Hindawi
Scientific Programming
Volume 2020, Article ID 8883746, 21 pages
https://doi.org/10.1155/2020/8883746

Research Article

A Pattern-Based Software Testing Framework for Exploitability Evaluation of Metadata Corruption Vulnerabilities

Fenglei Deng, Jian Wang, Bin Zhang, Chao Feng, Zhiyuan Jiang, and Yunfei Su

College of Electronic Science, National University of Defense Technology, Changsha 410073, China

Correspondence should be addressed to Jian Wang; jwang@nudt.edu.cn and Yunfei Su; suyunfei@nudt.edu.cn

Received 1 July 2020; Revised 4 August 2020; Accepted 9 September 2020; Published 27 September 2020

Academic Editor: Miroslav Bures

In recent years, increased attention is being given to software quality assurance and protection. With considerable verification and protection schemes proposed and deployed, today's software unfortunately still fails to be protected from cyberattacks, especially in the presence of insecure organization of heap metadata. In this paper, we aim to explore whether heap metadata could be corrupted and exploited by cyberattackers, in an attempt to assess the exploitability of vulnerabilities and ensure software quality. To this end, we propose *RELAY*, a software testing framework to simulate human exploitation behavior for metadata corruption at the machine level. *RELAY* employs the heap layout serialization method to construct exploit patterns from human expertise and decomposes complex exploit-solving problems into a series of intermediate state-solving subproblems. With the heap layout procedural method, *RELAY* makes use of the fewer resources consumed to solve a layout problem according to the exploit pattern, activates the intermediate state, and generates the final exploit. Additionally, *RELAY* can be easily extended and can continuously assimilate human knowledge to enhance its ability for exploitability evaluation. Using 20 CTF&RHG programs, we then demonstrate that *RELAY* has the ability to evaluate the exploitability of metadata corruption vulnerabilities and works more efficiently compared with other state-of-the-art automated tools.

1. Introduction

Since cyberattacks have severely damaged the quality of software [1, 2], considerable defensive schemes have been taken to protect software security [3, 4]. In recent years, due to the deployment of mitigation mechanisms for stack-based vulnerabilities and compiler-level inspections, stack exploits have become increasingly more difficult to execute against hardened programs [5, 6]. However, there are still a large number of heap-based vulnerabilities in software written in insecure programming languages such as C and C++ [7–9]. These heap vulnerabilities increase the possibility of malicious attacks and pose a serious threat to software security [10].

The threat from hacking mainly comes from heap metadata corruption [11–13]. To improve performance, the heap allocator usually places dynamically allocated user data in the same memory area as the metadata that control the behavior of the allocator [14]. These metadata are unprotected, and vulnerabilities related to processing user data may overwrite metadata, which causes subsequent operations of the heap allocator to violate security assumptions [15]. Since this is an inherent defect of the heap allocator, it is not limited to any specific applications. Any program that employs the heap allocator may trigger such metadata attacks. Although developers have introduced enhanced mechanisms to protect heap metadata [16, 17], such as the 2017 patch, attackers can easily bypass the protection by slightly modifying their behavior [18]. Because of this adaptation, there have still been widespread attacks on metadata [19].

AEG (automatic exploit generation) was introduced to quickly generate exploits and improve the ability to assess the exploitability of software vulnerabilities [20–23]. In recent years, heap-based AEGs have improved, and some simple exploitations for user-data corruption can be processed [24, 25]. However, the exploitation of metadata requires a high degree of skill, and so this process is usually

combined with human expertise [26]. Existing AEGs do not integrate the required high-level expertise well, making it difficult to complete metadata attacks. Specifically, we can subdivide the problems of AEGs in metadata attacks into the following two aspects.

1.1. Coarse-Grained Heap Layout Arrangement. The existing AEG methods lack the expertise of the intermediate exploitation process, and it uses a saltatory strategy to complete the migration of memory state (MMS). For example, *Revery* [24] jumps directly from the panic state to the memory state including arbitrary address writing (AAW). However, in most cases, the exploitation of heap vulnerabilities, especially metadata corruption, needs to be closely combined with human expertise, and it is necessary to traverse intermediate memory states in a step-by-step strategy.

1.2. Blindness for Driving MMS. The exploitation in the kernel [27] and the interpreter [28] can directly trigger a specific heap operation through the exposed interface to construct the target memory state. *Heaphopper* [29] and *Archeap* [11] search for exploitation primitives by connecting the heap allocator to the driver program. They analyze the exploitability of the metadata in the allocator, without considering exploit generation of real programs embedding in the allocator. However, driving the MMS for user-level applications is still a nontrivial problem. It is necessary to satisfy the constraints of both path and state reachability and solve for the corresponding input. Existing AEG drives MMS through a blind search, and it is difficult to solve the above constraints.

To solve the above problems, we propose a new solution called *RELAY*, a software testing framework that simulates human exploitation behavior at the machine level to facilitate exploitability evaluation for software vulnerabilities. *RELAY* employs exploit patterns converted from human expertise to guide MMS, and it can generate exploits for metadata corruption.

We propose a serialization method for heap layout, which transforms the human expertise of exploitation into a sequence of memory states that the machine can understand and solve. The process of a metadata attack can be viewed as a combination of memory states following in a specific order, and the attack will fail even if there is only a slight change in this order. These memory states and their sorting processes are highly dependent on artificial knowledge such as *Unlink* [30]/*Hof* [31] exploit methods. Based on this information, *RELAY* employs a predicate-based memory characterization method to depict the critical memory state of exploitation and build a relationship chain of state migration called an exploit pattern. Through building an exploit pattern, we can refine the memory status of the heap layout, making the fuzzy and disordered heap layout more ordered. At the same time, it also converts a single difficult migration problem of memory state into multiple simple problems, which reduce the difficulty in solving constraints.

RELAY builds exploit pattern at the logical level and then needs to construct a state migration path (p_{sm}) at the machine level and drives programs to complete the MMS. Based on the heap layout serialization method, we employ a novel heap layout procedural method to drive the MMS. Specifically, with the memory operation primitive (MOP) graph and heap allocator model, *RELAY* first solves a set of heap operation sequences, which satisfy numerical and structural constraints to build a p_{sm}, and then approximates the target memory state by exploring paths, generating finally corresponding inputs. With the heap layout procedural method, *RELAY* can effectively drive the process of exploitation at a small cost.

In summary, this paper makes the following contributions:

(i) We propose a new solution *RELAY*, a software testing framework for simulating human exploitation behavior at the machine level, which can evaluate exploitability for metadata corruption vulnerabilities. In particular, *RELAY* can be easily extended and can continuously assimilate human expertise to enhance its ability for exploitability evaluation.

(ii) We propose a serialization method of heap layout, which transforms the high-level expertise of exploitation into an orderly memory state combination that the machine can understand and solve.

(iii) We propose a novel procedural method of heap layout to achieve MMS driving and activate exploit patterns at the machine level.

(iv) We implement a prototype of *RELAY* and demonstrate its effectiveness with CTF (Capture the Flag) and RHG (Robot Hacking Game) programs.

The rest of this paper is organized as follows: Section 2 describes research motivation and presents the overview of *RELAY*; Section 3 proposes the heap layout serialization method to construct exploit patterns; Section 4 proposes the heap layout procedural method to manipulate the heap layout according to the exploit patterns; in Section 5, we explain the implementation of *RELAY* and give an experimental evaluation of the system; Section 6 summarizes the work that is most relevant to ours; finally, we summarize the work and discuss the potential research orientation in the future in Section 7.

2. Motivation Example

In this section, we display exploitation for a common vulnerability to illustrate the challenges faced by existing AEGs and introduce our solution.

2.1. Metadata Exploitation Example. The heap allocator usually maintains metadata (such as *size*, *presize*, *fd,* and *bk* in *ptmalloc* [32]) to manage heap blocks and control heap behavior. Since the heap allocator does not fully protect the metadata, if developers do not pay attention to the safety rules when programming and invoke the heap allocator without due consideration, a hacker will be able to break the software security by corrupting the metadata. Due to the

above reasons, metadata attacks are widely used throughout the real world.

Security analysts have further summarized some attack methods against metadata corruption, such as *FastBin attack* [33], *Unlink*, *Hof*, and *chunk overlap&FastBin attack* [34]. For *FastBin attack*, we need to modify the *fd* pointer of the released heap block and fake a *FastBin* heap block and then take down the forged heap block from the *FastBin* linked list through multiple allocations, resulting in arbitrary address writing (AAW). For *Unlink*, we need to modify the fd and bk pointers of the freed chunk and trigger the unlink operation by merging heap blocks for an AAW. For Hof, we need to modify the size of *TopChunk*, calculate the distance between target address and current *TopChunk* address, and trigger the allocation with a larger size. Finally, the system will return the target memory block to users as an allocated chunk. For *chunk overlap&FastBin attack*, we need to reuse the common memory between two heap blocks and thus tamper with *fd* pointer to continue with *FastBin attack* (more specific details will be described later).

It is complicated to implement these exploitation methods, including precise calculation and organized layout arrangement. But as shown above, the metadata may be damaged easily by *heap overflow*, *Use-After-Free* (UAF), *off-by-one* [35], and other vulnerabilities due to inadequate protection. So these exploitation methods are widely used together with the frequent metadata attacks. Therefore, the critical issue we need to solve is how to combine these popular but complicated exploitation methods to strengthen the ability of the machine to evaluate the exploitability of vulnerabilities automatically and thus maintain the software security.

Next, we will present the existing problems of AEG through case analysis and explore the solutions. We use the common *off-by-one* vulnerability as an example, which is a vulnerability often used to destroy size metadata, combined with the *FastBin attack* and *unlink* methods to cause arbitrary code execution. Figure 1 shows a typical exploitation process of an *off-by-one* vulnerability. The memory operations are required at each step, and the corresponding critical memory states are shown as follows:

(1) By overflowing one byte, the last byte of the size metadata of allocated chunk 2 can be controllable.

(2) By freeing chunk 2 and then reoccupying the position, chunk 2 can be extended and overlapped with chunk 3.

(3) By releasing chunk 3 and then controlling its contents with the extended chunk 2, *fd* pointer of chunk 3 can be controllable.

(4) By allocating chunks multiple times, memory 4 (callable memory) pointed by the controlled *fd* pointer can be allocated. We achieve an arbitrary address allocation in this step.

(5) By writing data to chunk 4, this callable memory is controllable.

(6) By invoking memory 4, the program reaches the state of control-flow hijacking.

2.2. Challenge. The above process can represent most metadata attacks, and they usually need to complete the migration between multiple states by multiple steps. The existing AEG method cannot refine these intermediate memory states. It is difficult to directly search for primitives such as control-flow hijacking and arbitrary address writing (AAW) from the panic state. Besides, it can be seen that the exploitation of metadata requires an in-depth understanding of the internal mechanisms of the heap allocator; the definition and ordering of intermediate states during exploitation is also based on human expertise. Existing AEGs do not use human knowledge well, and it cannot simulate the behavior of artificial exploitation at the machine level.

In addition, for each step, the program needs to trigger a specific operation such as write, *malloc*, and *free* to enter an intermediate state. After these operations are triggered, the correction of the parameters or write content corresponds to the solving of numerical constraints, which can be completed by the symbolic execution tool [36, 37]. How to trigger these heap operations corresponds to the solving of structural constraints. We need to drive the program to enter a specific path branch at a specific time to solve these structural constraints, which is usually done by exploring the path with fuzzing tools [38, 39]. Obviously, it is not easy to complete the oriented path coverage. However, the MMS usually requires triggering these operations frequently. For example, step 4 may require triggering *malloc* repeatedly to remove chunk 4 from the free list, which often brings with it greater complexity and consumption. Existing AEG methods have a limited ability to solve such structural constraints, and exploring paths blindly is usually inefficient.

2.3. Solution. Figure 2 shows *RELAY*'s framework. In the serialization part of the heap layout, *RELAY* transforms human expertise of exploitation into an exploit pattern library. The procedural part of the heap layout mainly consists of two modules: the migration path construction (MPC) module and the migration path drive (MPD) module. First, this part takes PoC as input, analyzes the current memory state, and obtains the corresponding state migration pair (r_{sm}) from the exploit pattern library. Then, the MPC builds the p_{sm} that satisfies the structural constraints in r_{sm}, and then, the MPD is responsible for driving the p_{sm} in the real environment and generates the corresponding input. The two iterate through the intermediate states, solve the constraints that satisfy path reachability and state reachability, and generate control-flow hijacking inputs. Finally, the exploit generation module outputs exploits to complete the specific attack.

