AutoPwn: Artifact-Assisted Heap Exploit Generation for CTF PWN Competitions

Dandan Xu, Kai Chen, *Member, IEEE*, Miaoqian Lin, Chaoyang Lin, and XiaoFeng Wang, *Fellow, IEEE*

***Abstract*— Capture-the-flag (CTF) competitions have become highly successful in security education, and *heap corruption* is considered one of the most difficult and rewarding challenges due to its complexity and real-world impact. However, developing a heap exploit is a challenging task that often requires significant human involvement to manipulate memory layouts and bypass security checks. To facilitate the exploitation of heap corruption, existing solutions develop automated systems that rely on manually crafted patterns to generate exploits. Such manual patterns tend to be specific, which limits their flexibility to cope with the evolving exploit techniques. To address this limitation, we explore the problem of the automatic summarization of exploit patterns. We leverage an observation that public attack artifacts provide key insights into heap exploits. Based upon this observation, we develop *AutoPwn*, the first *artifact-assisted AEG* system that automatically summarizes exploit patterns from artifacts of known heap exploits and uses them to guide the exploitation of new programs. Considering the diversity of programs and exploits, we propose to use a novel Exploitation State Machine (ESM), with generic states and transitions to model the exploit patterns, and then efficiently construct it through combining the dynamic monitoring of exploits and the semantic analysis of their text descriptions. We implement a prototype of *AutoPwn* and evaluate it on 96 testing CTF binaries. The results show that *AutoPwn* produces 22 successful exploits and 13 partial exploits, preliminarily demonstrating its efficacy.**

***Index Terms*— Vulnerability, heap exploitation, vulnerability analysis, automatic exploit generation, symbolic execution, state machine, capture-the-flag competition.**

I. Introduction

CTF competitions model real-world security issues as a set of challenges, requiring participants to exploit simulated systems and recover hidden flags. These challenges encompass breaking cryptosystems (*crypto*), binary exploitation (*pwn*), web hacking (*web*), etc. Nowadays, CTF competitions have become the leading event in security education [3], providing a high-quality vulnerability dataset for individuals to learn and practice their exploitation skills. Within the realm of CTF, *pwn* is arguably the most challenging one [6], as it requires deep expertise in binary and system internals. A typical *pwn* challenge involves first reversing the binary, identifying vulnerabilities such as stack overflow or heap corruption, and then developing exploit code to achieve goals like information leakage, privilege escalation, or setting up a reverse shell.

Manuscript received 18 October 2022; revised 11 June 2023 and 3 September 2023; accepted 18 September 2023. Date of publication 5 October 2023; date of current version 20 November 2023. This work was supported in part by NSFC under Grant 92270204 and in part by the Youth Innovation Promotion Association Chinese Academy of Sciences (CAS). The associate editor coordinating the review of this manuscript and approving it for publication was Prof. George Loukas. *(Corresponding author: Kai Chen.)*

This work involved human subjects or animals in its research. The authors confirm that all human/animal subject research procedures and protocols are exempt from review board approval.

Dandan Xu, Kai Chen, Miaoqian Lin, and Chaoyang Lin are with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China, and also with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: xudandan@iie.ac.cn; chenkai@iie.ac.cn; linmiaoqian@iie.ac.cn; linchaoyang@iie.ac.cn).

XiaoFeng Wang is with the Luddy School of Informatics, Computing and Engineering, Indiana University Bloomington, Bloomington, IN 47405 USA (e-mail: xw7@indiana.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TIFS.2023.3322319, provided by the authors.

Digital Object Identifier 10.1109/TIFS.2023.3322319

Among the house-of-exploits of *pwn*, heap exploits are perhaps the most rewarding types, due to their complexity and threat to real-world systems (e.g., the notorious Sudo Baron Samedit vulnerability [64]). Microsoft's statistics also reveal that heap corruption accounted for 57% of remote execution vulnerabilities in their products between 2010 and 2017 [32]. However, building a successful heap exploit is not easy, due to the onerous task of program analysis and the complex internals of heap allocators. In practice, triggering a heap corruption vulnerability is only the first step. Attackers must carefully manipulate memory layout and bypass security checks to gain control over memory regions and achieve further goals. These obstacles not only raise the bar of the competition but also impede the assessment of real-world heap vulnerabilities' threats and the timely response to associated risks.

A. Known Techniques and Their Limitations

The substantial complexity and workload of exploitation have led to the development of Automatic Exploit Generation (AEG) [2], which replaces the security analyst's manual work in constructing sophisticated exploits with machines. However, existing AEG literature faces the following limitations:

- *Only targetting non-heap vulnerabilities.* Early AEG efforts [2], [7], [16], [27] primarily target stack-based or format string vulnerabilities, which mainly manipulates the stack frames or format specifiers to synthesize exploits. These approaches cannot be directly applied to heap-based vulnerabilities, as they require multiple interactions with the heap allocator to maintain an exploitable memory layout.
- *Relying on manually summarized patterns.* To facilitate the exploitation of heap vulnerabilities, several solutions [9], [14], [15], [53] develop automated systems based on manually summarized patterns that describe the essential steps and constraints for launching successful attacks. Such a pattern tends to be specific: for example, Gollum [14] can only exploit heap overflow through overwriting function pointers

on the heap. However, when no such pointers are available, one often needs to corrupt heap metadata and abuse design flaws in heap allocators to expand the controlled memory regions and hijack other pointers, like *babyheap* in *FireShell CTF* [70] for overwriting *atoi@GOT* in the Global Offset Table (GOT) and *one* in *SECCON CTF* [72] for overwriting *free_hook* in glibc. The whole procedure of heap exploitation can vary significantly depending on the target program's context, limiting pattern-based approaches to a small subset of vulnerabilities.

