PoC that places the victim object right after the vulnerable object as an ePoC.
[2] **# of Overflowed Victims in PoC** refers to the number of victim objects that can be corrupted by the overflow in the initial PoC.
[3] **# of Victims in PoC** refers to the number of victim objects in the execution given the initial PoC.
[4] Mark ✗denotes running failure of SCATTER.

Table 4: # of Unique ePoCs when compared with the State-of-the-Art.

Vulnerability	AFL$_{crash}$	AFL$_{crit.}$	SCA.[1]$_G$	SCA.*	SCA.
CVE-2019-9077	2	3	2	3	3
CVE-2017-6965	0	0	0	0	0
CVE-2021-3156	2	3	3	8	4
Issue-456	6	15	15	15	15
CVE-2018-17230	2	13	12	13	13
CVE-2018-11531	4	13	14	14	14
CVE-2017-12955	1	1	1	2	2
CVE-2017-11339	0	1	1	2	2
CVE-2017-1000127	3	3	3	7	7
CVE-2018-17229	3	3	4	4	4
CVE-2020-6851	0	3	3	3	3
CVE-2014-7903	0	3	2	3	3
CVE-2017-14039	8	7	7	10	10
CVE-2017-14164	2	3	3	10	10
CVE-2021-4019	0	0	0	0	0
Issue-2466	5	5	5	5	5
CVE-2022-0359	0	0	0	0	0
CVE-2022-0392	0	0	0	0	0
Issue-1703	1	1	1	2	2
Issue-1317	8	7	11	10	22
CVE-2019-20162	2	2	2	2	2
CVE-2022-26967	5	5	7	5	5
CVE-2019-16352	0	0	0	0	0
CVE-2019-16346	0	0	0	0	0
Total	**54**	**91**	**96**	**118**	**126**

[1] SCA. is short for SCATTER.

respectively. Besides, SCATTER generates more ePoCs than AFL$_{crash}$, AFL$_{crit.}$, SCATTER$_G$, and SCATTER* in 14, 10, 11, and 1 cases respectively.

We can see that without focusing on the critical bytes recognized by SCATTER, AFL$_{crash}$ only generates ePoCs for 15 vulnerabilities. The reason is that AFL leverages the default strategy which focuses on path discovery but not layout exploration. Besides, in the successful cases of AFL$_{crash}$, SCATTER generates more unique ePoCs in 12 cases. We can also see that by focusing on the identified critical input bytes, AFL$_{crit.}$ successfully uncovers ePoCs in 18 cases.

Since the seed scheduling strategy in AFL$_{crit.}$ is based on code coverage, it ignores the seeds that identify new heap operation sequences. In contrast, SCATTER collects more "interesting" seeds in its queues. Figure 8 in Appendix shows the queues' average size (normalized to AFL$_{crash}$) of the evaluated tools after 24 hours' running for the 18 cases that successfully generate ePOCs. The average size of the queues of SCATTER is increased by 7.3x than AFL$_{crit.}$'s. SCATTER$_G$ generates more ePoCs than both AFL$_{crash}$ and AFL$_{crit.}$ (1.78x and 1.05x respectively). However, since the distance of Gollum less accurate, the number of ePoCs found by SCATTER$_G$ shows a decrease of 30 when compared with SCATTER.

5.3 Time Consumption (RQ3)

We evaluated the time consumption in generating ePoCs . The box plots for the evaluated cases are given in Figure 9 in Appendix, which shows the generation time in second of each ePoC. The x-axis is the index of ePoCs. If a tool uncovers less ePoCs than others, we set its generation time for the missed ePoC indexes as the time budget. For example, AFL$_{crash}$ generates only 1 ePoC for CVE-2017-12955 while SCATTER generates 2 ePoCs, we mark the generation time of the second ePoC of AFL$_{crash}$ as 86400 seconds. The average time to generate an ePoC for SCATTER is around 1 hour. The total time consumed to generate all ePoCs for SCATTER is decreased by 59.3%, 41.5%, 32.2%, 21.1%, when compared with AFL$_{crash}$, AFL$_{crit.}$, SCATTER$_G$, and SCATTER* respectively.