2.3.1. Heap Layout Serialization. First, this module employs the predicate-based memory characterization method to describe r_{sm}. Then, by combining the advanced knowledge observed in human exploitation behavior, it assembles the r_{sm} to build an exploit pattern for a typical exploitation process. This exploit pattern is a migration chain composed of a series of intermediate states in the exploitation process. For example, the migration from the state of controllable *fd*

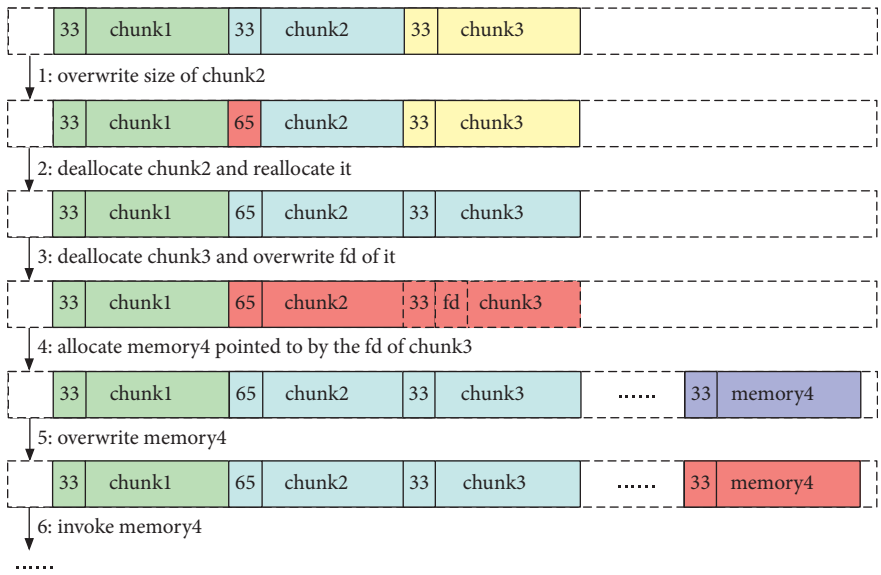

Figure 1: Typical exploitation process for off-by-one vulnerability, the number before the chunk represents its size.

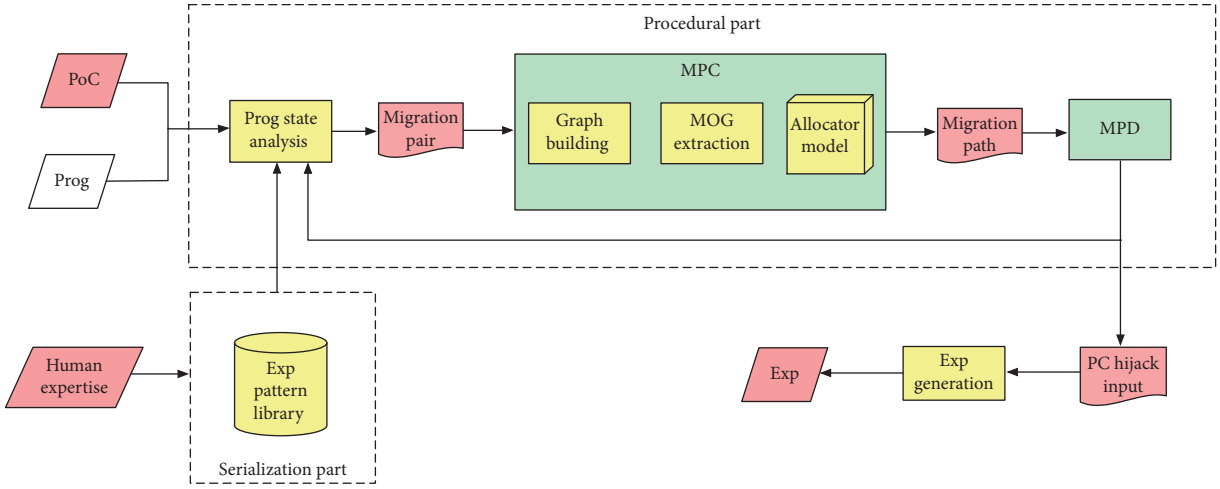

Figure 2: Overview of *RELAY*'s frame. PoC represents proof-of-concept.

pointer in step 3 to the state of arbitrary address allocation in step 4 can form a section of the exploit pattern.

2.3.2. Migration Path Construction Module (MPC). First, this part obtains a control-flow graph (CFG) through static analysis and based on this builds a memory operation primitive (MOP) graph from which the memory operation groups are extracted (MOGs). Then, we establish a heap allocator model to infer the memory state corresponding to the MOG and build the p_{sm} by comparing the target memory states of the migration pair. Finally, we utilize the triggering probability of the path to solve the p_{sm} with the smallest driving cost. For example, for a migration demand from state 3 to state 4, the p_{sm} can be solved as MMM (M represents *malloc*). This module can interact with an MPD.

2.3.3. Migration Path Driver Module (MPD). After receiving the $p_{sm} = A, A, A$ (A for allocation) from the MPC module, the MPD drives the p_{sm} with fuzzing at the machine level and attempts to activate state 4. If state 4 is not activated after driving, the MPD will reselect the p_{sm} constructed by the MPC and launch it again for the current migration pair. If activated, the MPD will retain the input and switch to the next migration pair 4-5. By repeatedly executing the MPC and the MPD, *RELAY* will activate all the intermediate states at the machine level.

2.3.4. Exploit Generation Modules. This module receives test cases that can hijack PC (program counter) and perform symbolic constraint solving on the PC and the related controllable memory space to generate the final exploits. In terms of AEG for EIP-hijacking vulnerabilities such as *Mayhem* [23] and *CRAX* [40], the critical part of these schemes is that, after hijacking EIP, the control flow will be guided to the target position by symbolic execution, and thus, the PoC input is transformed into EXP input. We reuse

their work for our exploit generation module. Specifically, if it is found that the EIP register contains a symbolic value, and we will arrange the shellcode into memory with permission $W + X$ and then add a symbol constraint to make the symbol value point to the shellcode location, redirecting control flow to that location and generating EXP for arbitrary code execution. Since this is a previous work, we will not repeat it in this paper.

3. Serialization Method of Heap Layout

The existing AEGs perform badly in refining the heap layout and often "jumps" during the processing of the heap layout. For example, *Revery* [24] jumps directly from the state of triggering a vulnerability to the state of hijacking the control flow. The limitation is that they adopt a random exploration strategy in the heap layout process, and they are not guided sufficiently by human expertise. Therefore, we need to explore a method that can guide the heap layout process, apply this expertise at the machine level, and make the machine solve the complex problems of metadata attacks using the way of human thinking.

To this end, we propose a serialization method of heap layout, combined with the way humans think about crafting exploits, to decompose the problem of solving the highly complex process of the heap layout into multiple subproblems with low complexity. First of all, we employ a machine-understandable memory descriptive language to represent the memory attributes, the data, and structural relationships between memory objects, to ensure accurate characterization of intermediate states. Then, we refine the "fragmented knowledge" of manual exploitation to model the migration pair and orderly combine the migration pair according to typical exploit methods. Finally, we can build an exploit pattern that can be utilized to guide the heap layout process.

3.1. Predicate-Based Memory Characterization Method. This section employs a formal method of describing memory states which can be easily translated into programming languages and can be implemented at the machine level. Specifically, this method characterizes the memory state through three elements: memory objects, object attributes, and relationships between objects. The relationships between objects include numerical and structural relations. This section first clarifies the definition of memory objects. Then, it describes the properties of these memory objects and the numerical and structural relationships between these objects based on predicates.

3.1.1. Extended Representation of Memory Objects. Memory objects are usually represented by a tuple $O_{NS} = a, w$, where a is the address of the memory unit and w is the width. For example, $0\text{xdeadbeef}, 4$ indicates the memory unit of consecutive 4 bytes located in the address 0xdeadbeef. We represent the structured memory objects such as the heap block on the basis of the tuple. These memory objects are discussed below in the 32 bit architecture.

Although the structure objects in the heap memory such as heap blocks and free linked lists are complicated, the basic processed objects are heap blocks. Therefore, the structured description of heap memory is also based on heap blocks. The heap block is divided into the allocated state and the *free*-ed state, and so we utilize different data structures to characterize them.

The heap block in the allocated state can be formalized as $c_A = a, s_p, s_c, D, A, M, P$, where a represents the address of the chunk; s_p represents the size of the physical neighbouring chunk at the low address; s_c represents the size of the chunk; D represents the data area of the chunk; and $A/M/P$ represents whether it is the main allocated area, whether it is mapped memory, and whether the physical neighbouring chunk at a low address is in the allocated state. All of the elements in the formal representation of c_A are O_{NS}-type memory objects (a can be regarded as a 4 byte memory block storing the chunk address). The heap block in the *free*-ed state adds a forward pointer P_{fd} and a backward pointer P_{bk} (both are O_{NS}-type memory objects) on the basis of c_A, that is, $c_F = a, s_p, s_c, D, A, M, P, P_{\text{fd}}, P_{\text{bk}}$.

Furthermore, the free list can be defined as $L = r, d_i, d_d, c_1, c_2, \ldots, c_N$, where c_i is a *free*-ed chunk; N is of numeric type and represents the length of the free list L; r is of Boolean type and represents whether the linked list is for random access; and d_i and d_d are only for nonrandom access linked lists, and they are of numeric types, respectively, representing the direction of inserting and deleting nodes of the linked list (0 represents the direction from the head of the linked list, and 1 represents the direction from the tail of linked list). According to whether d_i and d_d are the same, the free list for nonrandom access can be divided into a one-way operation list (such as *FastBin* in *ptmalloc*) or a two-way operation list (such as *SmallBin*).

Based on the heap block and free linked list, the heap memory can be formally described as $\mathscr{H} = A, L_F, L_S, L_L, U, T, R$, where $A = c_1, c_2, \ldots$ is the set of allocated chunks in the entire memory space (A contains not only the heap blocks allocated in the heap memory but also the heap blocks that attackers may forge); $L_F/L_S/L_L$ stands for *FastBin*, *SmallBin*, and *LargeBin*, respectively. U stands for *UnsortedBin*; T stands for *TopChunk*; R stands for *Last Remainder* (the structure of $U/T/R$ is similar to a chunk in the *free*-ed state, so we will not go into details).

With the representation method of structured heap memory, we can complete the fine-grained description of the heap memory state. For example, $\exists c_i \in \mathscr{H}.A(C(c_i.P))$ depicts the situation where the P flag of allocated memory has overflowed (we use $C(\text{obj})$ to indicate that obj are controlled by external input, which is strictly defined later), and $C(\mathscr{H}.T.s_c)$ depicts that the size field of the *TopChunk* has overflowed. It can be seen from the former expression that an accurate description of the memory state must be based on the accurate definition of the attributes of the memory

objects (such as controllable C) and relationships between memory objects (such as $c_i \in \mathscr{H}.A$), so we will separately introduce their definition in the following sections.

3.1.2. Attribute Predicate Definition. We have extended conventional memory attributes to support complex attribute descriptions of memory units. For each memory unit, we introduced three additional attributes besides the readable, writable, and executable attributes: controllable, callable, and chunk association, as shown in Table 1.

The following three examples explain these three new attributes. It can be expressed as $C(\mathscr{H}.T.s_c)$ if TopChunk's size field is overflowed. $I(0\text{xdeadbeef}, 4)$ indicates that the data stored in consecutive 4 bytes of memory unit located at address 0xdeadbeef can be directly or indirectly migrated to the PC register (for example, the data of the memory unit where the return address is located will be written to the PC register when the function returns, and the data in the GOT table will be migrated to the PC if invoking the corresponding function). $CP(0\text{xdeadbeef}, 4)$ indicates that the data stored in the 4 consecutive bytes of memory unit located at address 0xdeadbeef can be used as a heap block pointer for allocation, release, and writing operations.

3.1.3. Relation Predicate Definition. This section analyzes the relationships between memory objects from two aspects: the numerical relationship (γ_S) and the structural relationship (γ_D). The numerical relationship γ_S is used to indicate the relationship of memory objects at the numerical level. Common numerical relationships are shown in Table 2. For example, it can be expressed as $\mathrm{EQ}(c_1.s_c, c_2.s_c)$ if the size of the heap block c_1 is equal to that of the heap block c_2. If the size of the heap block c meets the requirement of the free linked list $\mathscr{H}.L_F$, it can be expressed as $\mathrm{SZ}(c.s_c, \mathscr{H}.L_F)$. Besides representing the numerical relationship between objects, γ_D can also be used to express the relationship between objects and values; for example, $\mathrm{EQ}(c_1.s_c, 64)$ indicates that the size of the chunk c_1 is 64 bytes.

The structural relationship γ_S is used to indicate the relationship between memory objects at the structural level. The 9 typical structural relationships are shown in Table 3. For example, it can be expressed as $\mathrm{IN}(c_1, \mathscr{H}.L_F)$ if the *free*-ed chunk c1 belongs to the *FastBin* list. For the unstructured memory object mem that stores the pointer pointing to the data area of chunk c, it can be expressed as $\mathrm{PT}(\mathrm{mem}, 4, c.D)$. An equivalence relation (IS) is used to indicate that the elements that constitute two objects are the same and the memory area occupied is completely the same; that is, the two objects are essentially the same object. An above relationship (HT) and below relationship (BT) are used to describe the positional relationship of memory objects in physical space. The forward/backward relationship (FD/BK) is similar to the HT/BT relationship. The only difference is that the FD/BK is used to indicate that the adjacency relationship is higher or lower in the current structured object level. For example, $\mathrm{FD}(c_1, c_2)$ indicates that the chunk next to the chunk c_1 in the direction of high address is c_2. $\mathrm{LAP}(c_1, c_2, \mathrm{mem})$ is used to indicate that there is a common memory object mem between c_1 and c_2.

3.2. Exploit Pattern Construction. Exploitation can be expressed as the migration process of a special memory state in the program state space. Therefore, we employ state-space notation to describe the exploit process. State-space notation can be defined as quaternion S, b, O, G, where S is the set of possible states, $b \in S$ is the initial state of the program, O is the set of possible operators, and G is the target state. This section describes the state migration process at the model level by constructing the exploit pattern. The definition of the operator and the realization of the state migration at the machine level are described in the next section.