In conclusion, existing AEG solutions are either focused on non-heap vulnerabilities or dependent on manually summarized patterns. Given the flexible and evolving nature of heap exploitation, it becomes challenging to manually distill patterns for a broad spectrum of vulnerabilities. This underscores the absence of an automated approach capable of encapsulating exploit patterns for such diverse heap vulnerabilities.

B. Challenges and Our Solutions

Inspired by the human training process of CTF competitions, we observe that public attack artifacts (e.g., CTFtime [60]) carry valuable insights into heap exploitation. Therefore, we aim to fill this gap through *artifact-assisted AEG*, a new paradigm that automatically summarizes exploit experience from artifacts of known exploits, simulating the learning process of CTF participants. This leads us to the following question: *How can we effectively extract and summarize the exploit patterns from existing artifacts, so that they can be applied to guide the generation of new exploits?*

To answer this question, we propose a novel Exploitation State Machine (ESM) to model the extracted patterns, effectively describing the multi-step procedure of heap exploitation. However, several notable challenges must be addressed to develop a robust ESM suitable for heap exploitation.

First, one needs to properly define the set of memory states and actions in the ESM, so that a model can be efficiently extracted to facilitate the exploitation of new programs. A naive way is to simply replay an exploit, and record all its memory changes as the states and operations as the actions during each step. However, this may lead to serious performance issues as the memory space of programs can be exponentially large. Moreover, it may extract memory states and operations that are irrelevant to the exploit, bringing additional noise that reduces the applicability of the extracted guidance.

Second, the actions in the ESM need to be properly handled, as useful patterns are often tied to specific contexts: the operations extracted from an exploit are often bound to the program's runtime context, which cannot be directly transferred to other programs. This is particularly problematic when protections like ASLR [47] are enabled, as the program is loaded with runtime offsets, resulting in different address parameters even for the same sequence of heap operations.

1) Our Solutions: In this paper, we design and implement *AutoPwn*, the first artifact-assisted AEG solution that automatically models the exploit patterns from existing attack artifacts as an ESM and uses it to construct heap exploits for CTF competition. We address the above challenges as follows.

For the first challenge, we use a vector of exploit-related pointers in the target program to represent the ESM's states and the operations performed on them as the ESM's actions. More specifically, we observe that the key step in heap exploitation is the manipulation of *critical variables*, particularly those associated with the management of the heap, which can often be found in text descriptions of the corresponding artifacts. Therefore, *AutoPwn* utilizes a unique strategy that combines the dynamic running of exploit code with semantic analysis of its text descriptions to construct the ESM. It first extracts critical variables from text descriptions, then monitors the execution of exploits to record the states of critical variables and the memory operations on them. Since the heap-related pointers in libraries (e.g., *malloc_hook*) are shared across programs, by properly generalizing the pointers inside individual programs, we can efficiently represent the ESM's states across different programs.

For the second challenge, *AutoPwn* employs an operation generalization method to remove the program-specific variance. Specifically, *AutoPwn* scans the operation traces of a program, performs correlation analysis to associate memory operations based on their target objects, then generalizes the concrete parameters with symbolic values. When *AutoPwn* applies the operation sequences to new programs, it concretizes the symbolic values according to the program's runtime context, so that variations of new programs are accommodated.

2) Contributions: Here we outline this paper's contributions:

- We propose artifact-assisted AEG, a new paradigm that automatically summarizes the exploit patterns from artifacts of known exploits, and uses them to guide the generation of new exploits. Compared to existing pattern-based solutions, our approach does not require deep expert involvement to summarize the patterns for diverse types of vulnerabilities.
- We design and implement *AutoPwn*, the first artifact-assisted AEG system to automate the generation of heap exploits for CTF *pwn* competitions. We propose to use a novel Exploitation State Machine (ESM) to model the patterns of different heap exploits, and design concrete techniques to construct it from attack artifacts and use it to generate new exploits.
- We evaluate *AutoPwn* using a testing dataset of 96 CTF *pwn* binaries. Our experiment result shows that it can generate 22 successful exploits and 13 partial exploits, preliminarily demonstrating its efficacy.

