Towards Automatic and Precise Heap Layout Manipulation for General-Purpose Programs

Runhao Li, Bin Zhang[✉], Jiongyi Chen[✉], Wenfeng Lin, Chao Feng, and Chaojing Tang
National University of Defense Technology
{*lirunhao, b.zhang, chenjiongyi, linwf16, chaofeng, chaojingtang*}*@nudt.edu.cn*

***Abstract*—A critical challenge in automatic heap-based exploit generation is to find out whether an exploitable state can be constructed by manipulating the heap layout. This is usually achieved by re-arranging the objects in heap memory according to an orchestrated strategy that utilizes the program's heap operations. However, hindered by the difficulty in strategically coordinating the use of heap operations given the complexity in the program logic and heap allocation mechanisms, the goal of precise heap layout manipulation for general-purpose programs has not been accomplished.**

In this paper, we present BAGUA, an innovative solution towards automatically and precisely manipulating heap layouts for *general-purpose programs*. Specifically, BAGUA first precisely identifies the primitives of heap layout manipulation using the heap operation dependence graph and thoroughly analyzes their dependencies and capabilities. On this basis, it models the heap layout manipulation as an integer linear programming problem and solves the constraints, in order to identify the sequence of primitives that achieves a desired heap layout. By triggering the primitives in such an order, we are able to construct new proof-of-concept inputs of target programs to achieve an exploitable heap layout. Highlights of our research include a set of new techniques that address the specific challenges of analyzing general-purpose programs, such as eliminating the side effect of heap allocators and extending the capability in manipulating heap layouts. We implemented a prototype of BAGUA and evaluated it on 27 publicly-known bugs in real-world programs. With BAGUA's strength in pinpointing primitives and handling the side effect of heap allocators, it successfully generates desired heap layouts for 23 of the bugs, which is way beyond what prior research can achieve.

I. INTRODUCTION

Nowadays, memory corruption vulnerabilities are still prevailing. Among them, heap-based vulnerabilities are becoming one of the most dominant threats to software and computer systems with their number continuing to expand [6]. With the popularity, generating exploits for a vulnerability is an imperative way to assess the potential damage of the vulnerability [27], [22]. Given the complexity in the workflow of exploitation, heap layout manipulation for heap-based vulnerabilities is fundamental to verifying whether an exploitable state can be constructed [13]. Given a crashing input for the heap-based vulnerability, heap layout manipulation is typically achieved by strategically utilizing the heap operations of the program and precisely placing the critical objects to a specific location in heap memory. For instance, to exploit a use-after-free vulnerability, a target object such as a pointer must be accurately placed to the memory location of a freed vulnerable object. Once the target object is placed on the specific location, hijacking the program control flow becomes feasible.

[✉] Corresponding authors

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23232
www.ndss-symposium.org

To automate heap layout manipulation, a line of research regard it as a search problem and leverage directed greybox fuzzers to gradually shorten the distance between the target object and the smashed area in the heap [28], [29]. The fuzzy approaches are suitable for heap-based buffer overflows that are more tolerable when placing the target objects. Nevertheless, they are not suitable for other types of heap-based vulnerabilities like use-after-free vulnerabilities and double-free vulnerabilities, as exploiting them requires accurate placement of target objects. On the other hand, by treating heap layout manipulation as a problem of digging and filling holes, MAZE [33] models the capability of each heap operation and establishes Diophantine equations [7] to solve the constraints about object placement. It only targets interpreters whose heap operations are highly capable of manipulating the heap layout and can be independently triggered. However, for *general-purpose programs*, the triggering of heap operations is constrained by the program's execution logic and is dependent on each other, which limits the heap operations' capabilities when performing heap layout manipulation. Even worse, complex heap allocation mechanisms would be activated when triggering a sequence of heap operations, introducing side effects during heap layout manipulation.

In this paper, we present BAGUA, a new approach that can automatically and precisely place target objects into target heap locations for *general-purpose programs*. In particular, BAGUA first extracts primitives of heap layout manipulation based on the heap operation dependence graph and analyzes the inter-primitive dependency and the constrained capabilities for manipulation. Based on that, BAGUA then models the heap layout manipulation as an integer linear programming problem. A particular challenge in our research is to handle the side effect of complex heap allocation mechanisms when the primitives of heap layout manipulation are triggered. To this end, BAGUA leverages an innovative algorithm to eliminate the side effects by adding new constraints to the integer linear programming model. In addition, constrained by the program's execution logic, the primitive's capability is often inadequate for the general-purpose programs. Therefore, BAGUA extends the capabilities by leveraging the state-varied feature of heap allocators to fill the target holes. In the end, by solving the constraints of the integer linear programming model, BAGUA outputs a sequence of primitives to construct an exploitable state for the heap-based vulnerability.

With the evaluation on 27 bugs collected from the well-known websites such as EXPLOIT DATABASE [10], HackerOne [12] and CVE MITRE [4], which involves 12 general-purpose programs, BAGUA successfully generates exploitable inputs to trigger corresponding primitive sequences and achieve the desired heap layout on 23 of them. Compared with the state-of-the-art solutions, BAGUA demonstrates its effectiveness in heap layout manipulation for the general-purpose programs with complex structures and execution logic, given its capability in pinpointing primitives of heap layout manipulation and handling the side effect of heap allocators. Besides, we build up an augmented set of primitive sequences to assess in what ways the heap layout manipulation can be affected.

Contributions. The contributions of this work are concluded as follows:

- **New Problem and new techniques.** To the best of our knowledge, this is the first work towards automatic and precise heap layout manipulation for real-world general-purpose programs. Particularly, we leverage the integer linear programming model to describe and solve the constraints and objectives in heap layout manipulation. A set of new techniques are presented such as eliminating the side effects of the complex heap allocators and extending the capability in manipulating the heap layout, to tackle the specific challenges in the adaption of heap layout manipulation to general-purpose programs.
- **Implementation and Evaluation.** We implement this approach and open source it [1] for continuous research. This tool is evaluated with 27 real-world bugs, and the assessment shows that BAGUA outperforms the state-of-the-art solutions by properly handling the side effects for real-world general-purpose programs.

II. BACKGROUND

A. Heap Layout Manipulation

Heap layout manipulation (*HLM*) is to place a target object into a designated location in heap memory, achieving an expected heap layout for further exploit generation. The manipulation of heap layout plays an indispensable role in the exploitation of heap-based vulnerabilities. First, to hijack the instruction pointer with a heap-based buffer overflow, an object that contains a function pointer or a data pointer needs to be placed adjacent to a vulnerable object (`VO`); On the other hand, to exploit a use-after-free vulnerability, the freed chunk should be reoccupied by a target object (`TO`) with suitable size. To fill a target hole, one has to use up all the *free chunks* before the target hole in the chain in specific order.

Free chunks in heap memory are managed by chains ($\mathbb{C}$) in the heap allocator, and each chain usually contains chunks with the same size. Free chunks in one chain are allocated in specific orders such as “First In, Last Out (*FILO*)” or “First In, First Out (*FIFO*)”. We regard the freed chunks as *holes* in memory. As such, HLM is achieved by freeing objects to dig holes or allocating chunks to fill holes [33]. The hole for accommodating a target object is called the *target hole*. Hence, with an expected heap layout, HLM is to put a target object into the target hole. In this paper, the number of the free chunks before filling the target hole is also known as *the number of holes to be filled*.

Heap operations are a particular type of system calls that are used to manipulate the heap memory. This paper focuses on four commonly used heap operations, namely `malloc()`, `free()`, `calloc()`, and `realloc()`. `calloc()` and `realloc()` are essentially different combinations of `malloc()` and `free()`. `malloc()` usually fills a hole in the heap memory and `free()` usually digs a hole. The size of the hole is determined by the allocation size of `malloc()` under specific allocators and operating systems. Regarding the effect on heap memory, the heap operations can be represented by the following two notations (or the combinations of the two notations):

- $\mathcal{A}(x)$: allocating one chunk with size x by bytes. $\mathcal{A}(\forall)$ means the allocation size can be controlled;
- $\mathcal{F}(ptr)$: freeing one chunk pointed by pointer ptr.

Primitives are the code snippets that can be triggered multiple times with a certain input [35], They serve as the basic units to achieve HLM. A primitive $\boldsymbol{p}$ can be described as a set of heap operations, and a target program contains a set of primitives. The primitives are strategically utilized to achieve an expected heap layout and have different capabilities to manipulate the heap layout, which is highly related to the types of programs:

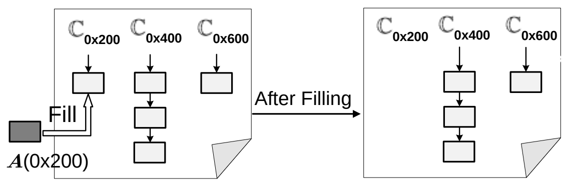
- For interpreters such as Python interpreters and PHP interpreters, a statement that operates the heap could be regarded as a self-contained primitive. Primitives of interpreters seldom depend on each other and are relatively easy to be triggered. For example, the statement in Python `a = bytearray()` can be leveraged to fill a hole of any size. Similarly, in PHP, the statement pair `$var = str_repeat("STR", x)` and `$var = 0` could also dig a hole of any size [21].
- However, a primitive in general-purpose programs often contains multiple heap operations that are dependent on each other. For example, in a network service program, a session close function that frees a chunk relies on the session creation function that allocates a chunk. The use of such a primitive involves sequential triggering of the dependent operations like `malloc()` and `free()`.

B. State-Varied Feature of Heap Allocators

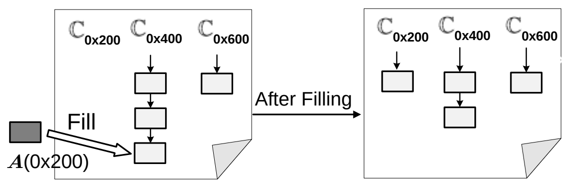
Unlike interpreters where a heap operation can be directly leveraged to dig or fill a hole with the given size, the complex mechanisms of heap allocators about splitting and merging chunks are enabled for general-purpose programs[2]. In other words, with the complex heap allocation mechanism, how a chunk of certain size is freed or allocated is dynamically determined according to the current heap layout. This feature is enabled in mainstream heap allocators such as the split and

[1]https://github.com/Stab1el/BAGUA

[2]Details about the complex heap allocation mechanism is elaborated in Appendix A.