The experience of artificial exploitation consists of a series of pieces of "fragmented knowledge;" for example, an "AAW" may result in "control-flow hijacking" and "controllable format string" may result in "AAW". The process of manual exploitation essentially organizes these pieces of "fragmented knowledge" in a specific order. For example, the process of implementing arbitrary code execution using format string vulnerabilities can be expressed as "format string controllable" $\longrightarrow$ "AAW" $\longrightarrow$ "control flow hijacking". How to represent this "fragmented knowledge" and how to organize it is the key to applying human expertise to the state migration process.

3.2.1. State Migration Pair. We employ state migration pairs (r_{sm}) to represent "fragmented knowledge." r_{sm} can be expressed as $S \longrightarrow^{P_{sm}} S'$, where S and S' are the memory states portrayed by predicates and $p_{\mathrm{sm}} = o_1, o_2, \ldots, o_N$ represents the migration path formed by the orderly combination of several operations. Table 4 lists several typical state migration pairs related to heap exploits under 32 bit systems, where $\mathrm{val}(x)$ represents the integer value at address x, and $\mathrm{add}(o_1, o_2/v)$ represents adding the objects o_1 and o_2 (or numerical value v) on the numerical level.

Table 4 does not need to ensure the completeness of the characterization. Firstly, the state migration pairs in this table can be continuously updated according to human experience; and secondly, the source and destination states of the migration pair do not need to be fully characterized for all memory and register objects but only focus on the critical memory objects, structural relationships, and numerical relationships based on the human experience.

3.2.2. Exploit Pattern. The artificial exploit methods can be regarded as a combination of "fragmented knowledge" in a specific order. We combine r_{sm} in the same order to build an exploit pattern to simulate human exploit methods.

It should be noted that, before constructing the exploit pattern, we need to backpropagate the constraints collected in the migration pairs to ensure the effectiveness of the

Table 1: Attribute predicate definition.

Object attribute	Attribute predicate	Description
Readable	$R(o)$	Can read from the memory of object o
Writable	$W(o)$	Can write to the memory of object o
Executable	$X(o)$	The data in the memory of object o can be executable
Controllable	$C(o)$	The data in the memory of object o can be controlled by input
Callable	$I(o)$	The data in the memory of object o can migrate to PC
Chunk association	$CP(o)$	The data in the memory of object o can be used as a pointer

Table 2: Numerical relationship between memory objects.

Object relation	Relation predicate	Description
Equal	$EQ(o_1, o_2/v)$	o_1 equals o_2/numeral v
Unequal	$NE(o_1, o_2/v)$	o_1 unequals o_2/numeral v
Greater	$GT(o_1, o_2/v)$	o_1 is greater than o_2/numeral v
Greater or equal	$GE(o_1, o_2/v)$	o_1 is greater than or equal to o_2/numeral v
Less	$LT(o_1, o_2/v)$	o_1 is less than o_2/numeral v
Less or equal	$LE(o_1, o_2/v)$	o_1 is less than or equal to o_2/numeral v
Meet size	$SZ(c, L)$	The size of the chunk c lets the requirements of the free list L into the chain

Table 3: Structural relationship between memory objects.

Object relation	Relation predicate	Description
Subordinate	$IN(o_1, o_2)$	o_1 belongs to o_2
Pointer	$PT(o_1, o_2)$	o_1 pointer to o_2
Equivalence	$IS(o_1, o_2)$	o_1 is same to o_2
Higher	$HT(o_1, o_2)$	o_1 is higher than o_2
Lower	$BT(o_1, o_2)$	o_1 is lower than o_2
Forward	$FD(o_1, o_2)$	o_1 is front of o_2
Backward	$BK(o_1, o_2)$	o_1 is behind o_2
Overlap	$LAP(o_1, o_2)$	o_1 and o_2 have the same memory mem

Table 4: Typical state migration pairs for heap vulnerabilities.

Serial number	Fragmented knowledge	Migration pair
#1	If callable memory is controlled, it may cause control-flow hijacking	$C(<p,4>) \wedge I(<p,4>) \xrightarrow{P_{sm}} C(PC)$
#2	If size of top chunk is controlled, it may cause arbitrary chunk allocation in the high address direction	$C(H.T.Sc) \xrightarrow{P_{sm}} IN(c, \mathcal{H}.A) \wedge C(c.a) \wedge GE(c.a, \mathcal{H}.T.a) \wedge LE(c.a,add(\mathcal{H}.T.a.\mathcal{H}.T.S_c))$
#3	The data area of the allocated chunk can be controlled by external input	$IN(c,\mathcal{H}.A) \wedge LE(p,add(c.a, c.s_c)) \wedge GE(p,c.D.a) \xrightarrow{P_{sm}} C(<p,4>)$
#4	If *FastBin* list has a controllable chunk, it may cause arbitrary chunk allocation	$IN(c,\mathcal{H}.L_F) \wedge C(c.a) \wedge SZ(c.s_c,\mathcal{H}L_F) \xrightarrow{P_{sm}} IN(c',\mathcal{H}.A) \wedge C(c'.a) \wedge EQ(c.a, c'.a)$
#5	Pointer to low address −12 bytes can be controlled by external input	$C(<p,4>) \wedge PT(<p,4>,<p\text{-}12,4>) \wedge W(<p,4>) \xrightarrow{P_{sm}} C(<p,4>) \wedge CP(<p,4>)$
#6	If the heap block pointer is controlled, a chunk with a controllable address can be obtained	$CP(<p,4>) \wedge C(<p,4>) \wedge SZ(<val(val(p)+4),4>,\mathcal{H}.L_F) \xrightarrow{P_{sm}} IN(c, \mathcal{H}.L_F) \wedge C(c.a) \wedge EQ(c.a,val(p))$
#7	Backward merge process of heap block	$IN(c, \mathcal{H}.L_s) \wedge BCS(c, c_1, c_2) \wedge BCD(c,c_1, c_2) \wedge PT(<p,4>, c_1.D) \xrightarrow{P_{sm}} PT(<p,4>,<p\text{-}12,4>)$
#8	The writable area pointed by the heap pointer can be controlled by external input	$CP(<p, 4>) \wedge PT(<p, 4>,<p', 4>) \wedge W(<p', 4>) \xrightarrow{P_{sm}} C(<p', 4>)$
#9	If size of allocated chunk is controllable, it may cause chunk overlap	$IN(c, \mathcal{H}.A) \wedge C(c.s_c) \xrightarrow{P_{sm}} IN(c_H, \mathcal{H}.A) \wedge HT(c_H, c) \wedge Lap(c, c_H, p)$
#10	If chunks are overlapped, it may cause chunk in *FastBin* list controllable	$IN(c_H, \mathcal{H}.A) \wedge HT(c_H, c) \wedge Lap(c, c_H, p) \xrightarrow{P_{sm}} IN(c_F,\mathcal{H}.L_F) \wedge C(c_F.a) \wedge SZ(c_F.s_c, \mathcal{H}.L_F)$

Dot notation is used to indicate access to fields in an object, and comma is used to separate two objects in predication.

exploit pattern. For example, migration pair $1^{\#}$ requires that the memory at p in the original state has a callable attribute, so $3^{\#} \longrightarrow 1^{\#}$ combination requires backpropagation $I(p, 4)$ to the source state of $3^{\#}$ migration pairs, i.e., $\mathrm{IN}(c, \mathscr{H}.A) \wedge \mathrm{LE}(p, \mathrm{add}(c.a, c.s_c)) \wedge \mathrm{GE}(p, c.D.a) \wedge I(p, 4)$. In order to facilitate expression, we distinguish memory objects in different states through the subscript (for instance, p_1 and p_2 represent the same memory area p in different states), so the $3^{\#} \longrightarrow 1^{\#}$ combination can be expressed as the following formula after the constraint propagation is completed:

$$
\begin{aligned}
& 3^{\#} \longrightarrow 1^{\#}, \\
& \mathrm{IN}(c_2, \mathscr{H}.A) \Lambda \mathrm{LE}(p_2, \mathrm{add}(c_2.a, c_2.s_c)) \\
& \quad \Lambda \mathrm{GE}(\langle p_2, c_2.D.a \rangle) \Lambda \mathrm{EQ}(\langle p_2, p_1 \rangle), \\
& \xrightarrow{P_{sm}} C(\langle p_1, 4 \rangle) \Lambda I(\langle p_1, 4 \rangle), \\
& \xrightarrow{P_{sm}} C(PC).
\end{aligned} \tag{1}
$$

We have established corresponding exploit patterns for common exploit methods. *FastBin attack* can be expressed as $4^{\#} \longrightarrow 3^{\#} \longrightarrow 1^{\#}$, and the exploit pattern is as follows:

$$
\begin{aligned}
& \mathrm{fast\_bin\_exp}, \\
& \mathrm{IN}(c_3, \mathscr{H}.L_F) \Lambda C(c_3, a) \Lambda \mathrm{SZ}(c_3.s_c, \mathscr{H}.L_F) \Lambda \mathrm{EQ}(c_3.a, c_2.a), \\
& \xrightarrow{P_{sm}} \mathrm{IN}(c_2, \mathscr{H}.A) \Lambda \mathrm{LE}(p_2, \mathrm{add}(c_2.a, c_2.s_c)) \Lambda \mathrm{GE}(p_2, c_2.D.a) \Lambda \mathrm{EQ}(p_2, p_1), \\
& \xrightarrow{P_{sm}} C(\langle p_1, 4 \rangle) \Lambda I(\langle p_1, 4 \rangle), \\
& \xrightarrow{P_{sm}} C(PC).
\end{aligned} \tag{2}
$$

Unlink can be expressed as $7^{\#} \longrightarrow 5^{\#} \longrightarrow 8^{\#} \longrightarrow 1^{\#}$, and the exploit pattern is as follows:

$$
\begin{aligned}
& \mathrm{Unlink\_exp}, \\
& \mathrm{IN}(c_4, \mathscr{H}.L_S) \Lambda \mathrm{BCS}\left(c_4, c_{4_1}, c_{4_2}\right) \Lambda \mathrm{BCD}\left(c_4, c_{4_1}, c_{4_2}\right) \Lambda \mathrm{PT}\left(p_4, 4, c_{4_1}.D\right), \\
& \Lambda \ \mathrm{EQ}(p_4, p_3), \\
& \xrightarrow{P_{sm}} \mathrm{CP}(\langle p_3, 4 \rangle) \Lambda \mathrm{PT}(\langle p_3, 4 \rangle, \langle p_3 - 12, \ 4 \rangle) \Lambda W(\langle p_3, 4 \rangle) \ \Lambda \mathrm{EQ}(p_3 - 12, p_2), \\
& \xrightarrow{P_{sm}} \mathrm{CP}(\langle p_2, 4 \rangle) \ \Lambda \mathrm{PT}(\langle p_2, 4 \rangle, \langle p_2', 4 \rangle) \Lambda W(\langle p_2', 4 \rangle) \Lambda \mathrm{EQ}(p_2', p1), \\
& \xrightarrow{P_{sm}} C(\langle p_1, 4 \rangle) \Lambda I(\langle p_1, 4 \rangle), \\
& \xrightarrow{P_{sm}} C(\mathrm{PC}).
\end{aligned} \tag{3}
$$

Hof can be expressed as $2^{\#} \longrightarrow 3^{\#} \longrightarrow 1^{\#}$, and the exploit pattern is as follows:

$$\begin{aligned}
&\text{hof_exp},\\
&C(\mathscr{H}.T.s_c),\\
&\xrightarrow{P_{sm}} \text{IN}(c_2, \mathscr{H}.A) \Lambda \text{GE}(c_2.a, \mathscr{H}.T.a) \Lambda \text{LE}(c_2.a, \text{add}(\mathscr{H}.T.a, \mathscr{H}.T.s_c)),\\
&\quad \Lambda \text{LE}(p_2, \text{add}(c_2.a, c_2.s_c)) \Lambda \text{GE}(p_2, c_2.D.a) \Lambda \text{EQ}(p_1, p_2),\\
&\xrightarrow{P_{sm}} C(\langle p_1, 4\rangle) \Lambda I(\langle p_1, 4\rangle),\\
&\xrightarrow{P_{sm}} C(PC).
\end{aligned} \tag{4}$$

The example in Section 2.1 shows a typical exploit method for *off-by-one* vulnerabilities, and we can call it *chunk overlap&FastBin attack*, which can be expressed as $9^{\#} \longrightarrow 10^{\#} \longrightarrow 4^{\#} \longrightarrow 3^{\#} \longrightarrow 1^{\#}$:

$$\begin{aligned}
&\text{overlap_exp},\\
&\text{IN}(c, \mathscr{H}.A)\ \Lambda C(c.s_c),\\
&\xrightarrow{P_{sm}} IN(c_H, \mathscr{H}.A) \Lambda \text{HT}(c_H, c) \Lambda \text{Lap}(c.c_H, p),\\
&\xrightarrow{P_{sm}} \text{IN}(c_3, \mathscr{H}.L_F) \Lambda C(c_3, a) \Lambda \text{SZ}(c_3.s_c) \Lambda \text{EQ}(c_3.a, c_2.a),\\
&\xrightarrow{P_{sm}} IN(c_2, \mathscr{H}.A) \Lambda C(p_2, \text{add}(c_2.a, c_2.s_c)) \Lambda \text{GE}(p_2, c_2.D.a) \Lambda \text{EQ}(p_2, p_1),\\
&\xrightarrow{P_{sm}} C(\langle p_1, 4\rangle) \Lambda I(\langle p_1, 4\rangle), \xrightarrow{P_{sm}} C(\text{PC}).
\end{aligned} \tag{5}$$

3.2.3. Guidance of Exploit Generation. It can be seen that the exploit pattern can characterize the migration relationship of the intermediate state during the exploitation process, including the data relationship and structural relationship between the memory objects. In order to further guide exploit generation, we divide the predicates used to characterize the memory state into two categories: numerically dependent predicates and structurally dependent predicates. The data dependency predicates refer to EQ, NEQ, SZ, and other predicates that reflect the numerical relationship between memory objects; structure dependency predicates mainly include the extended memory attribute predicate (C, I, and CP) and IN, FD, and other predicates that reflect the structural relationship between memory objects. These two types of predicates correspond to form two different types of constraints, namely, numerical constraints ψ_D and structural constraints ψ_S. The exploit pattern can transform the problem of automatic generation of heap exploits into the problem of solving numerical constraints ψ_D and structural constraints ψ_S.