II. Background and Related Work

A. Background

1) Heap Corruption: Heap corruption is a kind of memory corruption that occurs when a heap object is modified beyond the developer's original intention. Various bugs, such as heap overflow, use-after-free (UAF), and double free (DF), can lead to heap corruption. Listing 1 gives an example vulnerability. At Line 11, the object *alloc_list*[*index*] is freed without being set to NULL. Then an attacker can trigger a double free at

```
1    while ( 1 ){
2      scanf("%", &opt);
3      switch (opt) {
4        case 1: // allocate(size, data)
5          scanf("%u", &size);
6          buf = malloc(size);
7          read(0, buf, size);
8          alloc_list[alloc_idx++] = buf; break;
9        case 2: // free(index)
10         scanf("%u", &index);
11         free(alloc_list[index]); break;
12       case 3: // read(index)
13         scanf("%u", &index);
14         puts(alloc_list[index]); break;}}
```

Listing 1. Decompiled Code of *canakmgf_Remastered* [66].

```
void create_array() {
  scanf("%u", &number);
  if (8*(number+1) <= 0x400) {
    list[global_idx] = malloc(8*(number+1));
    list[global_idx++][0] = number; }}
void edit_element() {
  scanf("%u", &arrayIdx);
  if (arrayIdx <= 15) {
    scanf("%u", &Idx);
    if (Idx < list[arrayIdx][0]) {
      scanf("%ld", &value);
      list[arrayIdx][Idx+1] = value; }}}
void delete_array() {
  scanf("%u", &arrayIdx);
  if (arrayIdx <= 15)
    free(list[arrayIdx]); }
void view_element() {
  scanf("%u", &arrayIdx);
  if (arrayIdx <= 15) {
    scanf("%u", &Idx);
    if (Idx < list[arrayIdx][0])
      printf("%ld\n", list[arrayIdx][Idx+1]); }}
```

Listing 2. Decompiled Code of *House of Horror* [73].

Line 11 with the same *index* to free the object again, leading to a crash. The attacker may also trigger a UAF by accessing the dangling pointer at Line 14, resulting in information leakage.

2) Exploitation: A typical way to exploit a memory corruption vulnerability is to overwrite a function pointer with a malicious address, hijacking the program's control flow. Unlike stack exploitation which often exploits function pointers (e.g., return address) available in the stack, heap exploitation is more challenging due to the difficulties in identifying exploitable pointers. Even if a pointer is identified, overwriting it may not be easy, as vulnerabilities like UAF or double free may not directly enable arbitrary writes. To overcome this, attackers often exploit design flaws in heap allocators to expand their capabilities and then achieve goals like control flow hijacking. This process requires in-depth knowledge of the heap allocator (see Appendix A.B for the example of the *ptmalloc2* allocator).

Figure 1 illustrates an example attack for Listing 1. The left part shows the exploit's code and text descriptions, while the right part presents the program's memory states after each step in the exploit. In this attack, the attacker first allocates two heap chunks (chunk0 and chunk1) of size 0×68. Then, the attacker causes a double free by freeing chunk1 twice (Figure 1 (a)). After that, the attacker tries to allocate a new chunk of the same size. Obviously, the allocator returns chunk1 as it is the first available chunk in the fast bin. The attacker can set *chunk1.fd* to *&malloc_hook*-0×23 (the offset 0×23 is set to bypass the allocator's security check). Note that at this time, the fast bin still maintains a link to chunk1. As *fd* is set to *&malloc_hook*-0×23, from the viewpoint of the allocator, it is another available chunk in the fast bin (Figure 1 (b)). When the attacker further requests for three chunks of size 0×68, the allocator eventually returns *&malloc_hook*-0×23, which allows the attacker to overwrite *malloc_hook* with a malicious function (Figure 1 (c)) Finally, the attacker only needs to request a new chunk to trigger the malicious function (Figure 1 (d)). The whole process in this example involves multiple steps that require deep expert experience (e.g., bypassing security checks [43], exploiting suitable variables [8], choosing appropriate chunk sizes [53], etc.). This makes automatic exploit generation quite challenging.

3) Threat Model: We follow the threat model of most CTFs:

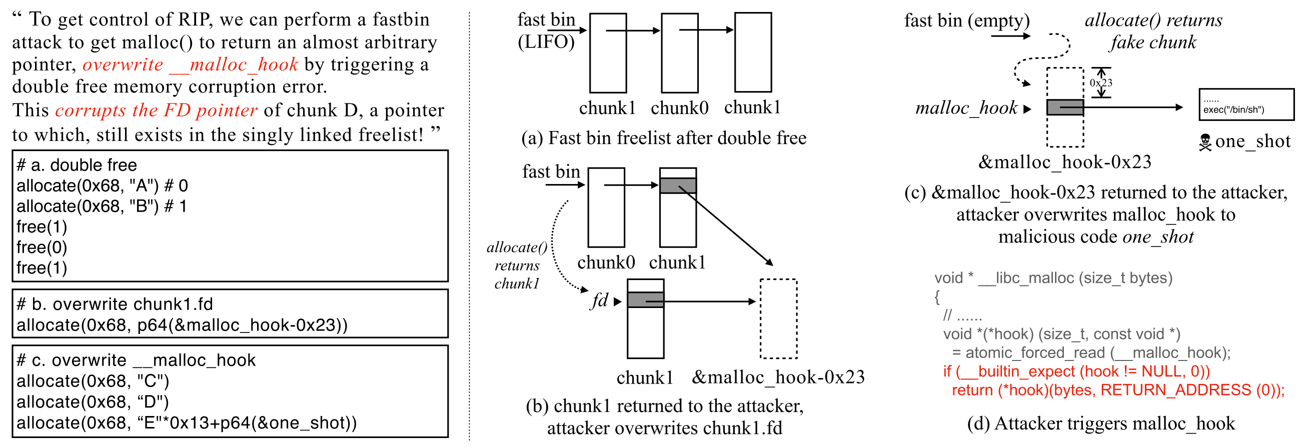
- We assume protections like No-eXecute (NX) [42], Address Space Layout Randomization (ASLR) [47], Position-Independent Executable (PIE) [48] and RELocation Read-Only (RELRO) [40] are enabled, Control-Flow Integrity (CFI) [1] is disabled. Note that although CFI is increasing in its popularity, it is not commonly used in practice.
- We only target deterministic heap allocators, which ensure a consistent memory layout for identical initial states and operations. This allows *AutoPwn* to effectively learn from exploit artifacts and apply them to other programs. This assumption aligns with existing heap AEG studies [13], [14], [24], [44], as the majority of heap allocators, including widely used ones in UNIX-style systems (e.g., ptmalloc [56], dlmalloc [22]), are deterministic [51], and are linked to numerous real-world attack scenarios. Moreover, our extensive CTF testing dataset confirms that out of the 96 challenges, only one [74] utilizes a non-deterministic allocator called mimalloc [23].

Note that our goal is to automatically generate exploits in CTF competitions. Real-world vulnerabilities (e.g., those in the Common Vulnerabilities and Exposures (CVE) database [62]) are not in the scope of this paper, as they often miss important details (e.g., environment information, exploit code), leading to reproducibility issues [34].

B. Related Work

1) Automatic Exploit Generation: Automatic exploit generation is an active area of research. APEG [5] is the first AEG solution that utilizes the patch to construct an exploit for unpatched systems. Subsequently, a line of research [2], [7], [16], [27], [49] focuses on automating the generation of exploits for stack or format string vulnerabilities. Some of them have explored the problem of path explosion [27] and the bypassing of security protections [49]. Despite such efforts, their approaches are not applicable to heap-based vulnerabilities, which are more flexible and complex.

For heap-based vulnerabilities, Repel et al. [37] propose a modular approach based on symbolic execution to automatically find exploit primitives for heap metadata



" To get control of RIP, we can perform a fastbin attack to get malloc() to return an almost arbitrary pointer, overwrite __malloc_hook by triggering a double free memory corruption error.
This corrupts the FD pointer of chunk D, a pointer to which, still exists in the singly linked freelist! "

```
# a. double free
allocate(0x68, "A") # 0
allocate(0x68, "B") # 1
free(1)
free(0)
free(1)
```

```
# b. overwrite chunk1.fd
allocate(0x68, p64(&malloc_hook-0x23))
```

```
# c. overwrite __malloc_hook
allocate(0x68, "C")
allocate(0x68, "D")
allocate(0x68, "E"*0x13+p64(&one_shot))
```

```
void * __libc_malloc (size_t bytes)
{
  // ......
  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));
```

fast bin
(LIFO)
chunk1
chunk0
chunk1
(a) Fast bin freelist after double free
fast bin
allocate() returns chunk1
chunk0
chunk1
fd
chunk1
&malloc_hook-0x23
(b) chunk1 returned to the attacker, attacker overwrites chunk1.fd
fast bin (empty)
allocate() returns fake chunk
0x23
malloc_hook
exec("/bin/sh")
one_shot
&malloc_hook-0x23
(c) &malloc_hook-0x23 returned to the attacker, attacker overwrites malloc_hook to malicious code one_shot
(d) Attacker triggers malloc_hook

Fig. 1. An attack artifact for Listing 1 and its memory states during the attack.

corruptions. Revery [43] combines layout-oriented fuzzing and control-flow stitching to generate heap-based exploits. Gollum [14] exploits heap overflows in language interpreters by attacking the function pointers in interpreters' data structures. Those AEG solutions neither consider bypassing protection mechanisms like ASLR nor can they handle multi-step exploits. To automate multi-step exploits, SLAKE [9], HADE [15], and HAEPG [53] employ manually crafted exploit patterns. Based on these patterns, they use static analysis, taint analysis, or symbolic execution techniques to generate exploits. Compared with these works, *AutoPwn* does not require manual work in summarizing the pattern of exploit generation. Instead, *AutoPwn* automatically learns it from existing artifacts.

Other works only target a subset of the AEG problem, e.g., critical data identification [8], [28], [38], [45], memory layout manipulation [13], [24], [44], [52], shellcode generation [25], [26], [50], etc. Particularly, researchers have explored the application of artificial intelligence technologies in these sub-problems. For example, Wang et al. [45] use deep learning to identify non-control critical data in binary programs. Shellcode IA-32 [25], EVIL [26], and DualSC [50] utilize neural machine translation to automate the generation of shellcode. Since these works only target sub-problems of AEG, they can only be used to assist part of the exploit procedure.