(a) Chunk Allocation without State-Varied Feature



(b) Chunk Allocation with State-Varied Feature

Fig. 1: The effects when allocating a chunk of size 0x200 under two different heap layouts.

merge mechanism under `ptmalloc`. The dynamic behaviors of chunk allocation are designed to improve security and utilization of heap memory. We call it *state-varied feature* in this paper. To illustrate the state-varied feature, we consider a simple heap layout with only three chains ($\mathbb{C}_{0x200}$, $\mathbb{C}_{0x400}$, and $\mathbb{C}_{0x600}$), each of size 0x200 bytes, 0x400 bytes, and 0x600 bytes. $\mathcal{A}$(0x200) represents the heap operation that occupies a chunk with 0x200 bytes. Fig. 1(a) and Fig. 1(b) show the two allocation outcomes under different heap layouts. In Fig. 1(a), $\mathcal{A}$(0x200) simply fills one hole in the chain 0x200. However, in Fig. 1(b), when the heap layout is changed, the allocator consumes a hole in the chain 0x400 and creates a hole in the chain 0x200. If one heap operation allocates or frees a chunk with size x, this operation is a *standard operation* to $\mathbb{C}_x$. Otherwise, the heap operation is a *noise operation*. For example, $\mathcal{A}$(0x200) is a standard operation to the chain $\mathbb{C}_{0x200}$ but a noise operation to the chain $\mathbb{C}_{0x400}$.

The state-varied feature causes side effects to HLM. As a result, the effect of using primitives becomes unexpected if the state-varied feature is not properly handled. From another point of view, such a feature can also be tactically leveraged to achieve a desired heap layout, by extending the primitive capability.

III. Motivation

In this section, we provide the motivation of this research by providing a running example to illustrate the challenges and highlighting our key insights to tackle them.

A. A Running Example

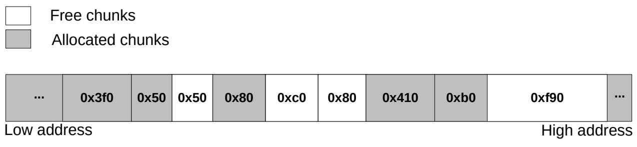
This example is from CVE-2016-10191 [5]. Listing 1 shows a code snippet of the vulnerable program `FFmpeg`. The heap-based buffer overflow occurs due to lack of validation for the `size` variable in the function at line 42. The program allocates a buffer by `ff_rtmp_packet_create()` at line 23 to store the message, and the buffer can be overflowed in `ffurl_read_complete()` (line 44), which is regarded as a vulnerable object `VO`. The target object `TO` is the `RTMPPacket` (line 1 to line 7), which contains one controllable data pointer `*data`, and function `ff_rtmp_check_alloc_array()` (line 16) uses `realloc()` to allocate a chunk to accommodate several target objects.

```
1  typedef struct RTMPPacket {
2      ...
3      uint8_t        *data;           //controllable pointer
4      int            size;
5      int            offset;
6      int            read;
7  } RTMPPacket;  // within size 48 bytes
8
9  void *av_realloc(void *ptr, size_t size){...
10     return realloc(ptr, size + !size);}
11
12 void *av_realloc_array(void *ptr, size_t nmemb, size_t size)
13 {...
14     return av_realloc(ptr, nmemb * size);}
15
16 int ff_rtmp_check_alloc_array(RTMPPacket **prev_pkt, int *
        nb_prev_pkt, int channel){...
17     nb_alloc = channel + 16;
18     // When *prev_pkt is NULL, p1 is triggered; otherwise,
        p2 is triggered
19     ptr = av_realloc_array(*prev_pkt, nb_alloc, sizeof(**
        prev_pkt));
20     ...
21 }
22
23 int ff_rtmp_packet_create(RTMPPacket *pkt, int channel_id,
        RTMPPacketType type, int timestamp, int size){...
24     if (size) {
25         pkt->data = av_realloc(NULL, size);
26         ...}
27 ...}
28
29 // vulnerable function
30 static int rtmp_packet_read_one_chunk(RTMPPacket *p){...
31     // allocate target object here
32      if ((ret = ff_rtmp_check_alloc_array(prev_pkt_ptr,
        nb_prev_pkt, channel_id)) < 0)
33         return ret;
34     ...
35     // allocate vulnerable object here
36     if (!prev_pkt[channel_id].read) {
37         if ((ret= ff_rtmp_packet_create(p, channel_id, type,
         timestamp, size)) < 0)
38             return ret;
39     ...}
40     ...
41     size = size - p->offset;
42     toread = FFMIN(size, chunk_size);
43     // overflow here
44     if (ffurl_read_complete(h, p->data + p->offset, toread)
        != toread){...}
45 ...}
```

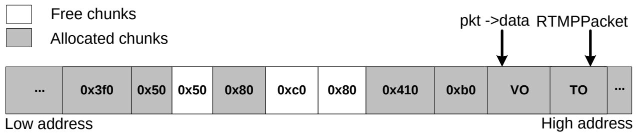
Listing 1: Code Snippet from CVE-2016-10191

The Primitives. This program uses `realloc()` function to allocate `TO`, which calls `malloc()` and `free()` under different conditions. Based on the program's logic, we can identify two primitives: p_1: $\mathcal{A}_1(\forall)$—$\mathcal{A}_2(\forall)$, and p_2: $\mathcal{A}_1(\forall)$—$\mathcal{F}(p_1{:}\mathcal{A}_1)$—$\mathcal{A}_2(\forall)$, where $\mathcal{A}(\forall)$ represents the allocation size is controllable and $\mathcal{F}(p_1{:}\mathcal{A}_1)$ means freeing $\mathcal{A}_1$ of p_1. When the program receives the first incoming packet, it uses p_1 to allocate two chunks. When more packets with different channel IDs are received, p_2 is triggered. $\mathcal{A}_1$ is used to allocate a chunk, whose size is larger than 0x300 bytes, to accommodate `TO`. $\mathcal{A}_2$ is used to allocate a chunk to accommodate `VO`. It should be noted that only after the execution of p_1 could p_2 be executed. Also, under the constraints of the program's logic, p_1 can only be triggered once.

Placing Objects. To exploit this bug, we are supposed to place TO to a higher address position that is adjacent to VO in the heap. To achieve that, appropriate holes should be selected and those objects should be placed into the holes by leveraging primitives. Unfortunately, with the crashing input of this bug, we were unable to find out suitable and adjacent holes in heap memory for accommodating TO and VO, as shown in Fig. 2(a). Also, it is rather difficult to dig proper holes using existing primitives due to their limited capability. Therefore, to achieve the expected layout, users have to place two objects adjacently, select one huge hole, and leverage the state-varied feature to occupy (part of) the hole for accommodating these two objects. Therefore, we leverage p_1 and p_2 sequentially to achieve the expected layout with TO and VO placed in the hole with size 0xf90 bytes, which is shown in Fig. 2.



(a) Heap layout from crash input



(b) Expected heap layout

Fig. 2: The initial heap layout and expected heap layout of CVE-2016-10191. The number in the chunk represents the size of chunk by bytes.

B. Challenges and Key Insights

Although previous approaches [33], [29], [28] could achieve HLM for interpreters, HLM is still challenging for general-purpose programs. Achieving this goal presents the following three particular challenges that previous research do not address:

- **How to precisely identify primitives?** A primitive consists of a set of heap operations that are dependent on each other. In the presence of a large number of heap operations scattered in the complex logic of real-world programs, it is difficult to identify which heap operations "logically" belong to the same primitive. Existing approaches [27], [28] only deal with the primitive that includes several heap operations in interpreters. In addition, those approaches are based on static analysis and cannot handle heap operations in third-party libraries, which limits the total number of primitives identified.
- **How to dig or fill holes with primitives' limited capabilities?** A primitive contains a set heap operations that must be sequentially executed. As a result, for real-world programs, digging or filling holes by triggering a single primitive is not as flexible as triggering a single heap operation. Even worse, determined by the program's execution logic, some primitives are dependent on others. The dependency among heap operations (or among primitives) would significantly restrict the primitives' capabilities on manipulating the heap layout.
- **How to deal with the side effect introduced by the state-varied feature?** As we mentioned before, the behaviors of chunk allocation are dynamically determined by the current heap layout, and modeling such behaviors are complicated. As a result, the triggering of primitives would lead to unexpected outcomes, which makes it difficult to accurate place the object to a suitable location. Prior research do not model the state-varied feature and fail to properly handle the side effect brought by this feature.

Fortunately, in this research, we have observed the following insights to tackle the aforementioned challenges.

- **Leveraging the heap operation dependence graph (*HODG*) to identify primitives.** We leverage a heap operation-guided fuzzer to explore the paths where heap operations are located and combine all those paths as a HODG, whose nodes are heap operations and edges are control flows and data flows. Based on the graph, we identify a loop dispatcher structure in the graph by firstly recognizing an anchor node and entry nodes of primitives, and then pinpointing the primitives on the basis of the entry nodes.
- **Leveraging the state-varied feature to extend capabilities.** With the state-varied feature, the allocation of chunks is dynamically determined by the current structure of the chains. Therefore, we model the behaviors of the state-varied feature and leverage its characteristics to extend the primitives' capabilities. For example, originally $\mathcal{A}$(0x200) in Fig. 1(a) can only occupy a hole with size 0x200. Such a capability can be different (e.g., filling part of a larger hole) when leveraging the state-varied feature in Fig. 1(b).
- **Formulating side effects as constraints.** To fill a target hole, one needs to eliminate noise operations' impacts on the target chain. To this end, we dig a new hole or fill a specific hole to eliminate the side effects and keep the target chain unchanged. This is achieved by modeling the allocation mechanism and adding new constraints to the constraint set of manipulation.