We employed the fast_bin_exp exploit pattern as an example to explain how to guide the exploit generation by solving these two types of constraints. The exploit consists of 4 steps corresponding to 4 states in fast_bin_exp:

(1) Now, the *fd* pointer of c_3 chunk in the *FastBin* list is controllable, which has satisfied the following structural constraints:

$$\psi_S = \text{IN}(c_3, \mathscr{H}.L_F) \wedge C(c_3, a). \tag{6}$$

But, we have not made the c_3.a and c_2.a equal, which means the following numerical constraints have not been satisfied:

$$\psi_D = \text{SZ}(c_3.s_c, \mathscr{H}.L_F) \wedge \text{EQ}(c_3.a, c_2.a). \tag{7}$$

In order to solve ψ_D, we should calculate the address of c_2. The data area of c_2 needs to cover p_2, and p_2 needs to be callable, such as *malloc_hook*. So, we can update the numerical constraints as follows:

$$\begin{aligned}
\psi_D = {} & \text{SZ}(c_3.s_c, \mathscr{H}.L_F) \wedge \text{LE}(\text{mallc_hook}, \text{add}(c_3.a, c_3.s_c))\\
& \wedge \text{GE}(\text{malloc_hook}, c_3.D.a).
\end{aligned} \tag{8}$$

We solve it by using a constraint solver.

(2) The following numerical constraints have been satisfied:

$$\psi_D = \mathrm{LE}(p_2, \mathrm{add}(c_2.a, c_2.s_c)) \wedge \mathrm{GE}(p_2, c_2.D.a) \wedge \mathrm{EQ}(p_2,\ p_1). \tag{9}$$

The following structural constraints require making c_2 become an allocated chunk:

$$\psi_S = \mathrm{IN}(c_2, \mathscr{H}.A). \tag{10}$$

If c_2 is located in the Nth position from the head end of *FastBin* linked list (namely, there are N-1 heap blocks before c_2), we need, according to the FIFO (first in last out) mechanism of *FastBin*, to trigger the allocation for consecutive N times to take out c_2, and the size of the allocation needs to satisfy the requirements of the *FastBin* list.

(3) Now that numerical constraints have been satisfied, we only need to solve the following structural constraints:

$$\psi_S = C(\langle p_1, 4\rangle) \wedge I(\langle p_1, 4\rangle). \tag{11}$$

$p_1$is malloc_hook, which satisfies $I(\langle p_1, 4\rangle)$, so we just need to cover the malloc_hook to satisfy $C(\langle p_1, 4\rangle)$.

(4) After controlling the *malloc_hook*, we need to invoke *malloc* to transfer the data stored in the *malloc_hook* to the PC register, which implements C(PC).

4. Procedural Method of Heap Layout

The serialization method of heap layout divides the exploitation process into an ordered set of multiple intermediate states at the logical level by constructing the exploit pattern and indicates the direction for the heap layout.

We also need to activate the intermediate state at the machine level and implement the migration process of the heap layout.

4.1. Framework. With the exploit pattern, we reduce the complexity of traversing the state space of the program, but the extremely large path space also places a significant burden on exploitation. It is often inefficient to drive the program to the target state by randomly exploring the program path. Taking the exploitation process for *off-by-one* as an example, in moving from state 3 to state 4, a certain number of heap allocation operations are required and the size of the allocated heap block must meet the requirements of the linked list. Blindly traversing the program path will result in a random combination of memory operations and their parameters, which is very inefficient.

Therefore, we divide the heap layout process into two stages: migration path construction (MPC) and migration path driving (MPD). The MPC phase constructs a set of state migration paths P_{sm} according to the exploit pattern, and the MPD phase attempts to drive the program along the p_{sm}. By building the p_{sm} to guide the state migration of the program, we can reduce the complexity of traversing the path space and improve the driving efficiency.

The overall design of the procedural method of heap layout is shown in Figure 3. Through the heap layout serialization module, we build the exploit pattern. The program state analysis module analyzes the current program state and finds r_{sm} which source state matches with it. We can decompose r_{sm} into numerical constraints ψ_D and structural constraint ψ_S (as explained in the previous section). In each round of processing an r_{sm}, the numerical constraint solving module first corrects the input related to ψ_D. Next, the MPC generates a P_{sm} set according to the structural constraint ψ_S. Then, the MPD performs searches according to the p_{sm} and drives the state migration of the program. In the loop, we first adopt an optimization strategy. For the migration path set P_{sm}, we give priority to the one with the smallest driving cost and reduce the complexity of the driving migration path as much as possible. Secondly, we add a feedback mechanism. If the new state is not matched with the target state after the path is driven (the matching method will be explained later), we give up the intermediate test case and restore to the original state of the program, remove the current migration path from the set, and then start a new round of retrying to drive. If matched, the program state analysis will pass the next r_{sm} to the MPC and the MPD. By repeatedly solving ψ_D and ψ_S, *RELAY* can complete the whole MSM, activate all intermediate states, and finally generate the hijacking PC input.

Since the existing symbolic execution technology can solve the numerical constraints in the exploit pattern [41], we will not repeat the process of solving ψ_D but explain how to automatically solve the structural constraints in the exploit pattern.

4.2. Migration Path Construction (MPC). The MPC constructs migration paths according to structural constraints. First, the MPC constructs the memory operation primitive (MOP) graph through static analysis and extracts the memory operation groups (MOGs) from it. Then, it employs the heap allocator model to solve the p_{sm} from the MOG according to the structural constraints and selects the k paths with the smallest driving cost (the relevant definition will be described later) to build the optimized set P^*_{sm}. In the MOP graph section, we give a formal definition of p_{sm} and explain why structural constraints can be solved by using an ordered set of MOPs. Based on this, we construct a MOP graph and introduce a method to extract the heap operation groups from it. The heap allocator model is relatively independent and easy to expand to support different heap allocators. It establishes a mapping relationship between heap operations and memory states, and it is used to examine the effect of heap operations. In order to find the solution, it takes the structural constraints as input and employs the MOP graph to obtain all possible groups of heap operations. Then, it utilizes the heap allocator model to solve the group of heap operations that can meet the structural constraints as p_{sm} and finally takes the k paths with the smallest driving cost to form a output set P^*_{sm}.

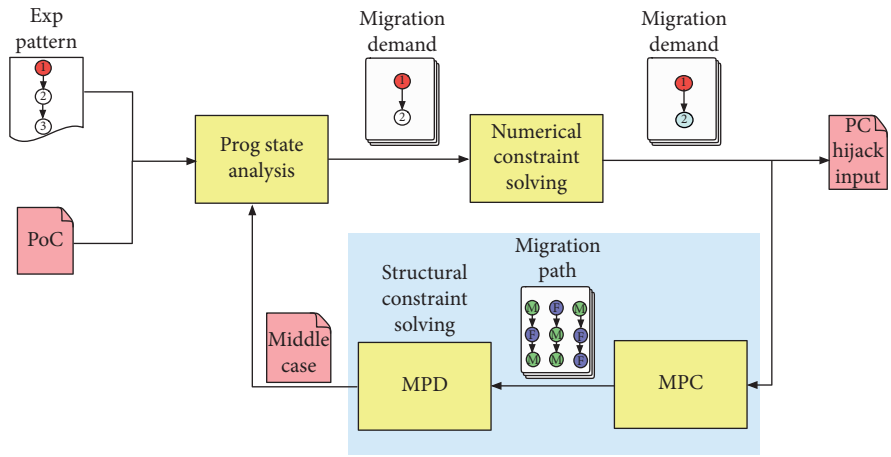

Figure 3: Design of the procedural part for *RELAY*.

4.2.1. MOP Graph. The structural constraint cannot be solved by calculation tools such as constraint solvers [42] and can only be solved by effectively combining basic operations on the heap memory structure. We abstract the typical heap memory structure operations into five basic memory operation primitives: allocation (A), deallocation (F), memory write (W), memory read (R), and memory call (V), which will be explained separately below.

The allocation primitives are built on the granularity of functions, such as *malloc* and *realloc* in *ptmalloc*, where *realloc* can be regarded as an allocation or a deallocation. The deallocation primitive is also built on function granularity, typically *free* in *ptmalloc*. Writing and reading primitives can be completed by an instruction (such as *mov [addr]* and *val*), basic block, or external function (such as read function). In order to reduce the analysis complexity, we define the memory write/read built on basic block granularity. The calling primitive is built on the calling method. For a memory unit with callable attributes, if the target calling method is empty, it means that the data for the memory unit (such as the return address when the function returns) can be automatically transferred to the PC register implicitly; if the target calling method is not empty, the calling primitive is generally triggered by invoking the data in the memory unit in the form of a function pointer (such as invoking *malloc_hook* when executing *malloc*).

Based on the above MOPs, we can formally define the p_{sm} and convert the problem of solving structural constraints into a problem of solving an ordered set of MOPs. The migration path between states refers to an ordered set of several MOPs expressed as $p_{\text{sm}} = \langle o_1, o_2, \ldots, o_N \rangle$, $(o_i \in (A, F, W, R, V))$. In the case of satisfying the numerical constraint ψ_D, after program G executes these operations, the memory state satisfies the structural constraint relationship ψ_S; that is, $\psi_S(G(p_{\text{sm}})) = \text{true}$.

In order to construct the p_{sm}, we need to select a group of MOPs that can satisfy the structural constraints from the program path. We must first establish an understanding of the program path from which to obtain all possible groups of MOPs, so we proposed the concept of MOP graph. The MOP graph represents all of the MOPs that may be traversed during the execution of a program. It represents all possible types and sequences of all MOPs in the form of a graph. As shown in Figure 4(b), each node in the figure represents a MOP, and the directed edge indicates the possible transfer direction of the MOP.

The trigger of a MOP is bound to the program path, and the program path can be reflected by CFG, so we employ CFG to extract the MOP graph. The extraction process is shown in Figure 4(a). First, we utilize a static analysis tool to obtain a complete CFG which includes the basic block address, instruction information, and the jump relationship between the basic blocks. Next, we mark the location of the MOP on the CFG as a new node. For the MOP of function granularity, we mark them in the basic block according to the function invocation address. For the MOP of basic block granularity, we mark the corresponding first address of the basic block. We use the tuple $n = \langle T_o, A_o \rangle$ to represent the node, where T_o indicates the type of MOP (5 types in total, A, F, W, R, V) and A_o represents the marked address.

After marking the new node, we establish a new directed edge according to the connection relationship of the new node in the CFG. Specifically, if there is a connectivity path p_c between the basic blocks where the two new nodes (n_i and n_j) are located and there are no other new nodes in p_c, then we can build a directed edge $e_{ij} = \langle n_i, n_j \rangle$. For example, the basic blocks where n_1, n_2 are located have a connection relationship shown in Figure 4(a), and therefore, a directed edge $e_{12} = \langle n_1, n_2 \rangle$ can be established. The directed edge indicates that the program may perform the o_2 operation after the o_1 operation, so we can append the o_2 operation after the o_1 operation when extracting the MOG.

In particular, for the directed edges $e_{ij} = \langle n_i, n_j \rangle$, if n_i has only one new successor node n_j and any path in CFG from n_i passes through the n_j node, we call $e^*_{ij} = \langle n_i, n_j \rangle$ a strengthened directed edge. For example, since there is a n_3, n_4 connection relationship in Figure 4(a), satisfying the above relationship, a strengthened directed edge $e^*_{34} = \langle n_3, n_4 \rangle$ can be established. The strengthened directed edge means that the program will inevitably invoke the o_4 operation after the o_3 operation, so we must append the o_4 operation after the o_3 operation when extracting the MOG.