2) Automated Penetration Testing: Metasploit framework [18] is a powerful tool for penetration testing. It provides an arsenal of tools for testing and exploiting a vulnerable system, but requires human intervention to identify and select appropriate exploits. Recently, several works leverage machine learning to automate this process. For example, DeepExploit [41] utilizes machine learning to automatically gain initial access based on Metasploit. Maeda et al. [30] automate post-exploitation activities based on the Empire framework [55]. Additionally, Kujanpää et al. [21] employ deep reinforcement learning to automate privilege escalation. These works use known exploits for penetration testing, while the goal of *AutoPwn* (and AEG) is to generate new exploits for unknown vulnerabilities.

3) Expert Knowledge Extraction: Zhuo et al. [11] propose a method that automatically learns expert knowledge from vulnerability descriptions to predict the severity of future vulnerabilities, thereby reducing manual work. VuRLE [29] learns patterns from vulnerable code examples and their patches to detect vulnerabilities in web applications. He et al. [12] use neural networks to learn how symbolic execution generates inputs to guide a fuzzer to explore unknown smart contracts. Unlike these studies, *AutoPwn* aims to learn experience from previous exploit artifacts to guide the generation of exploits.

4) CTF for Security Education: Numerous efforts have been made to integrate CTFs into security education. For example, AutoCTF [17] automatically generates new CTF challenges by injecting exploitable bugs into programs via a bug injection system. Git-based CTF [46] uses version control systems to host and manage CTF challenges. Kim et al. [19], [20] propose detailed evaluation metrics to assess students' performance in CTF *pwn* challenges. These works aim to facilitate the training of security beginners through CTFs, while the goal of *AutoPwn* is to automatically train an ESM using CTF artifacts for generating new exploits.

III. Design

A. Overview

1) Architecture: Figure 2 illustrates the architecture of *AutoPwn*, which includes two components: *Learning the previous Experience* (LE) and *Using Experience to generate exploits* (UE).

LE extracts the exploit experience from previous artifacts into an Exploitation State Machine (ESM). It first analyzes instructive documents and extracts *critical variables*. Then LE monitors a target program's execution when fed with the exploit code, viewing the changes of critical variables as different *states* and recording the program's memory operations triggered by the exploits as the *actions*. Note that the recorded memory operations are often specific to the target program, so LE generalizes the parameters in the operations to *symbolic* forms for migration. Subsequently, LE extracts states and actions from more programs and exploits, composing them into a final ESM for the next stage.

UE aims to generate new exploits using the ESM to verify its effectiveness. It takes a new program and a proof-of-concept (PoC) that triggers a vulnerability as input,

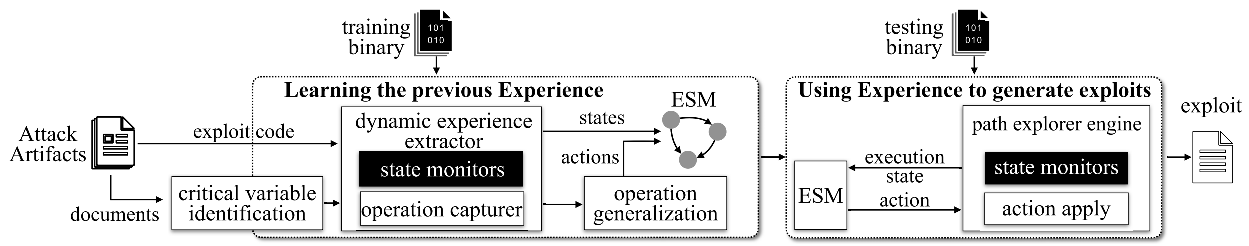


Fig. 2. The architecture of *AutoPwn*.

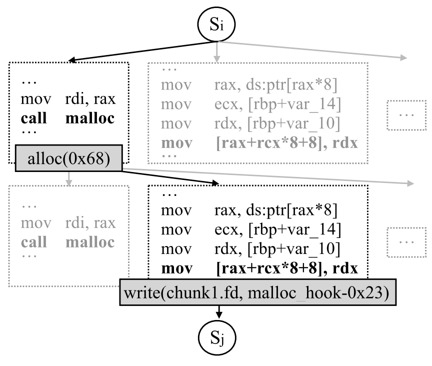
consistently evaluates the program's execution state and matches it to a state in the ESM, then tries to apply the corresponding actions in the ESM to trigger state transitions. When the final state (e.g., execute arbitrary code) is reached, UE outputs the input that lets the target program run to the final state as the exploit. Note that, in this process, the actions in ESM are in generalized form. So when utilizing the actions on another target program, UE concretizes the "symbolic" parameters and further mutates them, just as a human analyst usually does.

2) Example: For illustration, we obtain an ESM from the exploit of Listing 1, then use it to exploit the new program in Listing 2. First, LE analyzes documents and marks *malloc_hook* and *fd* as critical, as shown in Figure 1. Then LE executes the target program in Listing 1 with the exploit code. Assume the program reaches state S_i (Figure 1 (a)) before block b. After the execution of *allocate*$(0 \times 68$, *p64(&malloc_hook*-$0 \times 23))$, *fd* is overwritten by *&malloc_hook*-0×23. So LE adds a new state S_j (Figure 1 (b)) to the ESM and adds the operations as actions. Then LE continues this process to construct the ESM.