IV. Overview

A. Threat Model

The inputs of BAGUA are a target program, an input that can crash the program, and a specification of the desired heap layout. We focus on the crash caused by heap-based vulnerabilities such as heap-based buffer overflows and use-after-free vulnerabilities. The goal of BAGUA is to achieve an exploitable heap state by manipulating the heap layout, leveraging the attacker-controlled program inputs to steer the heap operations of the program during its execution. In an exploitable heap state, an attacker-controlled object is placed into the location of a freed object in the heap or placed into an overflowed heap area. The output of BAGUA is a primitive sequence that can be triggered by altering the crashing input. On this basis, we manually construct an exploit that can hijack the instruction pointer. Mitigations including ASLR [1] and DEP [8] are enabled[3].

[3]ASLR and DEP only determine the specific address of chunks, which does not change the size of chunks and the structure of the chains. Therefore, enabling them would not affect HLM.

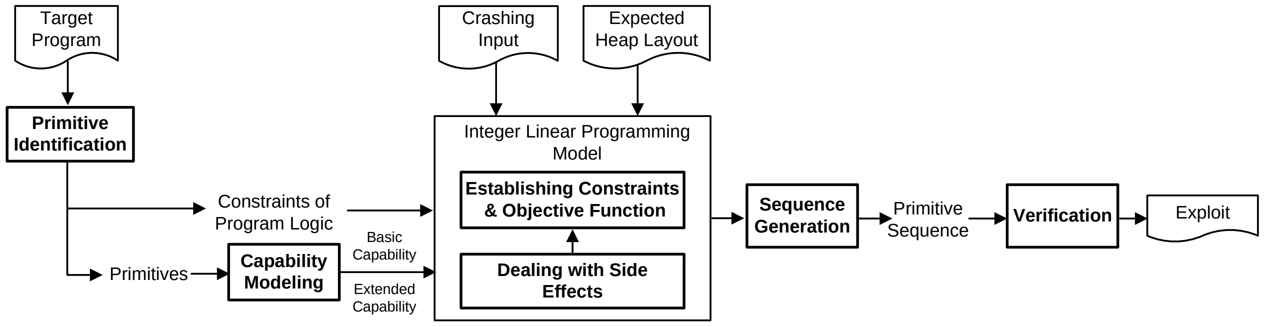


Fig. 3: System overview of BAGUA.

B. Problem Statement

Essentially, the aim of HLM is to fill d holes in the target chain for letting `TO` to occupy the target hole. To this aim, a sequence of primitives are leveraged to achieve filling and (in case of need) digging holes. Generating such a sequence involves two steps: ① which and how many times the primitives should be triggered, and ② how to sort them to follow a specific order.

To determine which and how many times the primitives should be triggered in step ①, constraints from two sources should be satisfied in this research: the constraints on the number of invocations for the primitives and the constraints of the target layout. Assume that the number of invocation for primitive p_i is t_i. t_i must satisfy the constraints of the program's execution logic (i.e., $\mathbb{L}_\mathbb{P}$). The constraints of the target layout is that d holes must be filled to occupy the target hole. Therefore, we can form HLM as a problem of integer linear programming [32] and establish the following formulas based on the constraints:

$$s.t. \begin{cases} \boldsymbol{c} \cdot \boldsymbol{t} + d = 0 \\ \boldsymbol{t} \in \mathbb{L}_\mathbb{P} \end{cases} \tag{1}$$

where $\boldsymbol{t} = (t_1, t_2, \ldots, t_n)^\mathrm{T}$, $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$. n is the number of primitives and c_i is the number of holes that p_i can fill or dig in the target chain. Equation (1) can give a set of solutions. To ease the burden on the procedure of manual exploit generation, we define an objective function ϕ to give priority to the solutions and output an optimal solution (i.e., $min\phi(\boldsymbol{t})$).

To sort the primitives in step ②, we consider the primitive's happen-before relationship and leverage two principles to enhance the steadiness of manipulation. This would produce a set of suitable primitive sequences. The detailed algorithms are described in §V.

V. SYSTEM DESIGN

A high-level workflow of BAGUA is shown in Fig. 3. Primitive identification of §V-A leverages a heap operation-directed fuzzer to identify a variety of heap operations and determines the number of invocations. Then it extracts parameters of the heap allocation operations and determines the freed pointers based on the data flows, and recovers the primitives' happen-before relationship based on the control flows. Capability modeling of §V-B models the primitives' capabilities by the number of digging and filling holes on the target chains. Based on the linear programming model, heap layout manipulation of §V-D collects and solves the constraints from the number of invocation for primitives, the goal of occupying the target hole, and the side effect of the state-varied feature. Notably, to observe the primitive's impact on the heap layout, we implement a heap allocator emulator by integrating mainstream heap allocators. It is able to accurately simulate the behaviors of allocation and freeing of heap chunks, in terms of the property of chains, the side effect of the state-varied feature, and the allocation order that are related to HLM. In the end, primitive sequence generation and verification of §V-D sorts the primitives and generates an exploit to achieve the desired heap layout.

A. Primitive Identification

1) Construction of HODG: To precisely identify primitives, BAGUA leverages the HODG that characterizes the data flows and control flows of the heap operations. A HODG is a graph whose nodes are heap operations and edges are data flows and control flows. BAGUA leverages a heap operation-directed fuzzer to explore paths and heap operations. To obtain diverse primitives for manipulation, it has a particular interest in discovering the sequences of heap operations that are different from the current set. Specifically, BAGUA first hooks the invocation of heap operations by instrumenting the source code. Then, it utilizes a modified fuzzer [2] to explore reachable heap operation invocations, by comparing the identified heap operation sequences and preserving the seed that triggers new heap operation sequences. After that, BAGUA determines which pointer is freed on `free()` using the execution traces and uses symbolic execution to recover the size of allocation on `malloc()`, `calloc()` and `realloc()`. In the end, BAGUA assembles the identified heap operations, the extracted data flows and the control flows to construct the HODG.

Similar to previous research [33], BAGUA is applicable to the programs that are driven by event loops. Considerable programs are driven by user input events or messages, and usually have function dispatchers enclosed in loops to handle these events. For instance, the programs of network interaction are driven by command messages and language interpreters are driven by the statements in scripts.

Triggering some primitives for multiple times can provide more flexibility to the HLM. Therefore, it is necessary to recognize "loop patterns of control flows" in the HODG. To

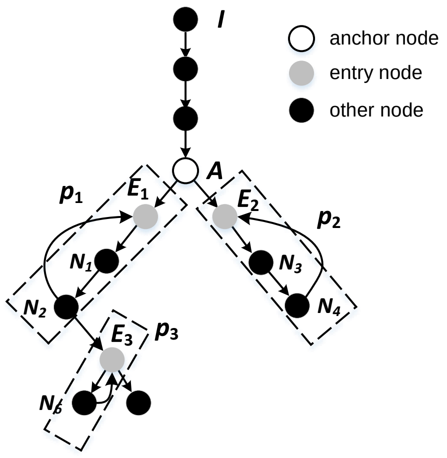


Fig. 4: Primitive identification from the HODG.

this end, we first define the *anchor node*, which is the first dispatcher that has more than one successors and is usually the target program's dispatcher of events. For example, Node A in Fig. 4 is an anchor node. Based on that, primitives reside in the dispatched events. The successors of the anchor node are entry nodes of primitives (e.g., E_1 and E_2). After locating the entry node, a primitive includes the nodes starting from the entry node and ending at the node that returns to another entry node. For example, In Fig. 4, N_2 returns to E_1, which means it is the last node of primitive, and the primitive p_1 includes E_1, N_1 and N_2. Also, there is another type of entry nodes that are not successors of the anchor node, but the successor of the last node of other primitives, for example, E_3 in Fig. 4. In the presence of loop structures, the dominating tree still works. For example, p_3 must be activated after p_1 as there is an edge from p_1 to E_3. For E_2 and E_1 in one loop structure, they cannot be activated in sequentially since there is no edge from p_1 to E_2, or from p_2 to E_1.

2) Constraints of Program's Execution Logic: The constraints from program's execution logic affect the way of triggering primitives. There are mainly two kinds of constraints from the program's execution logic: the happen-before relationship among primitives, and the number of primitive invocations. For instance, a primitive p_1 can only be triggered after the execution of another primitive p_2, and p_2 can only be triggered once during the execution.

Happen-Before Relationship. To identify happen-before relationship, we firstly build a dominating tree for all the primitives' entry nodes in the HODG. After that, we can extract the happen-before relationship among the primitives by simply locating the parent nodes of its entry node in the dominate tree. For example, as shown in Fig. 4, since the entry node E_1 dominates the entry node E_3, the primitive p_3 has a happen-before relationship with p_1.

Number of Primitive Invocations. The number of a primitive's invocation is constrained by the program's execution logic. In each run, we record the number of invocation for each primitive. Based on a large number of executions from the long-time fuzzing campaign, we obtain the maximum and the minimum values for the number of invocations. Despite the fact that the boundaries of the number of invocations based on the fuzzing campaign might be inaccurate, the results are within the theoretical range by the program logic. Therefore, when formulating constraints using the results from the real-world fuzzing, correctness of the solutions from solving constraints can be guaranteed.

B. Capability Modeling

We quantify the primitive's impact on the target chain by modeling its capabilities in manipulating the heap layout. A primitive's capability is the aggregation of its heap operations' capabilities. When the state-varied feature is enabled, the heap operation's capability is regarded as "extended from its basic capability".