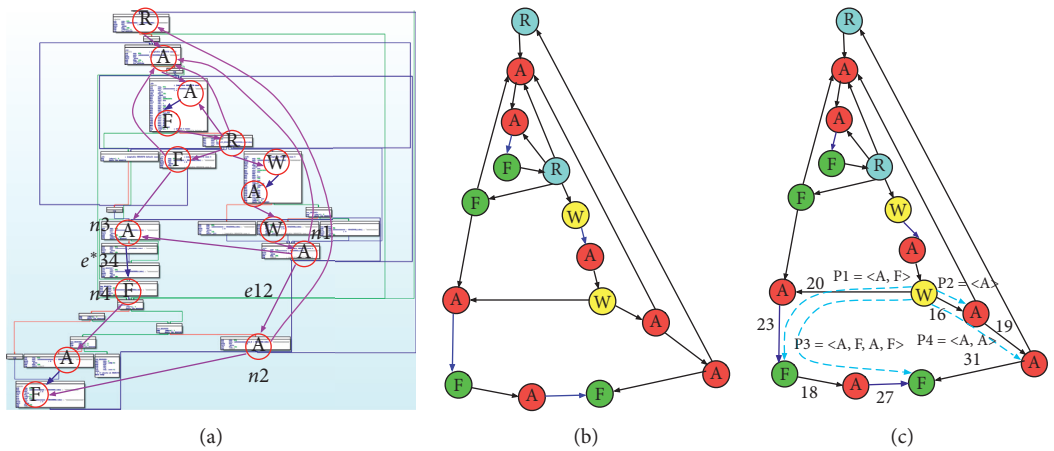

FIGURE 4: Constructing the MOP graph and extracting the MOG from it. The circles and lines represent the nodes and edges of the MOP graph, respectively. The blue line represents the strengthened edge. The dotted line represents the traversal path p_k. The number represents the weight of the edge called driving cost (explained later). (a) Mark nodes and edges in CFG; (b) build MOP graph; (c) extract MOG from MOP graph.

After traversing all of the marked new nodes, we keep the new nodes and the new directed edges, remove the irrelevant basic blocks, and jump relationships, to complete building the MOP graph, as shown in Figure 4(b).

From the MOP graph, we can explore the path with the graph traversal algorithm and extract all possible MOGs from it. The traversal method is as follows: first, we must determine the starting point of the traversal. Since the p_{sm} is a stitching of the paths in the test case, we need to ensure that the selected MOG can be appended to the original operation. This requires us to determine the starting point for the traversal according to the corresponding program state point in the test case. We record the memory operation execution trace $O_{\text{ini}} = \langle o_1, o_2, \ldots, o_N \rangle$, $(o_i \in (A, F, W, R, V))$ of the program in the test case and select the last memory operation o_N. We then find the node n_i in the MOP graph whose address matches with o_N's, which means we find out the current program memory operation in the graph. So, we use this node as the starting point for exploring to ensure that the constructed p_{sm} can be spliced into the current program path. After this, we need to determine the traversal method. Since we need to solve p_{sm} while exploring (explained later), which requires improving the efficiency of exploring shallow paths, we adopt a breadth-first traversal method along the directional edge (consider repeatedly traversing nodes in the ring).

As shown in Figure 4(c), we collect MOPs corresponding to the nodes on each traversed path p_k as a group $O(p_k)$. It should be noted that if a traversed node n_i has a strengthened directed edge $e_{ij}^* = \langle n_i, n_j \rangle$, the node n_i cannot be used as the endpoint of the traversal path p_k. The path p_k must be extended backward to add n_i's successor node n_j to the group $O(p_k)$. Because some memory operations are accompanied by noise, when program triggers these operations, it will inevitably trigger additional memory operations. It is reflected in the MOP graph that these operations have a strengthened edge. We can take the noise problem into consideration by adding special treatment of the strengthened edge, and so the operation group extracted is more reasonable. On the contrary, for general directed edges, there is not such a strong correlation between the front and back operations, so they can be randomly combined without special treatment.

4.2.2. Heap Allocator Model. We have already explained above that, on blindly driving the program, traversing the path space to meet structural constraints is often inefficient. We hope to find a specific path in advance and then drive the program along with it. By reducing the complexity of the path space, we can improve the efficiency of the drive. In order to find the p_{sm}, we first extracted a series of combinations of operations from the MOP graph, and then, we needed to find a group that satisfied the structural constraints. To this end, we established a heap allocator model to infer memory operations, observed the memory state after performing memory operations, and selected an operation group that allows the memory state to satisfy structural constraints.

It should be noted that the model is not only meant to replace the work of the heap allocator, such as managing the heap memory to store data, but only to imitate the characteristics of the heap allocator and analyze the relationship between memory operation and memory state. So we only established the mapping relationship between memory operations and memory state at the logical level and did not actually allocate or deallocate memory at the machine level. In addition, because we only focus on the critical objects associated with structural constraints in the heap memory, there is no need to fully characterize all heap memory.

The essence of the heap allocator is to input the memory operation o and output a memory state $\mathcal{H}_o$, so the key to building a heap allocator model is to build a mapping relationship between the memory operation and the memory state. After the program invokes a memory operation, the memory changes can be expressed as the relationship

$\mathscr{H}_i \longrightarrow^o \mathscr{H}_o$. The output state $\mathscr{H}_o$ is determined by the operation o and input state $\mathscr{H}_i$. Therefore, a binary functional relationship $\mathscr{H}_o = f(o, \mathscr{H}_i)$ can be established, and our goal is to parse this binary function. In order to express binary functions in an enumerable form, we need to employ finite symbolic expressions to represent infinite specific values of memory states and memory operations. To this end, we use the extended representation of the memory object to express the memory state and then use the unified operation symbol to express the memory operation to convert the binary function into an exhaustible segmented form.

Taking *ptmalloc* as an example, we show how to build a heap allocator model and express it as a binary mapping relationship. For convenience, the size represents the aligned parameter of allocation, c_A^* represents the newly allocated chunk, and c_F^* represents the released chunk. $\mathrm{SK}(\mathrm{size}, L)$ indicates that the size is within the range of the linked list$L$. $\mathrm{MinGT}(c_i, L, \mathrm{size}')$ represents that c_i is the smallest of all chunks in the linked list L whose size is larger than size$'$. $\mathrm{Split}(c_i, c_j, c_k)$ represents the split of c_i into c_j and c_k. $\mathrm{Merge}(c_i, c_j)$ indicates the merge of c_j into c_i. $\mathrm{ChBins}(L)$ represents moving chunks into the linked list L from UnsortedBins, and $\mathrm{ChSpl}(c_i)$ represents moving the split chunk $c_i$into UnsortedBins. MinSize represents the lower limit of size for UnsortedBins. First, we list 10 critical expressions for $\mathscr{H}_i$ as conditions for the piecewise function:

$$
\begin{aligned}
&C_1\colon \mathrm{IS}\left(\mathscr{H}_i.L_{F_i}, \langle 0,0,0,c_1,c_2,\ldots,c_N\rangle\right) \wedge \mathrm{SK}\left(\mathrm{size}, \mathscr{H}_i.L_{F_i}\right),\\
&C_2\colon \mathrm{IS}\left(\mathscr{H}_i.L_{S_i}, \langle 0,0,1,c_1,c_2,\ldots,c_N\rangle\right) \wedge \mathrm{SK}\left(\mathrm{size}, \mathscr{H}_i.L_{S_i}\right),\\
&C_3\colon \mathrm{IS}\left(\mathscr{H}_i.U, \langle 1,0,1,c_1,c_2,\ldots,c_N\rangle\right) \wedge \mathrm{EQ}\left(c_1, \mathrm{size}\right),\\
&C_4\colon \mathrm{IS}\left(\mathscr{H}_i.U, \langle 1,0,1,c_1,c_2,\ldots,c_N\rangle\right) \wedge \mathrm{GT}\left(c_1, \mathrm{size}+\mathrm{MinSize}\right),\\
&C_5\colon \mathrm{IS}\left(\mathscr{H}_{i.}L_{L_i}, \langle 1,0,1,c_1,c_2,\ldots,c_N\rangle\right) \wedge \mathrm{SK}\left(\mathrm{size}, \mathscr{H}_i.L_{L_i}\right)\\
&\qquad \wedge \mathrm{MinGT}\left(c_k, \mathscr{H}_i.L_{L_i}, \mathrm{size}\right),\\
&C_6\colon \mathrm{GT}\left(\mathscr{H}_i.T.s_c, c_F^*.s_c\right),\\
&C_7\colon \mathrm{SZ}\left(c_F^*\right), \mathscr{H}_i.L_{F_i},\\
&C_8\colon \mathrm{EQ}\left(c_F^*.P, 0\right) \wedge \mathrm{FD}\left(c_L, c_F\right),\\
&C_9\colon \mathrm{BK}\left(T, c_F\right),\\
&C_{10}\colon \mathrm{EQ}\left(c_h.P, 0\right) \wedge \mathrm{BK}\left(c_H, c_F\right) \wedge \mathrm{BK}\left(c_h, c_H\right).
\end{aligned}
\tag{12}
$$

Then, we can build the mapping relations listed in Table 5. It should be noted that we only built the mapping relationship for the changed heap memory in the structural dimension rather than across the entire area. In order to construct a p_{sm} in a unified form, we count the $W/R/O$ primitives as part of solving the structural constraints. However, strictly speaking, these primitives will not directly bring changes in the heap memory structure, so we have not listed them in Table 5, but only built a numerical mapping in the related memory region for them (e.g., marking the target address of the writing primitive as controllable). In addition, we do not consider the influence of *mmap* [43] and reasonably assume that the allocated memory will not exceed the size of *TopChunk* (avoid triggering system calls to reduce speed).

For example, an allocator model gets input $o = \mathrm{malloc}(16)$, and the current program state is $\mathscr{H}_i.L_{F_i} = \langle 0,0,0,c_1,c_2\rangle$, which satisfies C_1. So according to the mapping relationship of $f(o, \mathscr{H}_i)$, the model will output a new state $\mathscr{H}_o$, where the linked list changes to $L_{F_1} = \langle 0,0,0,c_1\rangle$ and returns an allocated chunk $c_A = c_2$.

If entering a memory operation group $O(p_k) = o_1, o_2, \ldots, o_N$, first of all, we will mutate their parameters. For a memory allocation primitive, we will mutate its size. For memory deallocation primitive, the mutation space of its parameter addr contains the addresses of all the allocated and deallocated chunks in the current state. For the writing primitive, the mutation space of its parameter addr contains the addresses to be controlled in the target state (the definition of target state will be given later), and the mutation space of width is the set [1, 2, 4, 8, 16, 32, 64, 128]. Then, we input these mutated MOPs into the heap allocator model one by one and process them according to the mapping relationship to obtain the final new program state.

With the help of the heap allocator model, after entering the MOG and the initial state of the program, we can infer the new state and match the new state with the target state (this matching method will be introduced later), and we can know whether the current MOG can make the memory state satisfy the structural constraint.

4.2.3. Solving the Migration Path. Our goal is to extract a MOG that satisfies the structural constraints from the program path to build p_{sm}. To achieve this goal, we should complete the following three processes: extract the MOG, solve p_{sm}, and consider the difficulty in driving p_{sm}. The method of extracting MOGs from the MOP graph was introduced earlier. Below, we first introduce the calculation method of the degree of state difference and explain how to find the sections that satisfy the structural constraints from MOGs by the degree of difference. Then, we introduce the concept of driving cost and explain how to use driving cost to reduce the difficulty of driving p_{sm}. Finally, we summarize the whole process for building p_{sm}.

(1) Degree of State Difference. We introduce the concept of state difference degree (d_D) to measure the distance of memory structures between different states. Since there are different data structures in memory regions such as linked lists and the set of allocated chunks, we must define the distance of these typical data structures involved in detail.

The operation for the random access linked list is undirected, and chunks can be inserted and removed from any position on the linked list. Typical random access-linked lists include *ptmalloc*'s *LargeBin* linked list and *UnsortedBin*. For the two random access linked lists A and B, the degree of difference between the structures is defined in the following equation, where $\mathrm{LCS}(A, B)$ refers to the longest common subsequence distance (a special kind of the edit distance [44] which only considers insertion and deletion):

Table 5: Typical mapping relation for a heap allocator.$AFC_2^*C_4^*$

o	$\mathscr{H}_i$	$\mathscr{H}_o$
A	C_1	$\mathscr{H}_o.L_{Fi}$=<0,0,0,c_1,c_2, . . .,c_{N-1}>,$c_A = c_N$
	$\neg C_1 \wedge C_2$	$\mathscr{H}_o.L_{Si}$=<0,0,1,c_2, . . .,c_N>,$c_A = c_1$
	$\neg C_1 \wedge \neg C_2 \wedge C_3$	$ChBins(\mathscr{H}_o.L_L, \mathscr{H}_o.L_S)$, $\mathscr{H}_o.U$=Nul, $c_A = c_1$
	$\neg C_1 \wedge \neg C_2 \wedge \neg C_3 \wedge C_4$	$Split(c_1,c_i,c_j) \wedge c_i.s_c = size$, $c_A = c_i$, $ChBins(\mathscr{H}_o.L_L,.L_S)$, $\mathscr{H}_o.U$=Nul
	$\neg C_1 \wedge \neg C_2 \wedge \neg C_3 \wedge \neg C_4 \wedge C_5$	$Split(c_k,c_i,c_j) \wedge c_i.s_c = size$, $c_A = c_i$,$ChSpl(c_j)$
	$\neg C_1 \wedge \neg C_2 \wedge \neg C_3 \wedge \neg C_4 \wedge \neg C_5 \wedge C_2^*$	$\mathscr{H}_o.L_{Si}$=<0,0,1,c_2, . . .,c_N>,$c_A = c_1$
	$\neg C_1 \wedge \neg C_2 \wedge \neg C_3 \wedge \neg C_4 \wedge \neg C_5 \wedge \neg C_2^* \wedge C_4^*$	$Split(c_1,c_i,c_j) \wedge c_i.s_c = size$, $c_A = c_i$, $ChBins(\mathscr{H}_o.L_L, \mathscr{H}_o.L_S)$, $\mathscr{H}_o.U$=Nul
	$\neg C_1 \wedge \neg C_2 \wedge \neg C_3 \wedge \neg C_4 \wedge \neg C_5 \wedge \neg C_2^* \wedge \neg C_4^* \wedge C_6$	$Split(T,c_i,c_j) \wedge c_i.s_c = size$, $c_A = c_i$,$T.s_c = c_j.s_c$
F	C_7	$\mathscr{H}_o.L_{Fi}$=<0,0,0,c_1,c_2, . . .,c_N,c_F>
	$\neg C_7 \wedge C_8$	$Merge(c_L,c_F)$, $\mathscr{H}_o.U$=<1,0,1,c_1,c_2, . . .,c_N,c_L>
	$\neg C_7 \wedge \neg C_8 \wedge C_9$	$Merge(T,c_F)$
	$\neg C_7 \wedge \neg C_8 \wedge \neg C_9 \wedge C_{10}$	$Merge(c_F,c_H)$, $\mathscr{H}_o.U$=<1,0,1,c_1,c_2, . . .,c_N,c_F>