To apply the generated ESM to Listing 2, UE first evaluates its execution state and finds the corresponding state S_i in ESM, then tries to apply action F_{ij} to reach S_j (see Figure 3 (a)). Specifically, UE searches for a path in the program that executes the operations in F_{ij} by running through the instructions in the black boxes in Figure 3 (b). Instructions in the gray boxes are other possible paths at S_i, but they are filtered out as they do not match with F_{ij}. After the program has successfully reached S_j, UE moves on to explore the next state.

$$S_i \rightarrow S_j : Fij = \begin{pmatrix} alloc(0x68) \\ write(chunk1.fd) \end{pmatrix}$$

(a) F_{ij} denotes the action to be taken from S_i to S_j



(b) UE uses the action F_{ij} to constrain path exploration of the new program

Fig. 3. An example of exploit generation for Listing 2 guided by the ESM for Listing 1.

B. Learning From Previous Experience

The LE stage extracts useful experience from artifacts and models it as a state machine to describe the multi-step procedure of heap exploitation. Specifically, each exploitation step can be represented by a *state* to be achieved. Letting the target program transfer between any two connected states is the memory operations (aka. *actions*). To capture key steps in exploitation, LE first identifies critical variables to represent the program's memory states. Then it captures the states and actions to recover an ESM for one exploit. Finally, it composes the ESM for multiple exploits. Below we provide the details.

1) Critical Variable Identification: In a document describing an exploit, the author often describes several variables in verb-object phrases. For example, in a writeup for challenge *Sword* [68], the author states that "*…,so I want to overwrite the printf@got. …After entering the weight of the sword in this part, …*". This sentence contains two variables, *printf@got* and *weight*, both used in verb-object phrases. Based on this observation, we leverage constituent-based parsing (popularly used in natural language processing (NLP)) to extract variables. Specifically, we represent a sentence via a context-free parsing tree, where the tree leaves are the words in the sentence, and the non-terminals are the types of phrases. As our target is the verb-object phrase, we focus on verb phrases (tagged as "VP" in Appendix B.A). This allows the LE stage to extract an initial set of variables from the document.

However, only part of the variables are critical for exploitation. In the sentence above, overwriting *printf@got* can lead to control-flow hijacking when *printf* is called, while *weight* does not help in exploitation. Therefore, a distinction needs to be made between the critical ones and others. Interestingly, when describing the operations on critical variables, the authors usually use verbs with malicious intentions (e.g., "overwrite", "corrupt"), indicating that the operations may corrupt the variables' original functionalities. Based on this observation, LE only identifies the objects of malicious verbs as critical variables. Specifically, LE first builds an initial dictionary of popular malicious verbs, then extends it by adding similar verbs. In particular, LE utilizes Word2vec [33] to embed the verbs in verb-object phrases and uses cosine similarity to measure the similarity between the embedding results (i.e., semantic word vectors) with those in the dictionary [35]. If the similarity exceeds a threshold θ_h, LE adds a new malicious verb to the dictionary. If the similarity is below the low threshold θ_l, LE ignores it. Otherwise, we manually analyze it. After that, LE extracts the objects of these malicious verbs as critical variables. Finally, among 964 verb stems in 305 documents, LE identified 15 verb stems (see Appendix B.C) using 2 verbs ("overwrite" and "corrupt") as the initial dictionary. In this process, only 89 verbs required manual analysis. Surprisingly, we found that the identified critical variables are even used as stepping-stones in real-world exploits (e.g., *free_hook* for CVE-2021-21974, *fd* for CVE-2020-6007).

2) ESM for One Exploit: As mentioned before, *AutoPwn* uses Exploitation State Machine (ESM) to model the patterns extracted from attack artifacts. Formally, an ESM is given by a 5-tuple: (S, A, F, s_0, s_e). Here, S is the set of memory states, A is the set of memory-changing actions, and $F : S \times A \to S$ is a function that drives the transition from one memory state to the next. $s_0 \in S$ is the initial memory state, and $s_e \in S$ is the final memory state, which indicates a successful exploitation. Below we present how to extract the states, actions, and transition functions from previously known exploits.

- ***States***. Since the memory space of programs can be exponentially large (cf. Section I-B), we only focus on memory positions that are closely related to the attack, i.e., the critical variables identified from the documents, and view their changes as exploitation states. Specifically, for n critical variables $v_1, \ldots, v_n$ identified in the documents, if an operation reads or writes v_i, a new state is created. We store the state as $< (w_{v_1}, r_{v_1}) \ldots (w_{v_n}, r_{v_n}) >$, where w_{v_i} and r_{v_i} are boolean values (0 or 1) indicating whether v_i is read or written.