1) Heap Operation's Capability: After invoking a heap operation, the increase or decrease on the number of holes to be filled are regarded as the heap operation's capability, in the form of digging or filling holes. If one heap operation digs one hole in the target chain, its capability is $c = +1$. Otherwise, if a heap operation fills one hole in the target chain, its capability is $c = -1$. The modeling of capability is different depending on whether the state-varied feature is enabled or not:

- **Basic Capability.** When the state-varied feature is not enabled, the capability of a heap operation is deterministic. We call it basic capability, represented by c. Specifically, if the target hole locates in the chain $\mathbb{C}_x$, the allocation $\mathcal{A}(x)$'s basic capability is $c = -1$, while the basic capability of other allocations to $\mathbb{C}_x$ is $c = 0$. Similarly, if a free operation $\mathcal{F}(ptr)$ frees the chunk with size x, the basic capability of $\mathcal{F}(ptr)$ is $c = +1$. Generally, a heap operation's basic capability to the target hole with size y could be modeled as follows:

$$c_{\mathcal{A}} = \begin{cases} -1, & x = y \\ 0, & x \neq y \end{cases} \quad c_{\mathcal{F}} = \begin{cases} +1, & x = y \\ 0, & x \neq y \end{cases} \tag{2}$$

- **Extended Capability.** With the state-varied feature enabled, a heap operation's basic capability cannot precisely describe the decrease or the increase on the number of holes to be filled. Therefore, we introduce the extended capability C that models the side effects on the target chain. To dig a hole ($C = +1$) by $\mathcal{A}(x)$ on the target chain $\mathbb{C}_y$, if the number of holes in the chain $\mathbb{C}_x$ is zero (i.e., $L_x = 0$), we leverage the state-varied feature to split another hole with size z in the chain $\mathbb{C}_z$ ($z = x + y$). To fill a hole ($C = -1$) on $\mathbb{C}_y$, we just fill a hole on $\mathbb{C}_y$ when $x = y$, or leverage the state-varied feature to split a hole with size z on $\mathbb{C}_y$ (i.e., $z = y$). The extended capability by the allocation $\mathcal{A}(x)$ on the target chain $\mathbb{C}_y$ is expressed as follows:

$$C_{\mathcal{A}} = \begin{cases} +1, & L_x = 0,\ z = x + y \\ -1, & (L_x = 0,\ z = y) \cup (x = y) \\ 0, & otherwise \end{cases} \tag{3}$$

  Assume $\mathcal{F}(ptr)$ frees a chunk with size x. To fill a hole ($C = -1$) on the $\mathbb{C}_y$ by $\mathcal{F}(ptr)$, we leverage the state-varied feature to merge the just-freed hole by $\mathcal{F}(ptr)$ with the to-be-filled hole on $\mathbb{C}_y$, which forms a larger hole with size z ($z = x + y$). Similarly, $\mathcal{F}(ptr)$ can fill two holes

($C = -2$) on $\mathbb{C}_y$. The just-freed hole by $\mathcal{F}(ptr)$ can be merged with two holes on $\mathbb{C}_y$, when it is adjacent to the two holes and placed in the middle of them in heap memory (i.e., $z = x + 2y$). The merge mechanism of the state-varied feature can also be used to dig a hole with size z, by merging the freed hole by $\mathcal{F}(ptr)$ and another hole. When $z = y$, the hole is digged on $\mathbb{C}_y$ and we have $C = +1$. The extended capability by $\mathcal{F}(ptr)$ is given by:

$$C_{\mathcal{F}} = \begin{cases} +1, & (z = y) \cup (x = y) \\ -1, & z = x + y \\ -2, & z = x + 2y \\ 0, & otherwise \end{cases} \tag{4}$$

2) Primitive's Capability: The basic capability of the primitive p is denoted as c_p. This value is deterministic and is the sum of the heap operations' basic capabilities. The primitive's extended capability C_p is the sum of the heap operations' extended capabilities. It is varied according to different heap layouts. We enumerate all possible values when solving constraints in §V-D.

C. Integer Linear Programming Formulation

The integer linear programming model is used to precisely manipulate the heap layout by establishing constraints and objective functions.

1) Constraints: Achieving the goal of manipulation needs to satisfy two kinds of constraints. One is from the program's execution logic when utilizing primitives. The other is from the demand of occupying the target hole.

Constraints from Program Execution Logic. As mentioned before, the program's execution logic determines the number of primitive's invocations and the happen-before relationship among the primitives. In the integer linear programming model, the quantitative constraints about the number of primitive's invocation is leveraged to formulate the equations. Particularly, For the target program's primitive set $\mathbb{P} = \{p_1, p_2, ..., p_n\}$, $\boldsymbol{t} = (t_1, t_2, \ldots, t_n)^{\mathrm{T}}$ is the vector of the primitive's invocation amount and they satisfy the quantitative constraints $\mathbb{L}_{\mathbb{P}}$ from the boundaries of the invocation amount collected from fuzzing campaigns. For example, $\mathbb{L}_{\mathbb{P}} = \{\boldsymbol{t} | 0 \leq t_1 \leq 1, 0 \leq t_2, \ldots\}$.

Constraints from Target Layout. The goal of occupying the target hole is to decrease the target distance until it becomes 0. We set up equations to describe the goal. For a primitive p_i from the primitive set $\mathbb{P}$, its basic capability is c_i and the invocation amount is t_i. Hence, the primitive p_i could manipulate the layout when the number of holes to be filled is $c_i * t_i$. To achieve the goal, we use primitives with proper number of invocations to gradually decrease the number of holes to be filled from d to 0. Let $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ and $\boldsymbol{t} = (t_1, t_2, \ldots, t_n)^{\mathrm{T}}$. We have the following equation to describe the constraint.

$$\boldsymbol{c} \cdot \boldsymbol{t} + d = 0 \tag{5}$$

2) Objective Function: To facilitate the procedure of exploit generation, we define an objective function to output one proper solution from all the solutions that satisfy the constraints. The intuition is that it tends to be easier to generate exploit when a solution contains less primitives and they are more frequently triggered during the fuzzing campaign. Therefore, the objective function is given as follows:

$$\phi(\boldsymbol{t}) = \sum_{i=1}^{n} \frac{t_i}{\rho_i} \tag{6}$$

where t_i is the number of invocation for primitive p_i in a solution and ρ_i is the maximum number of invocation of p_i during the fuzzing campaign. Based on the constraints and the objective function, the HLM problem can be expressed as finding $\boldsymbol{t}$ according to the integer linear programming model given by:

$$\begin{aligned} &min \quad \phi(\boldsymbol{t}) = \sum_{i=1}^{n} \frac{t_i}{\rho_i} \\ &s.t. \begin{cases} \boldsymbol{c} \cdot \boldsymbol{t} + d = 0 \\ \boldsymbol{t} \in \mathbb{L}_{\mathbb{P}} \end{cases} \end{aligned} \tag{7}$$

When the equations are set up, we use the well-known z3 [31] to solve the constraints. Since the ILP constraints are linear, it is easy for z3 to find a feasible solution as long as the constraints are solvable. To obtain the optimal solution, the solving process is carried out repeatedly until the objective function is minimal or a timeout (set to 600 seconds) is reached.

D. Dealing with Side Effects

We propose two complementary approaches to model and handle side effects. The first approach is based on basic capabilities and eliminates the state-varied feature's side effects, by adding new constraints to the equation set of the basic capabilities. This could introduce considerable constraints and lead to a small solution space. When the first approach gives an unsolvable equation set, the alternative approach is to directly leverage the extended capabilities. While the alternative approach can significantly expand the solution space by setting up new equation sets, it could output incorrect results because the condition about the heap state is not considered when solving constraints. Therefore, a follow-up verification is needed to verify the results.

1) Basic Capability-Based HLM with Side Effect Elimination: BAGUA analyzes the underlying cause of the side effects and turn the side effect elimination into a new problem of digging and filling holes. As an example, if $\mathcal{A}(x)$ occupies holes in $\mathbb{C}_y$ rather than $\mathbb{C}_x$, it indicates that $\mathbb{C}_x$ is empty. To avoid that, it requires digging one hole in $\mathbb{C}_x$. Similarly, if a freed chunk with size x is merged with a hole with size y that is adjacent to it in the heap memory, BAGUA will fill the adjacent hole to avoid the merge behavior of the side effect. This is accomplished by establishing new equations about digging or filling holes.

In the new equations, assume the new target chain is $\mathbb{C}_y$ and the number of holes to be filled or digged is d'. The primitives' basic capabilities and invocation amounts for $\mathbb{C}_y$ are $\boldsymbol{c'} = (c'_1, c'_2, \ldots, c'_n)$ and $\boldsymbol{\Delta t} = (\Delta t_1, \Delta t_2, \ldots, \Delta t_n)^{\mathrm{T}}$, respectively. Equation (8) describes how to eliminate the side effects on the chain $\mathbb{C}_y$ by activating primitives for $\boldsymbol{\Delta t}$ times. These extra primitives should not affect the target chain of $\mathbb{C}_x$, which is represented by (9) where $\boldsymbol{c}$ are the basic capabilities to $\mathbb{C}_x$.

$$\boldsymbol{c}' \cdot \boldsymbol{\Delta t} + d' = 0 \tag{8}$$

$$\boldsymbol{c} \cdot \boldsymbol{\Delta t} = 0 \tag{9}$$

Adding those side effect elimination equations to the original equation set, we have the following where $\widehat{\boldsymbol{t}} = \boldsymbol{t} + \boldsymbol{\Delta t}$:

$$\begin{cases} \boldsymbol{c} \cdot \boldsymbol{t} + d = 0 \\ \widehat{\boldsymbol{t}} \in \mathbb{L}_{\mathbb{P}} \\ \boldsymbol{c}' \cdot \boldsymbol{\Delta t} + d' = 0 \\ \boldsymbol{c} \cdot \boldsymbol{\Delta t} = 0 \end{cases} \tag{10}$$

Unfortunately, a procedure of side effect elimination might bring new side effect. Therefore, the side effect elimination is an iterative process until no more new side effect is produced by the digging or filling operations, or the constraints are unsolvable. To this end, multiple rounds of elimination is needed, which gives:

$$\begin{cases} \boldsymbol{c} \cdot \boldsymbol{t}^{k+1} + d = 0 \\ \boldsymbol{t^{k+1}} \in \mathbb{L}_{\mathbb{P}} \\ \boldsymbol{c}^k \cdot (\boldsymbol{t}^{k+1} - \boldsymbol{t}^k) + d^k = 0 \\ \boldsymbol{c} \cdot (\boldsymbol{t}^{k+1} - \boldsymbol{t}^k) = 0 \end{cases} \tag{11}$$

where:

- k is the round of iteration.
- $\boldsymbol{c} = (c_1, c_2, \dots, c_n)$ represents the primitives' basic capabilities.
- d is the number of holes to be filled for HLM.
- $\boldsymbol{t}^{k+1} = (t_1^{k+1}, t_2^{k+1}, \dots, t_n^{k+1})^{\mathrm{T}}$ represents the numbers of primitive invocations after k rounds of iteration.
- $\boldsymbol{c}^k = (c_1^k, c_2^k, \dots, c_n^k)$ represents the primitives' basic capabilities for a chain in the current k-th round of side effect elimination.
- d^k represents the target distance for a chain in the current k-th round of side effect elimination.
- $\mathbb{L}_{\mathbb{P}} = \{\boldsymbol{t}^{k+1} | t_1^{k+1} \geq 0, \dots\}$ is the constraint set on the number of primitive invocations.