A, allocation; *F*, deallocation; reallocation can be regarded as one of these. The allocator needs to check C_2^* and C_4^* again after checking the previous 5 conditions.

$$d_{\mathrm{RL}}(A,B) = \frac{1}{\mathrm{LCS}(A,B)+1}. \tag{13}$$

The direction of operation of the one-way access linked list is fixed, so insertion and deletion can only start from one end of the linked list. For example, insertion and deletion are performed in the same direction (FILO) for the *FastBin* list in *ptmalloc* and, however, in different directions (FIFO) for the *SmallBin* list. The definition of the distance between the two types of linked list structures can be expressed in the following equation, where L_A and L_B represent the length of the linked lists of A and B, respectively; for FILO-type linked lists, L_{AB} represents the longest matching length of $B \longrightarrow A$ after A and B are aligned backward in the direction of deletion. For FIFO-type linked lists, L_{AB} represents the length of the part that can completely match the subsequence of A after the first same node of B and A from the insertion direction:

$$d_{\mathrm{SL}}(A,B) = L_A + L_B - 2 * L_{AB}. \tag{14}$$

The degree of difference between unordered sets is defined in the following equation, where diff (B, A) represents the number of elements which are located in B but not in A. For example, assuming that the target state requires 4 memory units to be controllable and only one of the memory units is currently controlled, the degree of difference between the current state and the target state is 3:

$$d_{\mathrm{set}}(A,B) = \mathrm{diff}(B,A). \tag{15}$$

We total the degree of difference between the different data structures involved in the two states to obtain the total d_D between the states.

From the structural constraints, we can acquire the initial state and corresponding target state (we can link them as a state migration demand (d_{sm}) expressed as $d_{\mathrm{sm}} = \mathscr{H}_i \longrightarrow \mathscr{H}_o^*$). By inputting the extracted MOG and the initial state $\mathscr{H}_i$ into the heap allocator model, we can infer the new state $\mathscr{H}_o$ and calculate the d_D between the new state and the target state. If d_D is 0, it indicates to the MOG that it satisfies the structural constraints. For example, there is a structural constraint for IN$(c_2, \mathscr{H}.A)$. First, we use the current program state as the initial state $\mathscr{H}_i$, and the c_2 chunk in the linked list is expressed as $\mathscr{H}_i.L_{F_1} = \langle 0,0,0, c_1, c_2, c_3\rangle$. Next, we ensure the target state $\mathscr{H}_o^*, \mathscr{H}_o^*.A = c_2$. After receiving the operation group $O(p_k) = A, A$, from the heap allocator, the output state is $\mathscr{H}_o$, where the linked list changes to $L_{F1} = \langle 0,0,0,c_1\rangle$ and returns the chunk $c_A = c_2$. The d_D between the new state $\mathscr{H}_o$ and the target state $\mathscr{H}_o^*$ is 0, indicating that we have found a $p_{\mathrm{sm}} = \langle A, A\rangle$.

(2) Path Driving Cost. The complexity of branch constraints for program paths is different, so the difficulty in exploring these branches for a driver is also different. In order to improve the efficiency of exploration, we hope to find those parts with low driving difficulty from the set of p_{sm}. We put forward the concept of driving cost (d_C) to measure the difficulty in driving the p_{sm}. Because it is difficult to directly measure the complexity of path branch constraints by static analysis, we employ a dynamic method of path exploration to evaluate the difficulty in driving the path.

While exploring the path using fuzzer, we record the number of times it hit the directed edges in the MOP graph by hooking function and positioning the basic block. While receiving many test cases, we also have a large number of records on the directed edge e_{ij} in the MOP graph. We count the total number of test cases w_t and divide the total number of hitting (w_t) by the number of hitting the edge (w_{ij}). Then, we mark the weight of the directed edge e_{ij} as w_t/w_{ij}(if $w_{ij}$is 0 then it is marked as infinity), and a MOP graph with weights is established, which can reflect the difficulty in driving different paths. If the weight of the edge is higher, it means that it is more difficult to drive along the corresponding path. By totaling the weight of the edges included in the p_{sm} on the graph, we can evaluate the difficulty in driving p_{sm}.

So far, we can formally define the driving cost of $p_{\mathrm{sm}} = \langle o_1, o_2, \cdots, o_N\rangle$, $(o_i \in (A, F, R, W, V))$:

$$d_C = \sum_{i=1}^{N-1} \frac{w_t}{w_{i(i+1)}}. \tag{16}$$

$w_{i(i+1)}$ represents the weight of the directed edge $e_{i(i+1)} = \langle n_i, n_{i+1}\rangle$, and n_i, n_{i+1} corresponds to the o_i, o_{i+1}, respectively. As shown in Figure 4(c), the driving cost of the

path $p_3 = \langle A, F, A, F \rangle$ is 88. It is important to note if the p_{sm} includes an edge with an infinite weight, it means that such a path cannot be triggered during the running of the program, and so this path will be removed from group P_{sm}.

(3) MPC Workflow. Finally, we sort out the entire process of constructing p_{sm}, and the entire MPC workflow is shown in Figure 5. For a structural constraint, we first construct a MOP graph for the program and extract a group of memory operations $O(p_k)$ from the graph. Then, we mutate their parameters and input operations into the heap allocator model for inference and observe whether the structural constraint requirements are satisfied. If satisfying these structural constraints, the group of operations will be added to the set of preliminary migration paths. In this process, we adopt a synchronous strategy of inferring while extracting to improve efficiency, and we set a time threshold for mutation. For each group of operations extracted, we continuously perform mutations for them and then infer the results until the threshold is exceeded. Then, we extract the next group of operations and repeat the process. For the preliminary migration path set P_{sm}, we select the $k$paths with the smallest driving cost, arrange them in the optimized set P^*_{sm} in order, and pass the set to the MPD.

4.3. Migration Path Driving (MPD). The problem of driving the p_{sm} is essentially a reachability problem, and our purpose is to generate an input that can drive the program to reach the target point (the call location of target memory operation). The problem of the reachability of the target point can be decomposed into two problems: control-flow arrival (CFA) and data flow arrival (DFA). For example, for the F operation in the p_{sm}, the generated test cases not only need to trigger deallocation functions but also need to accurately set the function parameters as the chunk address is expected to be deallocated.

Currently, there are many fuzzing tools that have the ability to drive program reaching specific locations [45–47]. For example, *AFLGo* [48] uses a guided gray-box fuzz to generate test cases that can reach the target program point. We have made modifications on the basis of *AFLGo* to generate test cases satisfying both CFA and DFA.

The modification method is as follows: by constantly optimizing the test cases with higher path weights (ones which include a target memory operation), we output test cases that can reach the target program point and achieve control-flow arrival. Taking test cases with CFA as seeds, we continuously approach test cases with higher data weights (ones associated with target parameters) and finally output target test cases.

5. Implementation and Evaluation

We implemented the prototype of *Relay* on top of the framework mentioned previously. On the selective symbolic execution engine *S2E* [49], we defined four exploit patterns including *FastBin*, *Unlink*, *Hof*, and *Overlap* and implemented the program state analysis module. The function of solving numerical constraints and arranging shellcode is also

Input: Program P, CurrentState s_i, NewState s_o, MopPath p_k, MopGroup $O(p_k)$, StructuralConstraint ψ_S
Output: OptimizedMigrationPath P^*_{sm}
1 *MopGraph* ← BuildMopGraph(P)
2 **for** $p_k \in$ *TraverseMopGraph*(s_i, *MopGraph*) **do**
3 $O(p_k)$ ← ExtractMopGraph(p_k)
4 **for** $i \in$ *MutateThreshold* **do**
5 $O'(p_k)$ ← *Mutate*($O(p_k)$)
6 s_o ← AllocateModel(s_i,$O'(p_k)$)
7 d_D ← CalcDifferDegree(s_o, ψ_S)
8 **if** d_D ← 0 **then**
9 P_{sm} ← append($O'(p_k)$)
10 **end**
11 **end**
12 **end**
13 P^*_{sm} ← SelectMinDriveCost(p_{sm})

Figure 5: *MPC* workflow.

built on top of *S2E*. MOP construction and MOG extraction are implemented on top of *IDA Python* [50]. Also, we implemented a heap allocator model by building function mapping relationships in *Python*. And the MPD module was built on top of the fuzzing engine *AFL* [51]. *RELAY* contains a total of 8.5 K lines of C/C++ and 4.6 K lines of *Python*. It should be noted that although we only implement four exploit patterns currently, *RELAY* is an extensible framework that will include more types of vulnerabilities and exploit methods in the future. The experimental part mainly explores the following two problems:

(i) RQ1 : does *RELAY* has the ability to evaluate exploitability for metadata corruption vulnerabilities?

(ii) RQ2 : what factors mainly affect the efficiency of *RELAY*?

5.1. Benchmark Selection. Affected by program scale, running environment, and interaction mode, the testing process for real software is cumbersome and easily interfered by irrelevant factors. The *CTF* and *RHG* programs are specifically designed to test the ability of humans and machines to deal with vulnerabilities. They can be regarded as a set of programs that reduce the scale of real software and irrelevant factors but retain their vulnerability characteristics. They focus on human expertise for exploitation and usually require high-level skill to solve problems. We have selected 17 *CTF* programs and 3 *RHG* programs from the recent public events. The reason for choosing more *CTF* programs is that *CTF* is mainly used to evaluate the application ability of high-level knowledge than the machine's computation capability and can better test whether our system can effectively deal with the vulnerability in combination with artificial knowledge.

The exploitability evaluation for these target programs covers four common exploitation methods (*FastBin attack*, *Unlink*, *Hof*, and *chunk overlap&FastBin attack*), to verify whether our system can use the corresponding four exploit patterns to complete assessing the vulnerability. Each target

program represents some type of programs bounded to the specific exploitation method and the capability in dealing with one of the target programs indicating the capability in evaluating this type of program. Certainly, the capability of our system is not limited to these 20 selected programs; for instance, our system also completed the exploitation for two heap vulnerabilities on the spot in the open event of *DEFCON China* [52].

In a word, the target programs we selected have the following features:

(i) The program comes from the public event, and it is believed to reflect the vulnerability feature of real software.

(ii) The program does not contain source code and debug symbols, which can fully restore the exploitability evaluation in the real environment.

(iii) Each program contains metadata corruption vulnerabilities.

(iv) Each program represents some types of program bounded to the specific exploitation method.

5.2. Experimental Setup. All experiments were run on a Linux Ubuntu Server 18.04 LTS AMD64, equipped with an Intel(R) Core(TM) i9-9980XE CPU@3.00GHz and 64 GB RAM. Our experiments enabled *DEP* [53], but disabled *ASLR* [54], since *RELAY* does not support automatically bypassing *ASLR*.

5.3. Exploits by RELAY (RQ1). In this section, we first display the experimental results and then analyze the specific cases and demonstrate the strengths of *RELAY* compared to other AEG solutions.

5.3.1. Overall Results. As shown in Table 6, we tested a total of 20 programs, of which 16 successfully generated the exploits and 4 failed. The table details the program name, event name, and vulnerability type, respectively. In addition, it shows the exploit pattern corresponding to the initial state of the program detected by the program state analysis module after inputting the PoC (the correct exploit pattern is marked with an $*$ after manual verification), and it also records the total number of nodes N_m contained in the MOP graph.

We also recorded the total path length L_p of the migration path corresponding to the final exploit (by adding the number of memory operations included in each migration round) and the total driving cost D_c (by adding the driving cost corresponding to each migration round). For equality, we use the same number of test cases w_t generated by fuzzing to solve driving costs for all programs. In addition, we recorded the elapsed time T_m from inputting PoC to outputting EXP.

Finally, we compared it to other state-of-the-art AEG solutions. As far as we know, the current AEG solution for user-level application vulnerabilities is the only *Revery*. We got the experimental data from the *Revery* paper (programs without data are marked as "–"). To draw a more objective compare conclusion, we also chose another open-source AEG solution, *Rex* [55], as an experimental comparison. Developed by the *shellphish* team, *Rex* won the first place in the *Cyber Grand Challenge* (CGC) [56] contest although it is not specifically designed to deal with heap vulnerabilities.

It can be seen that the time to complete exploitation is ranging from 10 to 50 minutes. This suggests that *RELAY* is an efficient AEG tool, and as we know, other AEG tools that deal with heap vulnerabilities, such as *Revery*, often need over 50 minutes, or even hundreds of minutes to work out. The *RELAY*'s speed is even accepted by the professional security analyst (it usually takes tens of minutes to complete the exploitation of the heap vulnerability manually), so *RELAY* can be put into practice to deal with heap vulnerability for replacing the manual evaluation to some extent.