CVE-2017-14039, CVE-2017-1000127, CVE-2019-20162, CVE-2018-11531, and Issue-456 demonstrate a more stable performance, as can be seen that the time to generate ePoCs is less accidental. This could be attributed to our scheduling strategy that guides the generation of ePoCs to some extent. As an example, for CVE-2019-20162, AFL$_{crit.}$ and SCATTER both generated 2 ePoC in 10 runs, however from the result shown in the subfigure (at the 5th row and the 2th column) of Figure 9, we can see that the box's height of AFL$_{crit.}$, SCATTER$_G$, and SCATTER*'s ePoC generation time for the first ePoC are 3.0x, 2.7x, 2.1x than that of SCATTER's generation time, indicating that SCATTER's capability in generating ePoCs is relatively stable.

5.4 Case Study

CVE-2017-14164 is a heap overflow vulnerability in `opj_compress` of OpenJPEG 2.2.0. It happens in function `opj_j2k_write_sot` in `lib/openjp2/j2k.c`. When `p_total_data_size` is less than 12, the output buffer is too small to hold `SOT` marker which leads to heap overflow. The vulnerable object's size in the initial PoC is 64 bytes and its overflow length is 16 bytes.

SCATTER identifies 71 sensitive structures and instruments 84 victim objects. For example, SCATTER discovered structure `opj_tcd_image_t` (defined in src/lib/openjp2/tcd.h) as sensitive structure since its first member is a pointer `opj_tcd_tile_t *tiles`, and added instrumentation code to mark the object allocated by `opj_calloc(1, sizeof(opj_tcd_image_t))` (from `src/lib/openjp2/tcd.c:212`) as a victim object. By fuzzing the file input and command line options, SCATTER built the LDG for `opj_compress` which contains 908 nodes and 1397 edges. And identified 71 mutable cycles and 37 mutable operations. Among them, there are 2 specific heap operation sequences are mutable according to the "`-F`" command line option. The corresponding code is shown in Listing 3, which contains 3 mutable operations (`M1`, `M2`, and `F1`). `M1` allocates buffer `substr1` to hold the option argument `opj_optarg`, then `M2` allocates another buffer in heap to store `rawComps`, and `substr1` is freed by `F1` at last.

```
1  // line 696 - line 787
2  case 'F': { // Raw image format parameters
3    ...
4    substr2 = strchr(opj_optarg, '@');
5    if (substr2 == NULL) {
6      len = (OPJ_UINT32) strlen(opj_optarg);
7    }
8    ...
9    substr1=(char*)malloc((len+1)*sizeof(char)); // M1
10   ...
11   if (!wrong) {
12     raw_cp->rawComps = (raw_comp_cparameters_t*) malloc
     (((OPJ_UINT32)(ncomp)) * sizeof(
     raw_comp_cparameters_t));  // M2
13     ...
14   }
15   free(substr1); // F1
16   ...
17 }
```

Listing 3: Code snippet to process `-F` command line option.

```
...
006:0|352|0xfca3a0|4|0|0x0|0x0
007:0|48|0xfca500|5|0|0x0|0x0
008:1|352|0xfca3a0|4|0|0|0                    multiple '-F' options
009:0|160|0xfca530|6|0|0x0|0x0
010:0|48|0xfca5d0|7|0|0x0|0x0
011:1|160|0xfca530|6|0|0|0
...
202:0|1248|0xfcc0e0|136|0|0x0|0x0 ◄── allocate 1248 bytes at 0xfcc0e0
203:0|1248|0xfcc5c0|137|0|0x0|0x0
204:1|1248|0xfcc0e0|136|0|0x0|0x0 ◄── free chunk at 0xfcc0e0
...
210:0|80|0xfcc0e0|142|1|0xfcc130|0xfcc0e0
211:0|64|0xfcc130|143|1|0xfcc170|0xfcc0e0   split from chunk 0xfcc0e0
...                                          split another 80 bytes
215:0|80|0xfcc170|146|1|0xfcc1c0|0xfcc0e0 ◄── from at 0xfcc0e0, start
...                                          address is 0xfcc170
267:1|80|0xfcc170|146|0|0|0 ◄── from chunk 0xfcc170
...
270:1|1312|0xfcc650|140|0|0|0 ◄── from chunk 0xfcc650
...
274:0|1024|0xfcc650|175|1|0xfcca50|0xfcc650
275:0|48|0xfcca50|176|1|0xfcca80|0xfcc650
276:0|112|0xfcca80|177|1|0xfccaf0|0xfcc650   split from chunk 0xfcc650
277:0|48|0xfccaf0|178|1|0xfccb20|0xfcc650
...
288:0|80|0xfccb20|189|1|0x0|0xfcc650 ◄── allocate last reminder
...                                      from chunk 0xfcc650
292:0|64|0xfcc1c0|193|0|0x0|0x0 ◄── allocate the victim object
293:0|880|0xfd0200|194|0|0x0|0x0
294:0|224|0xfd0570|195|0|0x0|0x0
295:0|80|0xfcc170|196|0|0x0|0x0 ◄── allocate the vulnerable object
...
```

Figure 6: Exploitable heap trace record snippet for CVE-2017-14164. Each item is represent as four-tuples $\langle op|size|addr|index\rangle$ where "1" means the deallocate operation while "0" represents the allocate operation. Each chunk contains 16 more bytes to record other information.