With (11), we can calculate $\boldsymbol{t^{k+1}}$ after k rounds of side effect elimination.

2) Extended Capability-Based HLM: In some cases, basic capability-based HLM with side effect elimination is incapable of finding a solution, due to the following reasons: firstly, adding constraints to the equation set could lead to an unsolvable saturation [4]; Additionally, when a target object is required to be put into a large target hole, the state-varied feature must be leveraged to split the large hole, which cannot be modeled by the basic capability-based HLM.

Extended capabilities provide fine-grained manipulations. Let $\boldsymbol{C} = (C_1, C_2, \dots, C_n)$ represents the primitives' extended capabilities and d represents the number of holes to be filled.

[4]A demonstration is shown in Appendix B.

We can establish the following equations with the extended capabilities:

$$\begin{aligned} &min \quad \phi(\boldsymbol{t}) = \sum_{i=1}^{n} \frac{t_i}{\rho_i} \\ &s.t. \begin{cases} \boldsymbol{C} \cdot \boldsymbol{t} + d = 0 \\ \boldsymbol{t} \in \mathbb{L}_{\mathbb{P}} \end{cases} \end{aligned} \tag{12}$$

Since each C_i contains several values (e.g., the capabilities in (4) and (3)), this could result in multiple combinations of equation sets. For example, assume p_1:$\mathcal{A}$(0x200)—$\mathcal{A}$(0x200) and p_2:$\mathcal{A}$(0x600). The extended capability of p_1 to the target hole with size 0x400 bytes could be -2, -1, 0, $+1$, or $+2$ under different heap states. And that of p_2 could be $+1$ or 0. For each possible value of the primitive's extended capability, we could construct a set of constraints. Therefore, we could construct 10 sets of constraints based on p_1 and p_2. Those constraint sets significantly expand the solution space.

However, the extended capability can only be achieved under specific heap state, and the conditions of using the extended capabilities are not under consideration when solving the equations. As a result, we verify whether the solution of the equations can indeed occupy the target hole using the allocator emulator. When a solution is verified to be infeasible, we iteratively solve the next set of constraints until a feasible solution is found.

E. Placement of Multiple Objects

In some cases, multiple objects need to be placed in specific order to achieve an expected layout. For example, exploiting a heap-based overflow needs to place `VO` and `TO` respectively. This subsection describes how to place multiple objects on the basis of single object placement.

1) Placing Multiple Objects into Multiple Holes: In this case, placing multiple objects can be decomposed into a series of single object manipulation tasks. Note that the latter placement should not affect the earlier placement, meaning that $\mathcal{F}(ptr)$ of the latter placement should not free the occupied target object that is placed earlier.

2) Placing Multiple Objects into One Large Hole: Holes do not always have proper sizes to accommodate the objects with the same exact sizes. Multiple objects may also need to be placed in one large hole. The target objects are required to be placed in a specific order. For example, an expected heap layout of a heap-based buffer overflow requires to place object `TO` firstly and then place `VO` adjacent to `TO` at a lower address. We take the following steps to achieve this goal:

- *Occupying with `AO`.* Selecting a placeholder `AO` and placing it at the lower address of the target hole. The size of `AO` equals to the size of `VO`.
- *Occupying with `TO`.* Placing `TO` to the lower address of the rest of the hole, which is adjacent to `AO`.
- *Freeing `AO`.* Freeing `AO` and keeping `TO`'s occupation unchanged.
- *Occupying with `VO`.* Placing `VO` to the position of the freed `AO`.

Based on our experience, in general, finding an `AO` in general-purpose applications is not hard, as the `AO` could be a structure or data buffer with controllable size. If it cannot be found, we believe the exploitability of the bug is low as well and BAGUA gives up.

F. Primitive Sequence Generation and Verification

After determining which primitives and how many invocations should be leveraged, this section describes how to sort the primitives and to verify the generated sequences.

1) Primitive Sorting: The order of the primitives in the sequence is vital to the success of manipulation. On one hand, the combination of the primitive takes their happen-before relationships into consideration. On the other hand, if a bunch of primitives that fill holes are put to the front of the sequence (i.e., accumulated filling behaviors), there is a chance that the target hole is occupied before allocating the target object. In other words, the target hole is taken over by other objects. Or if the primitives that dig holes are put to the front of the sequence, the expected number of holes to be digged exceed the maximum number of chunks on the target chain. Both situations could fail the manipulation. As an example, for primitive p_1, p_2, p_3 and p_4 within capabilities -3, -4, +2, and -1 respectively, we want to trigger the first three primitives to fill 5 holes before the target hole, and leverage p_4, which allocates the target chunk to occupy the target hole. If the primitive sequence is p_1—p_2—p_3—p_4, the target hole will be occupied by p_2 rather than p_4, which means the target hole is wrongly occupied and the expected heap layout cannot be reached.

To address the aforementioned issues, we use the capability fluctuation rate, indicating the deviations of the primitives' capabilities from its average in the sequence, to measure the quality of sequences and select appropriate ones. Instead of brute-forcing, higher priorities are given to the sequences with less fluctuations. It is less likely to wrongly occupy the target hole and exceed the target chain's capacity, with less accumulated digging or filling behaviors. The detailed algorithm is described in Algorithm 1.

2) Verification: To verify whether the generated primitive sequence could reach an desired heap memory state, we launch new fuzzing campaigns to generate concrete inputs and leverage symbolic execution to facilitate input byte inference when the fuzzer is stuck. The test cases from the fuzzing campaigns in §V-A that trigger desired primitives in the sequence are kept as the initial seeds for the new fuzzing campaigns. However, This input generation still involves some manual efforts. Particularly, once the heap primitives are triggered using directed greybox fuzzing and the symbolic execution engine is used to solve path constraints and infer critical input bytes related to heap operations, still there are some input bytes that could not be inferred using those techniques. As a result, we manually inspect the program logic and debug the program to recover them.

Algorithm 1: Fluctuation Rate-Based Sorting

```
Input : Primitives set $\mathbb{P} = \{p_1, p_2, ..., p_n\}$;
        Capability of each heap operation $c_o$;
        Number of primitive invocation
  $(t_1, t_2, \ldots, t_n)^{\mathrm{T}}$;
Output: Primitive sequence $seq$.
Initialize the list $flist \leftarrow \emptyset$, $dlist \leftarrow \emptyset$, $seq \leftarrow$ None.
for each $p_i$ in $\mathbb{P}$ do
    Initialize the fluctuation rate of $p_i$ is $R_{pi} \leftarrow 0$,
    accumulated fluctuation of operation $S_o \leftarrow 0$,
    for each operation $o_j$ in $p_i$ do
        $S_o \leftarrow S_o + c_{o_j}$;
        $R_{pi} \leftarrow R_{pi} + S_o$;
    end
    if $R_{pi} < 0$ then
        for $j \leftarrow 1$ to $t_i$ do
            append $\{p_i : |R_{pi}|\}$ to $flist$;
        end
    else
        for $j \leftarrow 1$ to $t_i$ do
            append $\{p_i : |R_{pi}|\}$ to $dlist$;
        end
    end
end
sort $dlist$, $flist$ by $|R_{pi}|$ increases;
for $j \leftarrow 1$ to $min\{len(flist), len(dlist)\}$ do
    $seq \leftarrow seq + dlist[j]$.keys();
    $seq \leftarrow seq + flist[j]$.keys();
end
for each $\{p_i : |R_{pi}|\}$ not selected in dlist do
    $seq \leftarrow seq + p_i$;
end
for each $\{p_i : |R_{pi}|\}$ not selected in $flist$ do
    $seq \leftarrow seq + p_i$;
end
return $seq$
```

VI. EVALUATION

A. Experimental Setup

We implement a prototype of BAGUA based on the open-source tools: S2E [26], BooFuzz [15] and afl-fuzz [2], containing over 12,000 lines of code. Target programs are compiled and tested on Ubuntu 20.04 with a mainstream heap allocator ptmalloc [18]. The CPU of the machine is Intel i7-10875 2.3GHz with 64G RAM, utilizing 16 logical processors. We evaluate BAGUA on two benchmarks. On one hand, we collect 27 bugs from 12 real-world programs [9], [11], [4], [16], [17]. All CVEs and bugs are selected from online websites, including CVE MITRE [4], EXPLOIT DATABASE [10] and HackerOne [12]. We select the bugs with the following criteria:

- The bugs are heap-based, such as heap-based buffer overflows, use-after-free vulnerabilities and double free vulnerabilities.
- All the bugs are exploitable and could be reproduced. We search for the bugs whose exploit code is publicly known. If it is not released, we select the bugs that are proved to be exploitable by someone (we know that from online blogs' descriptions).
- With some preliminary manual analysis, we select the target programs whose inputs are relatively easy to be mapped to heap operations, such as event-loop driven programs, and command line parsers.

On the other hand, we build a test set to assess how HLM can be affected by some factors like the primitive length,

TABLE I: Results of Primitive Identification

Programs	Versions	# of Primitive[1]	Primitive Length[2]	Accuracy
FFmpeg	3.2.1	15 | 11	2 - 7	100%
Exim	4.89	21 | 18	1 - 12	95.2%
GoAhead	3.1.3	9 | 9	2 - 12	88.9%
DNSmasq	2.75	8 | 6	1 - 9	100%
ELOG	3.1.4	8 | 7	3 - 12	100%
atftp	0.7.4	10 | 10	3 - 7	90%
ProFTPD	2.78	9 | 8	2 - 14	88.9%
Netgear R7000	1.0.11.116	7 | 6	2 - 7	100%
sudo	1.8.31	7 | 6	4 - 9	100%
curl	7.54.1	11 | 9	1 - 10	90.1%
Python	2.7	2 | 2	1 - 2	100%
PHP	5.3	2 | 2	1 - 2	100%

[1] The number on the left-hand side is from the heap operation-guided fuzzer. The right-hand side is from the afl-fuzz or BooFuzz.
[2] The minimum primitive length and the maximum primitive length in extracted primitives.