To capture the exploitation states, LE monitors the program's execution and records the values of critical variables. In this process, we need to map critical variables' names to their memory addresses. We consider two types of variables: program-specific variable (*PVar*) and non-program-specific variable (*NPVar*). Since we focus on heap exploits, we explicitly classify the variables in heap metadata as NPVars. In addition, we view variables in the Global Offset Table (GOT) and the global variables of libraries as NPVars. Other variables are PVars by default. For example, NPVars "*fd*" is a heap metadata, "*printf@got*" is a GOT entry and "*__malloc_hook*" is a global symbol in Glibc, while "*description*" is a PVar.

For NPVar and PVar, we employ different strategies to calculate their addresses. Regarding an NPVar, we first identify its static offset within the heap, program, or library, and then calculate its actual address during runtime. Regarding a PVar, mapping its name to a memory address is hard at the binary level due to the loss of semantics. So our idea is to monitor all function and data pointers in the heap. If any pointer's value is changed or accessed, a new state is created. We can identify pointers by checking their values and consider them as function or data pointers if they point to the executable section (e.g., *.text*) or the data section (e.g., *.data*) of the program.

Now given an exploit, LE can recover its ESM states by monitoring the program's critical variables. Next, we discuss how to extract the ESM's actions.

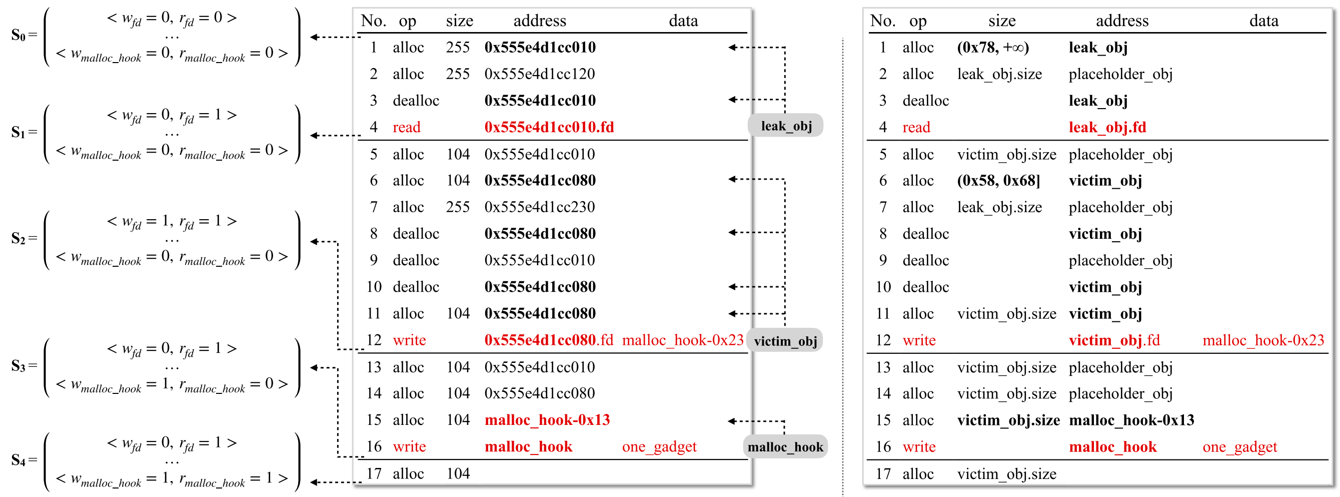
- ***Actions***. Actions are operations that change memory states. For any two connected states s_i and s_j, an action is an operation sequence that manipulates the heap (*alloc*, *dealloc*) or changes variable states (*read*, *write*). Same as state extraction, we dynamically run the exploit to record the state transition operations. However, these operations often use concrete values as parameters (e.g., $0 \times 555e4d1cc010$ in Figure 4), which cannot be directly transferred to other programs. Even for the same program, different executions may yield different values due to memory randomizations (e.g., ASLR), rendering the extracted actions inapplicable even to the same program.

To ensure the transferability of ESM actions, we generalize the extracted operations by replacing concrete parameter values with *symbolic* values.[1] There are two kinds of parameters that need to be replaced: address and the size of heap chunks. For the address of a heap chunk, we define three symbolic values: *leak_obj*, *victim_obj*, and *placeholder_obj*. The *leak_obj* represents a heap chunk read during the exploit, often used to leak important information like the *fd* pointer of a freed chunk. The *victim_obj* refers to a chunk overwritten by the attacker, which often contains pointers targeted for modification. The *placeholder_obj* represents other objects used for memory layout. Recall that a heap chunk is a structure with members, therefore we use the format like *leak_obj.fd* to represent the heap metadata. If a memory address can be calculated using a fixed offset from NPVar, we do not replace it with a symbolic value (e.g., *&malloc_hook*-0×23). Regarding the size of a heap chunk (usually in the *alloc* operation), we use the *range* rule to replace it with a range scope that fits with the heap bins used to store the chunk. For example, we use $(0 \times 78, +\infty)$ to replace the concrete size of a chunk in the unsorted bin. This replacement can help to know which freelist to exploit. In Appendix B.B we provide the size scope of different bins.