By fuzzing this "`-F`" command line option and its arguments, SCATTER successfully placed the vulnerable object adjacent to the lower address of a sensitive object with type `struct.opj_tcd_image`. The heap operation trace snippet is shown in Figure 6. The victim object with type `struct opj_tcd_image` is allocated to 0xfcc1c0 at record item 292, while the vulnerable object is allocated to 0xfcc170 at item 295 which locates right before the victim object. Record items from 006 to 204 are `M1M2F1` sequences that triggered by multiple `-F` options. Record item 202 leverages `M1` to allocate

a chunk at 0xfcc0e0 whose size is 1248, and this chunk is freed by `F1` at record item 204. This chunk is then splitted by item 210 and 211, leaving a 1108-bytes chunk at 0xfcc170. After that, item 215 splits another 80 bytes from the remainder whose start address is 0xfcc170. Then item 267 frees chunk 0xfcc170 which is linked to the corresponding free list. Since chunk 0xfcc170 can overflow to the victim object at 0xfcc1c0, SCATTER selects it as a potential free chunk whose allocation order is $\delta = 1$. Then by counting the number of operations that processes chunk with size 80, we know $n_A = 1$ (item 288) and $n_F = 0$. Therefore, according to (2), $\tau_d = 1$ which means we need one more free operation between item 267 and 295. However, a 1312-bytes free chunk at 0xfcc650 is created by item 270 and then split-ed by a series of allocation which leaves a 80-bytes free chunk. According to the case ② when handling chunk splitting in section 4.3.3, item 277 is treated as an interesting operation and n_F is increased by 1. Then the final distance to overwrite the victim object is updated to $d = 0$, meaning the vulnerable object is placed into chunk 0xfcc170 which constructs an exploitable layout.

6 Limitations

Other General Heap Managers. For prior efforts such as Gollum and Maze, extracting and modelling the capability of heap primitives is the key. Given that SCATTER does not rely on explicitly extracting heap primitives, it is more general and can be applied to general-purpose programs. However, since some programs may use heap managers like tcmalloc [16] and jemalloc [14] in other systems, we need to write the corresponding harnesses to parse the internal state in order to extend SCATTER to support other general heap managers.

Customized Heap Managers. To improve memory usage performance, some programs implement their customized heap managers such as ZendMM [27] in PHP and `store` in EXIM [25]. When allocating, these allocators acquire a large size of block memory and then cut it into different chunks. While deallocating, the chunk is not returned to system but to the block. SCATTER does not model such complicated behaviors when calculating the manipulate distance. In the future, this can be implemented by manually introducing a complete model of those customized heap managers to SCATTER.

Multi-threads. For multi-threaded programs, an input may trigger different executions over different runs. This makes the heap states hard to predict and analyze. In case of race conditions, even though we generate inputs to drive to an exploitable state, we may still fail to verify that in a new run.

7 Related Work

Automatic Exploit Generation for Heap-Based Vulnerabilities. Zhao et al. [35] propose a method called HAEPG, which can generate exploits for vulnerabilities that need multiple exploit steps. Wu et al. [33] implement an AEG system called FUZE for kernel Use-After-Free vulnerabilities by kernel fuzzing and symbolic execution. KOOBE [8] focuses on out-of-bound memory write in Linux kernel, by extracting and utilizing three system calls. Behard et al. [15] build an primitive searching system called PrimGen for browsers which can automatically search for possible exploitable program states. Moritz et al. [12] design a system called HeapHopper based on bounded model checking and symbolic execution to find flaws in heap managers. Insu Yun et al. [34] extend HeapHopper and implement ArcHeap which uncovers more flaws in tcmalloc [16], jemalloc [14] and so on. SHRIKE [20] and Gollum [21] focus on automatic exploitation generation of interpreters. SHRIKE discovers fragments and statements of PHP code as heap primitives to interact with the heap allocator and manipulate the heap layout. Similarly, Gollum leverages statements of interpreters to construct two types primitives. Maze [32] leverages static algorithm to analyze primitive dependency and semantics by recognizing the patterns of code structures, and then uses solving method to achieve exploitable layouts. Unfortunately, those systems all rely on explicit use of heap primitives to achieve AEG.