TABLE II: Statistics of the Number of Primitive Invocation

Primitive	Program	Boundary from Test*	Ground Truth
$\mathcal{A}_1(\forall)$—$\mathcal{A}_2$(0x30)—$\mathcal{F}(p_1{:}\mathcal{A}_2)$—$\mathcal{F}(p_1{:}\mathcal{A}_1)$	ELOG	$0 \le t \le 1$	$0 \le t \le 1$
$\mathcal{F}(p_5{:}\mathcal{A}_1)$—$\mathcal{A}_1(\forall)$	Exim	$0 \le t \le 12$	$0 \le t$
$\mathcal{A}_1$(0x20)—$\mathcal{A}_2$(0x8010)—$\mathcal{A}_3$(0xf8)—$\mathcal{A}_4$(0x3c)—$\mathcal{A}_5(\forall)$—$\mathcal{A}_6$(0x830)	FFmpeg	$0 \le t \le 1$	$0 \le t \le 1$
$\mathcal{F}(p_1{:}\mathcal{A}_2)$—$\mathcal{A}_1(\forall)$—$\mathcal{F}(p_2{:}\mathcal{A}_1)$	Exim	$0 \le t \le 7$	$0 \le t$
$\mathcal{F}(p_1{:}\mathcal{A}_1)$—$\mathcal{A}_1(\forall)$	atftp	$0 \le t \le 1$	$0 \le t \le 1$
$\mathcal{A}_1(\forall)$—$\mathcal{F}(p_1{:}\mathcal{A}_1)$—$\mathcal{A}_2(\forall)$	FFmpeg	$0 \le t \le 1$	$0 \le t \le 1$
$\mathcal{A}_1(\forall)$—$\mathcal{A}_2(\forall)$—$\mathcal{F}(p_3{:}\mathcal{A}_1)$—$\mathcal{F}(p_3{:}\mathcal{A}_2)$	DNSmasq	$0 \le t \le 1$	$0 \le t \le 1$
$\mathcal{A}_1$(0x20)—$\mathcal{A}_2$(0x110)	GoAhead	$0 \le t \le 1$	$0 \le t \le 1$

* t refers to the number of primitive invocation.

the number of provided primitives, the rate of allocation with arbitrary size, and the rate of noise operations.

B. Benchmark I: Real-world CVEs and Bugs

1) Results of Primitive Identification: The fuzzer is run for 24 hours to explore the program, in order to construct the HODG and identify the primitives. The statistics of the target programs, the number and the length of identified primitives, and the accuracy of primitive identification are given in Table I. For the Python interpreter and the PHP interpreter, there are two statements of heap manipulation are directly used as primitives. Regarding the number of identified primitives, we can see that there are some improvement made by the heap operation-guided fuzzer, when compared with the original fuzzers. On the fourth column, the length of primitive is diverse except for the two interpreters. This is attributed to the complexity of the target programs and indicates the difficulty of HLM. The accuracy of primitive identification is relatively high (96.1% on average). To obtain the accuracy, we manually debug the program using the GDB toolkit [19] to verify the extracted primitives. The inaccuracy mainly comes from the misidentification of heap operation's argument.

To measure the correctness of invocation times for the identified primitives, we randomly select 8 primitives among all the identified primitives of the target programs and compare their invocation times with the ground truth. The ground truth is manually analyzed by code comprehension and facilitated with debugging using some scripts. The results are shown in Table II. As can be seen, the results are correct in 6 out of 8 cases. Although there are two cases that have no upper bound, the number of invocation triggered during the fuzzing campaign is still within theoretical range. This ensures that the constraints provided to the equation set are accurate.

2) Results of Heap Layout Manipulation: Table III shows the results of HLM in terms of solving constraints and sorting primitives. As can be seen, BAGUA can achieve desired heap layouts for 23 out of 27 cases. Among the successful cases, except for the 11 bugs of the Python interpreter and the PHP interpreter, there are 11 cases that require to address side effects caused by the state-varied feature during manipulation. In particular, for the three bugs in DNSmasq and ELOG, BAGUA achieves target occupation using basic capabilities with side effect elimination. For the rest where side effects occur, such as CVE-2016-10191 and CVE-2018-0500, there are no suitable holes in the heap layout to accommodate the target hole. As such, BAGUA achieves occupation for them using extended capabilities. The fourth column of Table III shows that 40.7% of the cases are required to occupy multiple objects in specific order. For example, to achieve an unlink heap layout in CVE-2018-6789, `TO` and `VO` must be placed with specific order and positions [20]. The fifth column shows the number of unique primitives for manipulation by solving the constraints. On average, only 32.6% of the identified primitives are utilized, indicating that the fuzzer's searching efforts for the primitives are sufficient. The column of program constraint shows that the constraints from the program's execution logic exist in 12 out of 23 cases. The eighth column gives the lengths of the primitive sequences. On average, a primitive sequence contains 3.3 primitives, which is not a large number after the selection of the objective functions in §V-D.

BAGUA fails to achieve expected manipulation in 4 cases. For CVE-2020-9273, we find that ProFTPD manages heap memory with customized allocator, whose allocation and free operations are not performed by system calls. In this case, BAGUA is unable to model the capability of its heap operations. For CVE-2021-31802, the program is extracted from firmware, where heap allocator is slightly different from that of glibc in Ubuntu, and it leads to inaccurate modeling of the primitives' capabilities. For CVE-2017-16943 and CVE-2019-5096, we find out that BAGUA fails to solve the collected constraints until reaching a timeout. After manual inspection, we find that BAGUA misses some primitives that are capable of achieving the desired heap layout.

3) Comparison with State-of-the-Art: We compare BAGUA with the automated HLM tools SHRIKE [28], GOLLUM [29]

TABLE III: Overall Results

Bug ID (Program)	Bug Type	Handling	# of	# of Uni.	Logic Constr.[3]		Seq.	Constr. Solving		Sorting		B.[6]	S.|G.|M.
		Side Effect [1]	TO	Prim.[2]	Num.	H.-B.	Len.[4]	Time	PMU[5]	Time	PMU		
CVE-2016-10190 (FFmpeg)	heap overflow	③	1	3	✓	✓	3	47	18%	11|23	10%	✓	✗| ✗| ✗
CVE-2016-10191 (FFmpeg)	heap overflow	③	1	2	✓	✓	2	54	21%	5|7	4%	✓	✗| ✗| ✗
CVE-2018-6789 (Exim)	off-by-one	③	2	4	✓	✓	4	89	27%	12|29	13%	✓	✗| ✗| ✗
CVE-2019-16928 (Exim)	heap overflow	③	2	1	✓	✗	1	51	20%	2|1	3%	✓	✗| ✗| ✗
CVE-2020-28020 (Exim)	heap overflow	③	2	5	✓	✓	10	163	37%	34|109	19%	✓	✗| ✗| ✗
CVE-2014-9707 (GoAhead)	heap overflow	③	2	3	✓	✓	3	71	23%	5|18	8%	✓	✗| ✗| ✗
CVE-2017-14491 (DNSmasq)	heap overflow	②	1	3	✓	✓	3	78	26%	6|17	8%	✓	✗| ✗| ✗
CVE-2009-2957 (DNSmasq)	heap overflow	②	1	2	✓	✓	2	46	21%	4|5	5%	✓	✗| ✗| ✗
CVE-2019-3994 (ELOG)	double free	②	1	2	✗	✓	2	62	25%	6|7	8%	✓	✗| ✗| ✗
CVE-2021-41054 (atftp)	heap overflow	③	1	2	✓	✓	2	53	21%	5|6	9%	✓	✗| ✗| ✗
CVE-2021-3156 (sudo)	heap overflow	①	1	2	✓	✓	2	49	20%	5|5	12%	✓	✗| ✗| ✗
CVE-2018-0500 (curl)	heap overflow	③	1	5	✓	✓	5	82	29%	21|42	15%	✓	✗| ✗| ✗
CVE-2014-1912 (Python)	heap overflow	①	2	2	✗	✗	3	12	15%	5|11	5%	✓	✗| ✓| ✓
Issue 24094 (Python)	use-after-free	①	1	2	✗	✗	2	21	16%	4|6	7%	✓	✗| ✗| ✓
Issue 24095 (Python)	use-after-free	①	1	2	✗	✗	2	32	18%	5|4	5%	✓	✗| ✗| ✓
issue 24105 (Python)	use-after-free	①	1	2	✗	✗	2	26	17%	5|5	6%	✓	✗| ✗| ✓
issue 24481 (Python)	heap overflow	①	2	3	✗	✗	4	18	15%	12|33	9%	✓	✗| ✓| ✓
CVE-2016-5636 (Python)	heap overflow	①	2	2	✗	✗	4	19	16%	14|27	10%	✓	✗| ✓| ✓
issue 27211 (Python)	heap overflow	①	2	2	✗	✗	5	43	19%	21|63	15%	✓	✗| ✓| ✓
CVE-2013-4113 (PHP)	heap overflow	①	2	2	✗	✗	4	29	17%	14|25	9%	✓	✓| ✓| ✓
CVE-2013-2110 (PHP)	heap overflow	①	2	2	✗	✗	2	16	14%	4|4	4%	✓	✓| ✓| ✓
bug 76047 (PHP)	use-after-free	①	1	2	✗	✗	4	31	17%	18|32	14%	✓	✗| ✗| ✓
bug 72697 (PHP)	heap overflow	①	2	2	✗	✗	5	37	20%	23|46	15%	✓	✓| ✓| ✓
CVE-2020-9273 (ProFTPD)	use-after-free	-	-	-	-	-	-	-	-	-	-	✗	✗| ✗| ✗
CVE-2017-16943 (Exim)	use-after-free	-	-	-	-	-	-	-	-	-	-	✗	✗| ✗| ✗
CVE-2021-31802 (httpd)	heap overflow	-	-	-	-	-	-	-	-	-	-	✗	✗| ✗| ✗
CVE-2019-5096 (GoAhead)	double free	-	-	-	-	-	-	-	-	-	-	✗	✗| ✗| ✗