We can also see that both N_m and L_p have an approximately positive correlation trend with the elapsed time. This is because N_m is positively correlated to the program scale, and there is a positive correlation trend between L_p and the complexity of driving migration path to some extent. And D_c has a stronger relationship with T_m, and the two are generally positively correlated, which proves that we can improve efficiency for exploitation by choosing migration paths with a low driving cost (detailed in Section 5.2).

5.3.2. Case Study. In this section, we investigated these programs in detail and analyzed why our solution was successful or not.

(1). Successful Cases. RELAY successfully generated exploits for 16 programs, covering multiple types of vulnerabilities and the four defined exploit patterns, which shows *RELAY* has the ability to process metadata corruption vulnerabilities. In contrast to *Rex*, it cannot generate an exploit for any programs listed. *Rex* only works under the circumstance that the control flow has been hijacked, but the above programs did not enter the state of control-flow hijacking after running PoC. So we conclude that *Rex* does not have the capability in dealing with common heap vulnerabilities. Likewise, *Revery* cannot solve any listed program (for programs without data, we reasonably conclude that *Revery* fails on them). Below we employ specific programs to analyze the reasons why *RELAY* succeeds and *Revery* fails.

First, *RELAY* can solve numerical constraints on specific locations and time. For example, *bcloud* stipulates that the chunk can only be overflowed by four bytes. It is necessary to modify the size of *TopChunk* to a large specific value at the beginning. For *simple note*, it needs to point *fd* and *bk* to a specific position in the heap table to construct an AAW. By analyzing the numerical constraints in the exploit pattern, *RELAY* can modify the target value before the heap block structure changes. However, *Revery* cannot perceive these constraint relationships. And if we fail to complete the memory value modification within an effective time period, such as modifying *fd* and *bk* before the heap block merges or modifying the size before splitting *TopChunk*, it will make the memory allocation and deallocation meaningless or even prevent exploitation.

Table 6: Overall results of *RELAY*.

	Name	CTF	Vul type	Exp. pattern	N_m	L_p	D_c	T_m/s	Revery
Successful cases	BRHG-13	BRHG	UAF	fast_bin_exp	6	5	38	1283	—
	pwn14	DEFCON China&BCTF	Heap overflow	hof_exp	8	4	32	927	—
	pwn40	DEFCON China&BCTF	Heap overflow	unlink_exp	5	5	43	1194	—
	note2	ZCTF 2016	Heap overflow	unlink_exp	18	10	83	1427	✘
	note3	ZCTF 2016	Heap overflow	unlink_exp	19	12	74	1483	✘
	fb	AliCTF 2016	Heap overflow	unlink_exp	32	13	96	1638	✘
	Stkof	HITCON 2014	Heap overflow	unlink_exp	27	12	117	1829	✘
	Simple note	Tokyo westerns 2017	Off-by-one	unlink_exp	21	10	99	1595	✘
	Search engine	2015 9447 CTF	Double free	fast_bin_exp	21	13	106	1762	—
	Badint	Defcon qualifier 2017	Heap overflow	fast_bin_exp	31	13	138	2053	—
	Weqpon	Delta Ctf 2019	UAF	fast_bin_exp	24	12	82	1518	—
	Babyheap	0ctf 2017	Heap overflow	overlap_exp	36	16	175	2431	—
	Pwnme	NCTF2019	Off-by-one	overlap_exp	25	15	152	2218	—
	Bcloud	2016 BCTF	Heap overflow	hof_exp	13	9	75	1364	—
	Gyctf2020_force	BUUCTF	Heap overflow	hof_exp	16	8	86	1487	—
	Bamboobox	Hitcon training	Heap overflow	hof_exp	12	8	69	1329	—
Failed cases	Bookstore	2015 hacklu	Heap overflow	unlink_exp	25	10	126	—	—
	Babyheap	0ctf2018	Off-by-one	overlap_exp	21	13	153	—	—
	Wheelofrobots	2017 insomni'hack	Off-by-one	overlap_exp	18	11	141	—	—
	Babypwn	N1ctf2019	Double free	fastbin_exp	23	11	148	—	—

Second, *RELAY* can convert structural constraints problems to solve the memory operation sequence and transform the state coverage problem into a specific path exploration problem, which can greatly reduce the complexity of the state space and the difficulty in path exploration. For *pwnme*, it needs to take out the fake chunk from the *FastBin* chain, and then, *RELAY* can calculate the shortest path A-A-A to complete the task. However, *Revery* directly uses fuzzing to search for the target state, which may repeatedly trigger another high-frequency path A-F-A-F. On this path, we cannot quickly take out the target chunk, which reduces the efficiency of promoting heap layout, and may even cause the program to exit after reaching the allowed maximum number of allocations.

(2). Failed Cases. RELAY currently supports four modes, does not combine the global analysis of the program sufficiently, and arranges the corresponding fixed mode according to the current program state. The contradiction between the complexity of the program path and the fixed mode brings confusion to the exploitation. If there are special restrictions on the program paths, the migration path obtained according to the exploit pattern may not be driven. For example, *RELAY* built exploit modes for 4 programs in the table, but these programs cannot be driven in a real environment.

For example, for *bookstore*, the size of the allocation is fixed and does not belong to the *FastBin* range. And the overflowed content is not controlled, which makes the conventional *Hof/Unlink/FastBin attack* impossible. The solution is to destroy the size metadata and cause a chunk overlap; then, the content of the format string can be modified, and AAW can be triggered by the format string vulnerability. For *babyheap*, the size of allocation cannot exceed 0×58, and the callable fake chunk cannot be directly constructed on the *FastBin* chain. But the *TopChunk* pointer at *Main_Arena* [57] can be modified with *FastBin attack* to cause an arbitrary address allocation. For *wheelofrobots*, the callable fake chunk also cannot be constructed directly. Attackers can only launch *chunk overlap&FastBin attack* to modify one control variable to achieve heap overflow of any length and then employ *Unlink* to achieve AAW. Similarly, for *babypwn*, attackers also need to modify a variable that controls the allowed number of allocations with *FastBin attack*, and then the *malloc_hook* can be modified through *FastBin attack* again.

It can be seen that if there are strict restrictions on the program path, a single exploit pattern can no longer fully guide the exploitation. And more flexible combinations of multiple vulnerabilities and multiple patterns are required to bypass the restricted path and reactivate the exploitation. In future work, we will consider adding some ML (machine learning) methods to build the overall perception of the program paths and explore ways of organizing multiple vulnerabilities and exploit patterns to approach the target heap layout.

5.4. Efficiency of RELAY (RQ2). Previously, we analyzed the effectiveness of *RELAY* and we assess its efficiency in this section. The working time of *RELAY* mainly comes from MPC and MPD modules, so we conducted an evaluation for the performance of these two modules and analyzed the factors that may have affected their performance.

First, we evaluated the impact of the introduction of driving cost on MPC, MPD, and overall speed. We selected 13 *CTF* programs and recorded the working time of the MPC and MPD modules under the two conditions of firstly using the driving cost (C_1) and then of not using it (C_2). For C_1, the MPC module introduced the time consumption of calculating the driving cost (mainly from the path exploration of fuzzing tools), and the MPD module attempts to drive the migration path of the optimization set P^*_{sm} in

ascending order for driving cost. These experimental results are reflected on the right bar of Figure 6 for each program. For C_2, the MPC module does not need to calculate the driving cost, and the MPD module attempts to drive the migration path in the set P_{sm} with a random selection strategy. The results of these experiments are recorded on the left bar of Figure 6 for each program.

Figure 6 shows the experimental results. From the horizontal comparison, although the introduction of driving cost increases the working time of the MPC module, the total exploitation time is reduced. During the experiment of C_1, we calculated the standard deviation σ for the weight of the MOP graph edges for each program. Then, we further found that, for a program with more uneven weight distribution on the edges in the MOP graph (i.e., with a larger σ), there is a more obvious reduction in the total exploitation time from C_2 to C_1. Therefore, we infer that the introduction of driving cost can effectively improve the efficiency of the system, and this improvement is obvious when processing programs with an uneven distribution of complexity for branch constraints.

In addition, in terms of longitudinal comparison, no matter under C_1 or C_2 conditions, MPC has a small working time, and there is no large fluctuation in the time for different programs. So we infer that the efficiency of the MPC is only slightly affected by the program logic. On the contrary, MPD takes a lot of time to drive the migration path. We calculated the average driving cost μ for the migration paths during the experiment of C_1. It can be clearly seen that the higher the driving cost, the more time the MPD takes, which indicate that the performance of the MPD is greatly affected by the program logic. In conclusion, we can infer that the MPC is a low-consumption and stable module. Below, we continue to discuss the performance improvements brought by the MPC to the system.

In the process of promoting the heap layout, the largest difference between *RELAY* and other tools such as *Revery* is that *RELAY* adds a guide of the migration path for the heap layout driver (i.e., adding the MPC module). The key for *RELAY* to driving heap layout efficiently is that the MPC can extract migration paths that satisfy structural constraints from the huge path space. Since we lack the experimental data of other tools, we expect to perform a self-contrast experiment on the MPC module to observe the reduction effect of the program path space after the MPC module works so that we can evaluate the performance improvement of *RELAY* compared to other tools.

The path space can be expressed as a set of tuples p_i, d_i, where path $p_i = \langle o_1, o_2, \ldots, o_N \rangle$, $(o_i \in (A, F, R, W, V))$ and d_i represents the driving cost of path p_i. For the two path spaces and G_2, the efficiency ratio of the complete exploration can be expressed as follows (N, M represent the total number of paths for G_1 and G_2, respectively):

$$\eta = \frac{\sum_{i=1}^{M} d_i}{\sum_{j=1}^{N} d_j}. \tag{17}$$

If assuming that the driving costs of all program paths are the same, this expression can be simplified to $\eta = M/N$, which indicates that the efficiency of exploring path space is inversely proportional to the number of paths in path space.

Based on this assumption, we conducted a separate test on the MPC module. For each program, within a 2-minute limit, we recorded the number of program paths tried by MPC (denoted byg_1 and shown as the whole bars in Figure 7) and the number of migration paths that can satisfy the structural constraints (denoted by g_2 and shown as blue segments in Figure 7). We can suppose that, for those tools without the MPC function, the path space to be explored is G_1 (g_1 for size); then, the path space of the same program to be explored by *RELAY* with MPC function is G_2 ($g_2$for size), so the efficiency ratio of *RELAY* to other tools is rough $\eta = g_1/g_2$.

As shown in Figure 7, it can be seen that, after using MPC, the program path space is roughly compressed to less than 1/5. The efficiency of exploring the path space is improved accordingly, which means we can promote the heap layout more effectively. We accept this conclusion under the assumption that all driving costs of program paths are the same. In fact, MPC can build the path space G_2^* composed of migration paths with low driving costs, and according to the previous formula, the efficiency of exploring space G_2^* will be higher than G_2.

6. Related Work

As described above, our work focuses on assessing exploitability for metadata corruption vulnerabilities in applications. It should be noted that *RELAY* is the first system that addresses AEG problems aimed at metadata corruption vulnerability characteristics in user-level applications. As a result, the works most relevant to ours is heap-based AEG solutions. In the following, we describe the existing works of this type and discuss their limitations.

In order to find exploit primitives for metadata corruption in the allocator, Repel et al. [11] described the first method of AEG for heap overflow. They connect the driver to the target allocator and then use summary execution to discover exploit primitives. To explore weaknesses in the heap allocator, Eckert et al. [29] proposed a system *HeapHopper* for discovering primitives in the allocator and building an exploit in the context of a driver connected by an allocator. In order to solve the derivative problem of heap exploits, Wang et al. [24] described *Revery*, a system that employs a layout-oriented fuzzer and control-flow stitching technique to discover the exploitable state in the divergent path instead of the crash path. Their approach can generate some heap exploits, but only in the case where their fuzzer randomly reaches the required heap layout. To facilitate the exploitation of kernel UAF vulnerabilities, Wu et al. [58] constructed *FUZE*, a system employing kernel fuzzing and symbolic execution to identify useful system calls for kernel UAF exploitation, and utilize dynamic tracking and an off-the-shelf constraint' solver to guide exploitation. In order to promote the exploitation of kernel vulnerabilities, Chen et al. [27] proposed *SLAKE*, which utilizes static and dynamic analysis techniques to identify kernel objects and useful system calls, and then modeled common exploitation

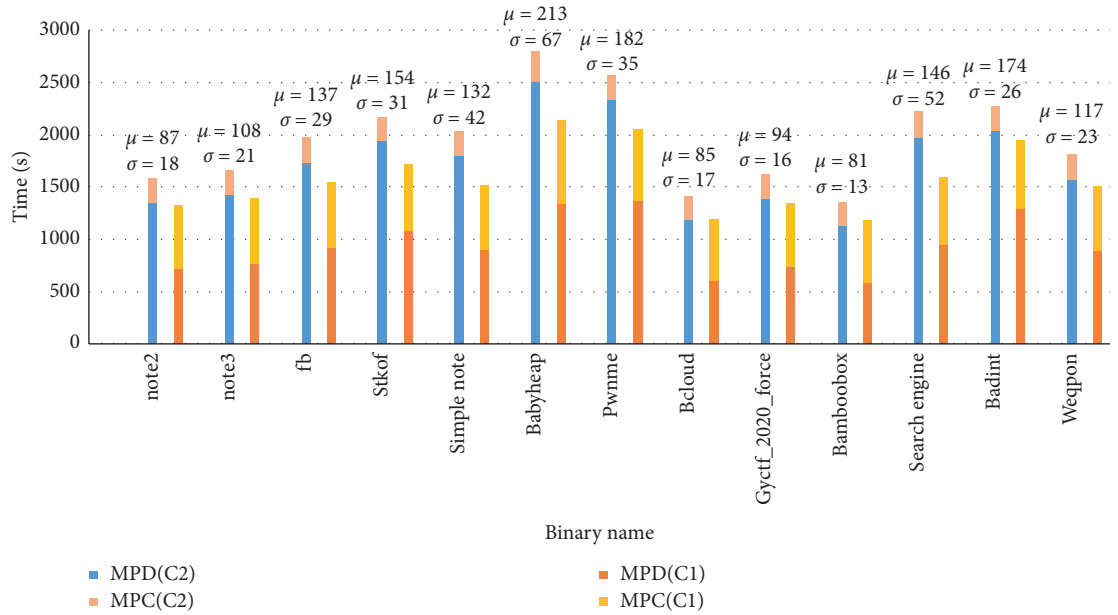

Figure 6: Working time for MPC and MPD under different conditions.