Automatic Exploit Generation for Other Memory Corruptions. Early AEG solutions are confined to vulnerabilities such as stack overflows and format string vulnerabilities. Brumley et al. [6] propose a patch-based method to generate exploit code. Heelan [19] introduce dynamic analysis to automatically generate exploit code for control flow hijacking . Avgerinos et al. [4] leverage symbolic execution to generate exploit code for control flow hijacking. SK Cha et al. built a fully automated AEG system called Mayhem [7] for binaries and advanced it to Q [29] to support ASLR and DEP bypassing. CRAX [22] is an AEG system built on top of S2E to support more real-world binaries. Kyriakos et al. proposed a method called BOP [23] which can generate data-only exploits to bypass CFI [3]. Nevertheless, none of those work focuses on heap-based vulnerabilities.

8 Conclusion

In this paper, we have present SCATTER, an automated system to generate exploitable heap layouts for heap overflows of general-purpose programs, working in a primitive-free manner by adopting a fuzzing-based approach to search for exploitable heap layouts. By defining a new distance and handling the side effects, it accurately models the allocation and free operations of the heap manager to guide the fuzzing process. The evaluation on 27 vulnerabilities of 10 general-purpose programs demonstrates its effectiveness and efficiency in generating exploitable heap layouts.

Acknowledgment

This work was supported in part by Natural Science Foundation of Hunan Province, China (Grant No. 2022JJ40553) and Research Funding of NUDT (Grant No. ZK22-53).

References

[1] Bugzilla main page. http://bugzilla.maptools.org/. Accessed: 2022-10-10.

[2] Github issues. https://docs.github.com/en/issues. Accessed: 2022-10-10.

[3] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[4] Thanassis Avgerinos, Sang Kil Cha, Brent Lim, Tze Hao, and David Brumley. AEG : Automatic Exploit Generation. 14, 2011. http://security.ece.cmu.edu/aeg/aeg-current.pdf.

[5] Dion Blazakis. Interpreter exploitation: Pointer inference and jit spraying. In *BlackHat DC*, 2020.

[6] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 143–157, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[8] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Conference on Security Symposium*, USA, 2020. USENIX Association.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[10] CVE. Common vulnerabilities and exposures. https://cve.mitre.org/. Accessed: 2022-01-30.

[11] NATIONAL VULNERABILITY DATABASE. Nvd - home. https://nvd.nist.gov/. Accessed: 2022-10-10.

[12] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. SEC'18, page 99–116, USA, 2018. USENIX Association.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[14] FreeBSD. jemalloc. http://jemalloc.net/. Accessed: 2022-01-30.

[15] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. Towards automated generation of exploitation primitives for web browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 300–312, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Sanjay Ghemawat. Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. Accessed: 2022-01-30.

[17] Google. Honggfuzz. https://github.com/google/honggfuzz. Accessed: 2022-01-30.

[18] Google. Project zero: What is a "good" memory corruption vulnerability?, 06 2015. https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html.

[19] Sean Heelan. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.

[20] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. SEC'18, page 763–779, USA, 2018. USENIX Association.

[21] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1689–1706, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 78–87, 2012.

[23] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[24] lcamtuf. american fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2022-01-30.

[25] The University of Cambridge. Exim internet mailer. https://www.exim.org/. Accessed: 2022-01-30.

[26] Sudo organization. Sudo. https://www.sudo.ws/. Accessed: 2022-01-30.

[27] PHP. Zend engine - php internals book. https://www.phpinternalsbook.com/php7/zend_engine.html. Accessed: 2022-01-30.

[28] Qualys. Heap-based buffer overflow in sudo. https://www.qualys.com/2021/01/26/cve-2021-3156/baron-samedit-heap-based-overflow-sudo.txt. Accessed: 2022-01-30.

[29] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 25, USA, 2011. USENIX Association.

[30] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. 26th USENIX Security Symposium (USENIX Security 17), 2017.

[31] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[32] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1647–1664. USENIX Association, August 2021.

[33] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. pages 781–797, 2018.

[34] Insu Yun, Dhaval Kapil, and Taesoo Kim. *Automatic Techniques to Systematically Discover New Heap Exploitation Primitives*. USENIX Association, USA, 2020.