[1] "①" means BAGUA uses basic capabilities without any side effects, and "②" means BAGUA uses basic capabilities with side effect elimination to achieve occupation. "③" means BAGUA uses extented capabilities to achieve occupation, and "-" means failure for manipulation.
[2] "Uni. Prim." refers to unique primitives solved by constraints for manipulation.
[3] "Logic Constr." means the constraints of program logic. "Num." represents the constraint of the number of primitive invocation and "H.-B." represents the happen-before relationship.
[4] "Seq. Len." means the length of primitives sequence generated by BAGUA.
[5] "PMU" refers to peak memory usage, which is obtained by sampling system memory usage every second.
[6] "B." represents BAGUA. "S." represents SHRIKE. "G." represents GOLLUM, and "M." represents MAZE.

and MAZE [33]. SHRIKE and GOLLUM focus on interpreters and do not support other applications. Therefore, in the evaluated cases, they only work on heap-based buffer overflows such as CVE-2013-4113 and CVE-2013-2110 of the interpreters. Since these two tools are designed based on search-based methods, they might face low efficiency when dealing with precise manipulation. An example is the issue 24015 that involves precisely reoccupying the freed hole using the use-after-free vulnerability. MAZE sets up Diophantine equations to precisely place the target object, which works well for the bugs of the interpreters. We contacted the authors of MAZE and obtained the code. It turns out that the primitive capabilities of MAZE are fixed without the state-varied feature. All in all, BAGUA outperforms the three tools on 12 cases, involving 8 different general-purpose programs. This is mainly attributed to the fact that BAGUA is able to leverage the limited capabilities in the programs and deal with the side effects of the heap allocator.

4) Cost Evaluation: To evaluate the cost of BAGUA during manipulation, we measure the running time and the peak memory usage (PMU) during the constraint solving and the primitive sorting. We reproduce each step for 10 times and take their averages. As can be seen from Table III, once the primitives are ready, it does not take a long time (averagely 57.7 seconds for constraint solving constraint and 10.1 seconds for primitive sorting) to accomplish HLM. Also, the computing resources required by HLM are acceptable, given that the average PMU is only 20.5% on our machine. Regarding the approaches to model and eliminate the side effects, the extended capability-based approach takes averagely 76.25 seconds and the basic capability-based approach takes 62 seconds to solve constraints. This is mainly because the former approach takes more equation sets as its inputs. We also compared the fluctuation-based sorting strategy with a random combination strategy, in terms of time consumption. As shown in the 11th column of Table III, the proposed sorting strategy reduces the time of sorting with random combination strategy from 22.8 seconds to 10.5 seconds. As the length of the primitive sequence increases, the fluctuation-based sorting can save more time.

C. Benchmark II: Augmented Test Set of Primitive Sequence

We construct an augmented test set of primitive sequence to evaluate how the success rate of BAGUA can be affected. Four important factors are measured, including the primitive length (i.e., the number of heap operations in a primitive), the number of provided primitives, the rate of allocation with arbitrary size, and the rate of noise operations. We set up an initial heap layout, take constructed primitive sequences as the inputs, and verify whether the desired layout could be achieved. Before the experiments, we have manually confirmed that all the generated primitives are valid and can be leveraged to achieve the desired heap layout. The primitive sequences are constructed by varying the to-be-evaluated factors. For example, to evaluate the impact of the primitive length, we generate 6 primitive sets whose primitives contain 2, 4, 6, 8, 10 and 12 heap operations respectively, and use those primitives for HLM. The experiments are carried out under `ptmalloc 2.31` and `ptmalloc 2.24`.

As shown in Fig. 5 and Fig. 6, the success rate of HLM is barely affected by the primitive length and the number of provided primitives. It turns out that longer primitives or more provided primitives increase the time consumption, but do not improve the success rate of BAGUA. This also indicates that although discovering more primitives from the program is necessary, generally speaking, the success rate is barely affected as long as certain amount of primitives are provided (e.g., 4 primitives).

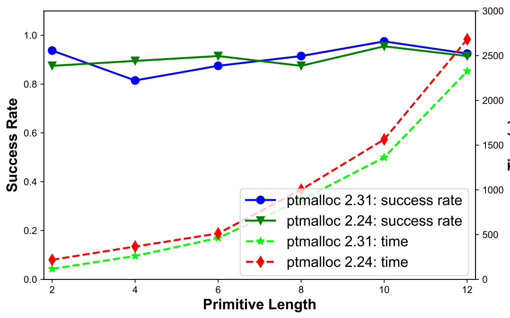


Fig. 5: Influence on manipulation by primitive length.

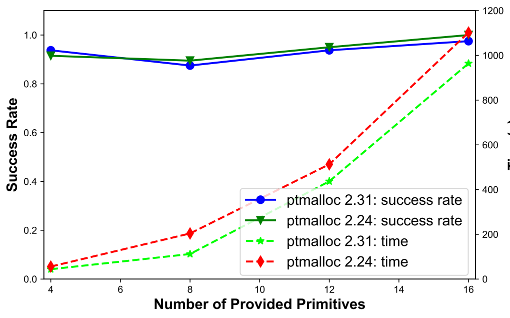


Fig. 6: Influence on manipulation by number of provided primitives.

The rate of controllable allocation is the ratio of the number of $\mathcal{A}(\forall)$ to the total number of allocation operations. Fig. 7 illustrates the influence on HLM by such a rate. As the rate of controllable allocation increases, the success rate of HLM and the time consumption increase as well. The rate of noise operation is the ratio of the number of noise operation to the number of all heap operations. Fig. 8 shows the impact of such a rate on HLM. The success rate decreases as noise operation increases. The main reason is that more noise operations increase the degree of side effects, which adds the complexity of the constraints. Some of the constraints become unsolvable within a timeout. Despite that, BAGUA still achieves desired heap layouts for more than 50% test cases, even with 80% of noise operations.

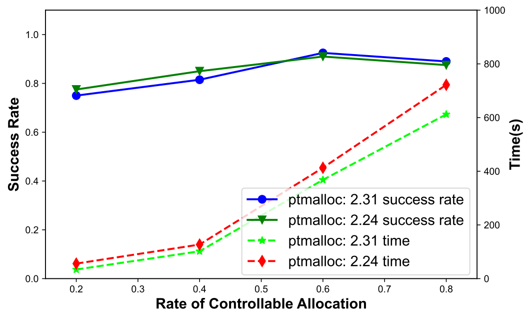


Fig. 7: Influence on manipulation by rate of controllable allocation.

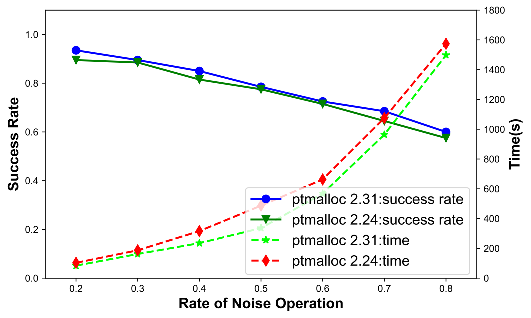


Fig. 8: Influence on manipulation by rate of noise operation.

VII. DISCUSSION

Code Coverage. As BAGUA leverages fuzzing to trigger heap operations and identify primitives, its capability of discovering primitives relies on the fuzzer's ability to explore the code. Although we design a heap operation-guided fuzzer to improve that, as the evaluation shows, the HLM could still fail due to lack of certain primitives for two cases. To address this problem, an alternative approach is to leverage the results of static analysis to provide more fine-grained guidance to the fuzzers. If more accurate identification of primitives and their dependency can be given for the complex real-world general-purpose programs, the performance of BAGUA can be further enhanced.

Scalability. BAGUA is implemented with ptmalloc, a system allocator widely used in UNIX. For the system allocators with deterministic behaviors (e.g., ptmalloc, dlmalloc [3] and jemalloc [14]), BAGUA is applicable as long as the calculation of TO-VO distance and the primitives' capabilities are adjusted accordingly. However, for the system allocators with nondeterministic behaviors like mimalloc (with random allocation), BAGUA is not applicable. On the other hand, for the program's customized heap allocators, the challenges lie in the program's diverse APIs and complicated mechanisms of heap management. For example, the mechanism of "delay-free" in Internet Explorer's allocator prevents the reuse of a just freed area, which makes distance calculation difficult to model. Extending the support for more types of allocators is left as the future work.

VIII. Related Work

A. Automated Exploit Generation for Stacked-based Vulnerabilities

Recent years have seen an increasing interest in automated exploit generation (AEG) techniques. In earlier years, a line of research focus on automated exploit generation for stack-based buffer overflows. Those AEG frameworks focus on exploiting stack-based buffer overflows and format string vulnerabilities based on classic exploitation methods. Heelan et al. [27] proposed an approach to generate control-flow-hijacking exploits based on a crashing input, which leverages dynamic taint analysis techniques. Mayhem [23] leverages symbolic execution to exploit the stack-based buffer overflows, which is implemented as an end-to-end exploitation system. CRAX [30] performs symbolic execution in concolic mode, using a whole system environment model, to generate exploits that hijack the control flow with stack-based vulnerabilities.

B. Automated Exploit Generation for Heap-based Vulnerabilities

There is a growing interest in AEG for heap-based vulnerabilities in recent years. Compared with AEG for stack-based vulnerability, exploiting heap-based vulnerabilities is more challenging, due to the requirement of specific heap layouts. As such, primitive analysis and automatic heap layout manipulation are two critical steps for AEG of heap-based vulnerabilities.