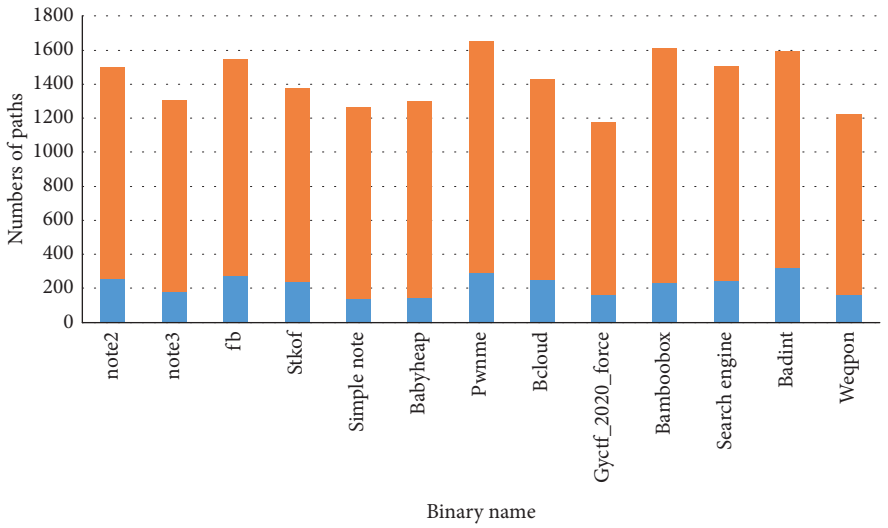

Figure 7: Performance of MPC and efficiency improvement for *RELAY*.

methods to manipulate the slab layout. In order to connect an exploitable state to trigger the execution of a code-reuse payload, Wu et al. [59] constructed *Kepler*, which accepts an input including control-flow hijacking primitive and bootstraps any kernel ROP payload by symbolically stitching an exploitation chain with corresponding gadgets. In order to automatically generate exploits for heap overflow vulnerability in the PHP interpreter, Heelan et al. [28, 60] employed a pseudorandom black-box search method to manipulate heap layout and described an automatic solution *Gollum*, which contains a number of novel ideas, including the mining of tests for code fragments that provide primitives, lazy resolution of heap layouts, a genetic algorithm for heap layout manipulation, and a completely gray-box approach to automatically generate exploits.

7. Conclusion and Future Work

Given that the organization of heap metadata is usually complicated, it is vulnerable to variant forms of heap-related attacks. Despite its importance, fewer efforts have been made to analyze whether heap metadata could be corrupted and exploited by cyberattackers. To improve software quality and ensure cybersecurity, we proposed *RELAY*, a software testing framework to assess the exploitability of metadata corruption vulnerabilities, by simulating human exploitation behavior. Experiments suggest that *RELAY* can effectively evaluate hazardous degree for programs containing metadata corruption vulnerabilities and works more efficiently compared to other state-of-the-art automated tools. However, *RELAY* does not work properly in some cases since the

contradiction between the complexity of the program paths and the unicity of the exploit patterns confuses the evaluation (detailed in the Failed Cases in Section 5.3.2). In future work, we will try to add the ML method to help *RELAY* build the overall concept of program paths and explore ways of organizing multiple vulnerabilities and exploit patterns, to enhance the capability of *RELAY*. Besides, we find that the current discussion on the security issues in Internet of Things (IoT) systems is very intense, and we can also see that some excellent research results [61–65] have been published. In the course of studying the above work, we have found that there are some similarities with our research direction in this field, such as the path exploring and constructing method. Therefore, in future work, we will consider integrating the method of exploitability evaluation for vulnerabilities with the analysis method of IoT security issues.

Data Availability

All data included in this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant no. 61702540) and the Hunan Provincial Natural Science Foundation of China (Grant no. 2018JJ3615).

References

[1] C. Luo, Bo Wang, K. Huang, and Y. Lou, "Study on software vulnerability charac- teristics and its identification method," *Mathematical Problems in Engineering*, vol. 2020, 2020.

[2] G. Spanos and L. Angelis, "A multi-target approach to estimate software vulnerability characteristics and severity scores," *Journal of Systems & Software*, vol. 2018, 2018.

[3] V. V. D. Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, "Memory errors: the past, the present, and the future," 2012.

[4] B. Padmanabhuni and H. B. K. Tan, "Defending against buffer-overflow vulnerabilities," *Computer*, vol. 44, no. 11, pp. 53–60, 2011.

[5] T. W. L. Szekeres and M. Payer, "Sok: eternal war in memory," 2013.

[6] P. K. Kudjo, J. Chen, M. Solomon, R. Amankwah, and C. Kudjo, "The effect of bellwether analysis on software vulnerability severity prediction models," *Software Quality Control*, vol. 1, pp. 1–34, 2020.

[7] M. Mouzarani, B. Sadeghiyan, and M. Zolfaghari, "Smart fuzzing method for detecting stack-based buffer overflow in binary codes," *Iet Software*, vol. 10, no. 4, pp. 96–107, 2016.

[8] J. Park, Y. Choo, and J. Lee, "A hybrid vulnerability analysis tool using a risk evaluation technique," *Wireless Personal Communications*, vol. 10, 2018.

[9] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: state of the art," *IEEE Transactions on Reliability*, vol. 10, 2018.

[10] Shellphish.how2heap, 2020, https://github.com/shellphish/how2heap.

[11] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, vol. 25–35, New York, NY, USA, 2017.

[12] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and effiiicently protecting the heap," *Computer Architecture News*, vol. 10, 2006.

[13] D. Tian, X. Li, M. Chen, and C. Hu, "iCruiser: an improved approach for concurrent heap buffer overflow monitoring," *IEICE Transactions on Information and Systems*, vol. 97, no. 3, pp. 601–605, 2014.

[14] glibc Libc, 2020, https://www.gnu.org/software/libc/libc.html.

[15] S. Silvestro, H. Liu, C. Corey, Z. Lin, and T. Liu, "Freeguard: a faster secure heap allocator," 2017.

[16] A. Brahmakshatriya, P. Kedia, D. P. Mckee et al., "An instrumenting compiler for enforcing confiifidentiality in low-level code," 2017.

[17] Institute of Computer Sciencetechnology, Peking University, Beijing, China, Beijing Key Laboratory of Internet Security Technology, and Beiji, "Using type analysis in compiler to mitigate integeroverflflow-to-buffer-overflflow threat," *Journal of Computer Security*, vol. 10, 2016.

[18] Shellphish. how2heap-fifix for the new check, 2020, https://github.com/shellphish/how2heap/compare/58ae...d1ce.

[19] R. Jukka, R. Sampsa, H. Sami, and L. N. Ville, "A case study on software vulnerability coordination," *Information & Software Technology*, vol. 10, 2018.

[20] P. Poosankam, D. X. Song, D. Brumley, and J. Zheng., "Automatic patch-based exploit generation is possible: techniques and implications," in *Proceedings of the IEEE Symposium on Security and Privacy*, New York, NY, USA, 2008.

[21] S. Heelan, *Automatic Generation of Control Flflow Hijacking Exploits for Software Vulnerabilities*, University of Oxford, Oxford, UK, 2009, https://www.cprover.org/dissertations/thesis-Heelan.pdf.

[22] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: automatic exploit generation," in *Proceedings of the Network and Distributed System Security Symposium*, New York, NY, USA, 2011.

[23] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy SP'12*, IEEE Computer Society, Washington, DC, USA, 2012.

[24] Y. Wang, C. Zhang, X. Xiang et al., "Revery: from proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2018.

[25] H. Liang, Y. Cai, H. Hu, P. Su, and D. Feng, "Automatically assessing crashes from heap overflflows," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, 2017.

[26] Y. Shoshitaishvili, A. Bianchi, K. Borgolte et al., "Mechanical phish: resilient autonomous hacking," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 12–22, 2018.

[27] 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*, pp. 1707–1722, New York, NY, USA, 2019.

[28] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *Proceedings of the*

27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 763–779, New York, NY, USA, 2018.
[29] M. Eckert, A. Bianchi, R. Wang, S. Yan, C. Kruegel, and G. Vigna, "Heaphopper: bringing bounded model checking to heap implementation security," in *Proceedings of the 27th {USENIX} Security Symposium ({USENIX} Security 18)*, New York, NY, USA, 2018.
[30] Unlink Exploit, 2020.
[31] House of force Exploit, 2020.
[32] Glibc Malloc-Pthreads Malloc, 2020.
[33] Fastbin Attack, 2020, https://github.com/ylcangel/exploit/tree/master/heap/fastbin_attackl.
[34] Chunk Overlap and FastBin Attack, 2020, https://ctf-wiki.github.io/ctf-wiki/pwn/linux/glibc-heap/chunk_extend_overlapping/.
[35] CWE-193: Off-By-One Error, 2020.
[36] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.
[37] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the IEEE Symposium on Security and Privacy*, New York, NY, USA, 2010.
[38] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.
[39] C. Lemieux and K. Sen, "Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference On Automated Software Engineering*, pp. 475–485, New York, NY, USA, 2018.
[40] S. K. Huang, M. H. Huang, P. Y. Huang, W. L. Chung, and W. M. Leong, "Crax: software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations," in *IEEE Sixth International Conference on Software Security & Reliability*, New York, NY, USA, 2012.
[41] C. S. Pasareanu, R. Kersten, L. Kasper, and Q. S. Phan, "Symbolic execution and recent applications to worst-case execution, load testing and security analysis," *Advances in Computers*, vol. 22, 2018.
[42] Tools of Constraint Solver, 2020.
[43] "Mmap. munmap, map or unmap fifiles or devices into memory," 2020.
[44] "Edit distance. Measure degree of difference between character string," 2020.
[45] W. You, P. Zong, K. Chen, X. F. Wang, and B. Liang, "Semfuzz: semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the Acm Sigsac Conference*, London, UK, 2017.
[46] Y. Li, S. Ji, C. Lv et al., "Vfuzz: vulnerability-oriented evolutionary fuzzing," 2019.
[47] W. You, X. Wang, S. Ma, J. Huang, and X. Zhang, "Profuzzer: on-the-flfly input type probing for better zero-day vulnerability discovery," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, London, UK, 2019.
[48] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, New York, NY, USA, 2017.
[49] V. Chipounov, V. Kuznetsov, and C. George, "S2e: a platform for in-vivo multipath analysis of software systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011.
[50] IDA Disassembler and Debugger, 2020.
[51] AFL: American Fuzzy Lopa, 2020.
[52] DEFCON CHINA, 2020.
[53] S. Andersen and A. Vincent, *Changes to Functionality in Microsoft Windows Xp Service Pack 2: Part 3: Memory Protection Technologies*, Microsoft TechNet, 2004.
[54] PaX Team, "Address space layout randomization (aslr)," 2003.
[55] "Rex-shellphish's automated exploitation engine," 2020.
[56] DARPA, "Cyber grand challenge," 2020.
[57] Main Arena, "Root of all managed heap memory," 2020.
[58] 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 *Proceedings of the 27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 781–797, New York, NY, USA, 2018.
[59] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: facilitating control-flflow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1187–1204, New York, NY, USA, 2019.
[60] S. Heelan, T. Melham, and D. Kroening, "Gollum: modular and greybox exploit generation for heap overflflows in interpreters," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1689–1706, New York, NY, USA, 2019.
[61] D. M. Mena, I. Papapanagiotou, and B. Yang, "Internet of things: survey on security," *Information Security Journal: A Global Perspective*, vol. 27, no. 3, pp. 162–182, 2017.
[62] B. S. Ahmed, M. Bures, K. Frajtak, and T. Cerny, "Aspects of quality in internet of things (IoT) solutions: a systematic mapping study," *IEEE Access*, vol. 7, pp. 13758–13780, 2017.
[63] A. Mosenia and N. K. Jha, "A comprehensive study of security of internet-of-things," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 586–602, 2017.
[64] M. Bures, B. S. Ahmed, and T. Cerny, "Internet of things: current challenges in the quality assurance and testing methods," in *Proceedings of the 9th Icatse International Conference on Information Science & Applications*, New York, NY, USA, 2018.
[65] A. Mosenia, "Addressing security and privacy challenges in internet of things," 2018.