[35] Zixuan Zhao, Yan Wang, and Xiaorui Gong. Haepg: An automatic multi-hop exploitation generation framework. pages 89–109, 07 2020.

Appendices

A Sensitivity to Overflow Length

We conduct an experiment to assess how the overflow length can affect ePoC generation. Since we cannot manually setup the overflow length, we assume the overflow region ranges from 16 bytes to 512 bytes. Again ,we run each vulnerability case for 24 hours with 10 runs and collect: (1) the number of unique ePoCs, (2) the number of all ePoCs, and (3) the time consumption to generate ePoCs , as the outcome to measure the sensitivity to overflow lengths. For a vulnerability case, suppose we conduct N experiments $(e_1, e_2, \cdots, e_N)$ to evaluate how overflow length affects the ePoC generation. And we want to normalize the final result to e_i where $i \in [1, N]$. The three criteria mentioned above are calculated as follows:

- Suppose the number of unique ePoCs for N experiments are $u_1, u_2, \cdots, u_N$, then the normalized number of unique ePoCs are $u_1/u_i, u_2/u_i, \cdots, u_N/u_i$.
- Suppose the number of all ePoCs for N experiments are $a_1, a_2, \cdots, a_N$, then the normalized number of all ePoCs are a_1/a_i, a_2/a_i, $\cdots$, a_N/a_i.
- Suppose the ePoC time for N experiments are $[t_{(1,1)}, \cdots, t_{(1,u_1)}]$, $[t_{(2,1)}, \cdots, t_{(2,u_2)}]$, $\cdots$, $[t_{(N,1)}, \cdots, t_{(N,u_N)}]$, and the minimum number of unique ePoCs among all N experiments is u_{min}, then the normalized ePoC times are $\frac{1}{u_{min}} \sum_{x=1}^{u_{min}} \frac{t_{(1,x)}}{t_{(i,x)}}, \frac{1}{u_{min}} \sum_{x=1}^{u_{min}} \frac{t_{(2,x)}}{t_{(i,x)}}, \cdots, \frac{1}{u_{min}} \sum_{x=1}^{u_{min}} \frac{t_{(N,x)}}{t_{(i,x)}}$.

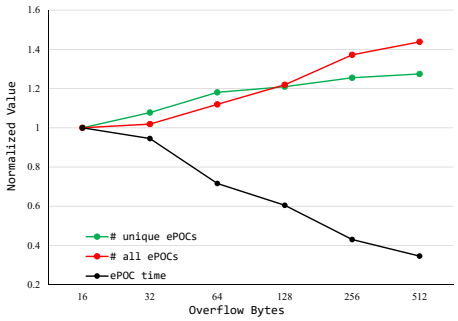

Figure 7: The results of how overflow length affects the number of unique ePoCs, all ePoCs, and time of ePoC generation.

To avoid the noise from the target programs, we merge the results of all vulnerabilities by averaging them. The final results are shown in Figure 7 where we normalized the three criteria to the results of 16 bytes. We can see that the number of unique ePoCs and all ePoCs grows along with the growth of overflow length. But the number of unique ePoCs grows less for longer overflow lengths. This is because even though we have a longer overflow length, but number of all victim objects exhibited in all crashing paths remains unchanged. For example, there are only 5 victim objects can be triggered in explored crashing paths for CVE-2019-20162. Therefore, the number of unique ePoCs has an upper bound for each

vulnerability. However, even though the number of ePoCs grows slowly, the time to generate ePoCs reduces sharply with the growth of overflow length. For example, the normalized time reduces to 0.35 for 512 bytes when compared to 16 bytes. This is because more overflow length brings more potential free chunks to place the vulnerable object. This leads to a higher chance of ePoC generation.

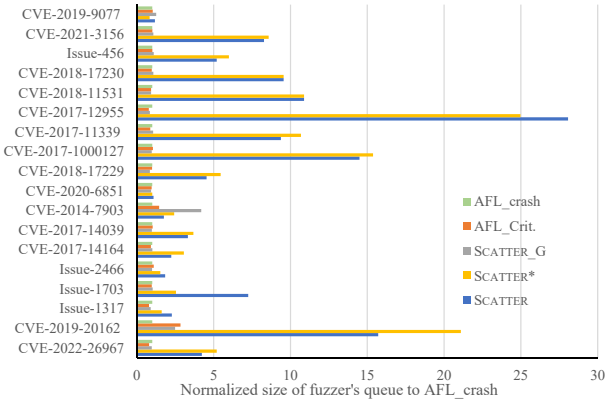

Figure 8: The average number of seeds (normalized to AFL_{crash}) in fuzzer's queue for the successful cases after 24 hours' fuzzing.