Primitives Analysis. As for primitives identification, existing works leverage static approaches and dynamic approaches to extract primitives in different types of programs. Sean Heelen et. proposed SHRIKE [28] and GOLLUM [29], which 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. SLAKE [25] tackles automatic manipulation problem towards vulnerabilities in kernel space, utilizing systems calls for exploitation. KOOBE [24] focuses on out-of-bound memory write in Linux kernel, by extracting and utilizing three system calls. FUZE [34] leverages kernel fuzzing to identify useful system calls for the exploitation of use-after-free vulnerabilities. ARCHEAP [35]analyzes modern heap allocators and leverages fuzzers to systematically identify heap exploitation primitives against various allocators. HAEPG [36] statically identifies the paths that make up dispatchers and dynamically executes those paths to collect heap interactions as heap primitives. MAZE [33] regards building blocks that could interact with target programs, which could be used by users to achieve heap layout manipulation. MAZE leverages static algorithm to analyze primitive dependency and semantics by recognizing the patterns of code structures. Unlike those works, our approach targets general-purpose programs and relies on a heap operation-guided fuzzer to explore program paths, identify heap operations that can be triggered with concrete inputs, and identify primitives based on the exploration results.

Automatic Heap Layout Manipulation. Existing works towards automatic heap layout manipulation for real-world targets mainly consider two kinds of programs, namely interpreters and the Linux Kernel, due to their simplicity. SHRIKE leverages random search algorithm to assemble primitives to automatically manipulate the heap layout. GOLLUM improves this method by importing genetic algorithm to increase manipulation efficiency for interpreters. Both of them aim for heap-based overflow and are restricted in some situations such as precisely reoccupying a freed object to exploit a use-after-free vulnerability. MAZE leverages the linear Diophantine equation to address the manipulation problem. It mainly works for interpreters and CTF challenges without such complicated primitives. SLAKE establishes several manipulations methods based on typical characteristics of Linux kernel vulnerabilities. KOOBE and FUZE both leverage heap spray techniques to consume holes and manipulate the target object. Different from those works, BAGUA leverages the integer linear programming model and focuses on precise and automatic HLM in the presence of side effects for real-world general-purpose programs.

IX. Conclusion

In this paper, we have presented a new approach towards automatically and precisely manipulating heap layouts for general-purpose programs, which is a key step of automated exploit generation for heap-based vulnerabilities. We have presented new techniques by solving the challenges that are specific to general-purpose programs such as identifying primitives on the basis of program's logic, eliminating side effects of the state-varied feature of heap allocators, and extending the limited capabilities. The evaluation shows that this approach outperforms state-of-the-art in terms of the applicability and the effectiveness on 27 bugs in general-purpose programs.

Acknowledgment

We want to thank our anonymous reviewers for their valuable comments. This work was supported in part by Natural Science Foundation of Hunan Province, China (Grant No. 2022JJ40553) and Research Funding of National University of Defense Technology (Grant No. ZK22-53).

References

[1] "Address space layout randomization," https://en.wikipedia.org/wiki/Address_space_layout_randomization, Accessed: September 2022.

[2] "american fuzzy lop," https://lcamtuf.coredump.cx/afl/, Accessed: September 2022.

[3] "C dynamic memory allocation," https://en.wikipedia.org/wiki/C_dynamic_memory_allocation#dlmalloc, Accessed: September 2022.

[4] "CVE," https://cve.mitre.org/, Accessed: September 2022.

[5] "CVE-2016-10191," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10191, Accessed: September 2022.

[6] "CVE Details," https://www.cvedetails.com/, Accessed: September 2022.

[7] "Diophantine equation," https://en.wikipedia.org/wiki/Diophantine_equation, Accessed: September 2022.

[8] "Executable space protection," https://en.wikipedia.org/wiki/Executable_space_protection#Windows, Accessed: September 2022.

[9] "Exim internet mailer," https://www.exim.org/, Accessed: September 2022.

[10] "EXPLOIT DATABASE," https://www.exploit-db.com/, Accessed: September 2022.

[11] "FFmpeg," https://ffmpeg.org/, Accessed: September 2022.

[12] "HackerOne," https://www.hackerone.com/, Accessed: September 2022.

[13] "Heap feng shui," https://en.wikipedia.org/wiki/Heap_feng_shui, Accessed: September 2022.

[14] "jemalloc memory allocator," https://jemalloc.net/, Accessed: September 2022.

[15] "Network Protocol Fuzzing for Humans," https://boofuzz.readthedocs.io/en/stable/, Accessed: September 2022.

[16] "Php bug tracking system," https://bugs.php.net/, Accessed: September 2022.

[17] "Python bug tracker," https://bugs.python.org/, Accessed: September 2022.

[18] "The gnu c library (glibc)," https://www.gnu.org/software/libc/, Accessed: September 2022.

[19] "The GNU Project Debugger," https://sourceware.org/gdb/, Accessed: September 2022.

[20] "Unlink exploit," https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html/, Accessed: September 2022.

[21] D. Blazakis, "Interpreter exploitation: Pointer inference and jit spraying," *BlackHat DC*, 2010.

[22] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 143–157.

[23] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[24] W. Chen, X. Zou, G. Li, and Z. Qian, "{KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1093–1110.

[25] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1707–1722.

[26] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[27] S. Heelan, "Automatic generation of control flow hijacking exploits for software vulnerabilities," Ph.D. dissertation, University of Oxford, 2009.

[28] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 763–779.

[29] ——, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1689–1706.

[30] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. 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*. IEEE, 2012, pp. 78–87.

[31] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[32] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[33] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "Maze: Towards automated heap feng shui," in *30th USENIX Security Symposium (USENIX Security 21). USENIX Association. https://www.usenix. org/conference/usenixsecurity21/presentation/wang-yan*, 2021.

[34] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "{FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 781–797.

[35] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1111–1128.

[36] Z. Zhao, Y. Wang, and X. Gong, "Haepg: An automatic multi-hop exploitation generation framework," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Cham: Springer International Publishing, 2020, pp. 89–109.

APPENDIX

A. *The State-Varied Feature in* `ptmalloc2`

The state-varied feature of heap operations is raised by the complex mechanism of heap allocators. For `ptmalloc2`, it mainly includes the following three mechanism.

Split. For operation $\mathcal{A}(x)$, allocators will firstly allocate the freed chunk with equivalent size. If there is no appropriate freed chunk in heap layout, it will cut part of the best fitting chunk for occupation, which is shown in Fig. 1(b).

Merge. For operation $\mathcal{F}(ptr)$, allocators will check adjacent chunks whether satisfies the merging condition. After merging, two or more chunks will turn to be a large free chunk.

Move. If more than one chains are used to manage freed chunks, for example, `fastbin` and `tcache` in `ptmalloc 2.31`, holes in these chains are occupied with different priority. Holes in `tcache` are always allocated first until they becomes empty and then holes in `fastbin` will be moved the `tcache`. This will cause the number of holes to be filled changes unexpectedly.

Fig. 9 shows a typical process of the move mechanism. Assume $\mathbb{C}_{x1}$ within maximum capacity M and $\mathbb{C}_{x2}$ are two chains to manage holes with size x, and $\mathbb{C}_{x1}$ has higher priority to allocate holes than $\mathbb{C}_{x2}$. Each hole in the chain is marked with number 1 to $M+2$ in case of identification. If holes in $\mathbb{C}_{x1}$ are all consumed, the next allocation $\mathcal{A}$(0x200) will occupy hole 1 and drive the allocator to transfer M holes in $\mathbb{C}_{x2}$ to $\mathbb{C}_{x1}$ at the same time. Due to the transfer order is different from that of allocation order in the chain, d changes from M to 2 after $\mathcal{A}$(0x200), and the original equation will fail to solve primitives' number as the d changes. In our solution, we reconstruct the equation within new d after move to deal with the state-varied feature.

B. *No Solution Demonstration Using Basic Capability-Based HLM with Side Effect Elimination*

$$\begin{cases} \boldsymbol{c}\cdot\boldsymbol{t}+d=0 \\ \boldsymbol{c}'\cdot\boldsymbol{\Delta t}+d'=0 \\ \boldsymbol{c}\cdot\boldsymbol{\Delta t}=0 \\ \widehat{\boldsymbol{t}}\in\mathbb{D}_{\mathbb{P}} \end{cases} \tag{13}$$

To eliminate side effect, we use above constraints to calculate $\boldsymbol{\Delta t}$. However, sometimes $\boldsymbol{\Delta t}$ may not be found, and we clarify the nonexistence of $\boldsymbol{\Delta t}$ in this part.

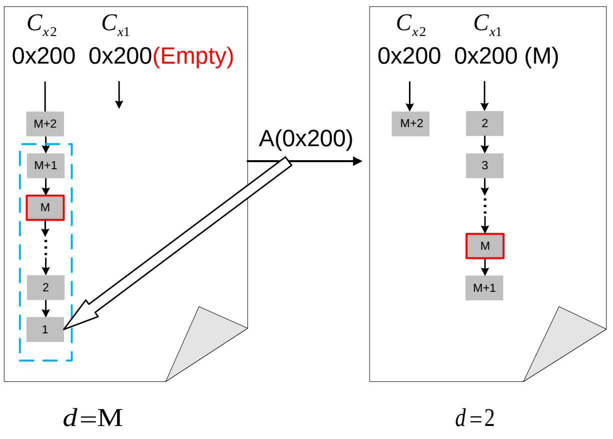


Fig. 9: Illustration of move mechanism.

Considering $n = 2$, which means only two primitives p_1 and p_2 with basic capabilities c_1 and c_2 respectively. While dealing with side effect, we could obtain following equations to calculate $\boldsymbol{\Delta t}$.

$$\begin{cases} c_1'\Delta t_1 + c_2'\Delta t_2 + d' = 0 \\ c_1\Delta t_1 + c_2\Delta t_2 = 0 \end{cases} \tag{14}$$

It is easily to solve the equations:

$$\Delta t_1 = \frac{c_1}{c_1c_2' - c_2c_1'}d', \quad \Delta t_2 = \frac{c_2}{c_1c_2' - c_2c_1'}d'$$

when $c_1c_2 < 0$, we have $\Delta t_1\Delta t_2 < 0$. Since Δt_1 and Δt_2 are non-negative integers, it means we cannot find $\boldsymbol{\Delta t}$.

In this circumstance, the constraints can not eliminate side effects, and we need extend capability to utilize side effects.