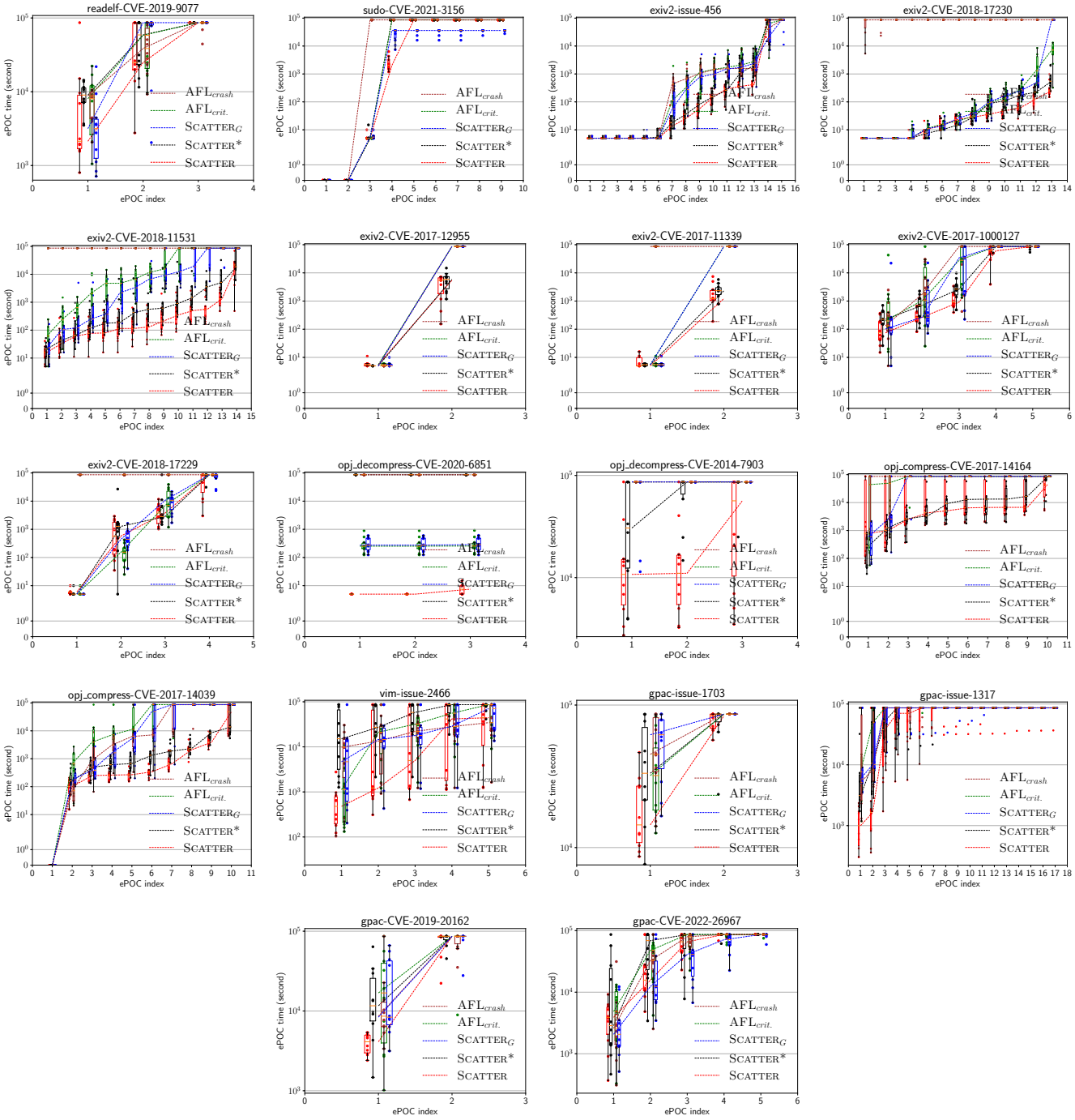

Figure 9: The generation time of each ePoC for AFL$_{crash}$, AFL$_{crit.}$, SCATTER$_G$, SCATTER*, and SCATTER in 24 hours of 10 runs. The x-axis is the index of ePOCS. The y-axis is the generation time in second. The dashed curves plot the median time of generation time for each ePoC in 24 hours of 10 runs.