Algorithm 1 describes how to generalize memory operations into actions. It first traverses the operation traces and identifies *leak_obj* and *victim_obj* in the exploit according to the read and write operations (Line 5-10). Then it performs correlation analysis from the beginning of the exploit to the current operation (Line 19-29), replacing the address parameters with the corresponding symbolic values if they are pointing to the same heap chunk (Line 21-22). For the parameter indicating the size of a heap chunk, LE replaces it with a symbolic value according to the rules introduced above (Line 24-28). Note that we keep the original form of the memory address related to NPVar since it is already symbolic (Line 4). After the traversal round, the address parameters that are not symbolic will be replaced with *placeholder_obj* (Line 12-16), and the size of *placeholder_obj* will be replaced with *leak_obj.size* or *victim_obj.size* based on its relationship with them.

Figure 4 gives an example. The left side shows the monitored memory states (S_0-S_4) and operations when running the exploit code, and the right side shows the generalized actions by Algorithm 1. Concretely, the first loop of the algorithm traverses the operations and identifies the parameter of *read* operation (op_4) as a *leak_obj*, then calls *Generalize* to process the operations before it. For op_1, the return address

[1] Not to be confused with concepts in symbolic execution. Here we use the word *symbolic* to denote the variables' semantics in exploits.



$$S_0 = \begin{pmatrix} < w_{fd} = 0,\ r_{fd} = 0 > \\ \cdots \\ < w_{malloc\_hook} = 0,\ r_{malloc\_hook} = 0 > \end{pmatrix}$$

$$S_1 = \begin{pmatrix} < w_{fd} = 0,\ r_{fd} = 1 > \\ \cdots \\ < w_{malloc\_hook} = 0,\ r_{malloc\_hook} = 0 > \end{pmatrix}$$

$$S_2 = \begin{pmatrix} < w_{fd} = 1,\ r_{fd} = 1 > \\ \cdots \\ < w_{malloc\_hook} = 0,\ r_{malloc\_hook} = 0 > \end{pmatrix}$$

$$S_3 = \begin{pmatrix} < w_{fd} = 0,\ r_{fd} = 1 > \\ \cdots \\ < w_{malloc\_hook} = 1,\ r_{malloc\_hook} = 0 > \end{pmatrix}$$

$$S_4 = \begin{pmatrix} < w_{fd} = 0,\ r_{fd} = 1 > \\ \cdots \\ < w_{malloc\_hook} = 1,\ r_{malloc\_hook} = 1 > \end{pmatrix}$$

No.	op	size	address	data
1	alloc	255	**0x555e4d1cc010**	
2	alloc	255	0x555e4d1cc120	
3	dealloc		**0x555e4d1cc010**	
4	read		**0x555e4d1cc010.fd**	
5	alloc	104	0x555e4d1cc010	
6	alloc	104	**0x555e4d1cc080**	
7	alloc	255	0x555e4d1cc230	
8	dealloc		**0x555e4d1cc080**	
9	dealloc		0x555e4d1cc010	
10	dealloc		**0x555e4d1cc080**	
11	alloc	104	**0x555e4d1cc080**	
12	write		**0x555e4d1cc080**.fd	malloc_hook-0x23
13	alloc	104	0x555e4d1cc010	
14	alloc	104	0x555e4d1cc080	
15	alloc	104	**malloc_hook-0x13**	
16	write		**malloc_hook**	one_gadget
17	alloc	104		

(a) Memory states and operations

No.	op	size	address	data
1	alloc	**(0x78, +∞)**	**leak_obj**	
2	alloc	leak_obj.size	placeholder_obj	
3	dealloc		**leak_obj**	
4	read		**leak_obj.fd**	
5	alloc	victim_obj.size	placeholder_obj	
6	alloc	**(0x58, 0x68]**	**victim_obj**	
7	alloc	leak_obj.size	placeholder_obj	
8	dealloc		**victim_obj**	
9	dealloc		placeholder_obj	
10	dealloc		**victim_obj**	
11	alloc	victim_obj.size	**victim_obj**	
12	write		**victim_obj**.fd	malloc_hook-0x23
13	alloc	victim_obj.size	placeholder_obj	
14	alloc	victim_obj.size	placeholder_obj	
15	alloc	**victim_obj.size**	**malloc_hook-0x13**	
16	write		**malloc_hook**	one_gadget
17	alloc	victim_obj.size		

(b) Generalized operations (actions)

Fig. 4. An example of operation generalization using Algorithm 1.

$(0 \times 555e4d1cc010)$ of *alloc* is the same as the *leak_obj*, thus they are linked using the symbolic value *leak_obj*. As this heap chunk is placed into the unsorted bin, the size parameter *255* is replaced with the range $(0 \times 78,\ +\infty)$. For op_2, the return address of *alloc* is different from *leak_obj*, so it is skipped at this stage. For op_3, the address of *dealloc* is the same as *leak_obj* so it is also generalized. Now that Algorithm 1 has finished the correlation analysis for the *leak_obj*, it moves on to scan subsequent operations and resolves a *victim_obj* in op_{12}. At this point, the first loop of Algorithm 1 has finished. In the second loop, the algorithm scans the remaining operations and marks their parameters as placeholder objects. For example, in op_2, the address of *alloc* is generalized as *placeholder_obj*, and its size is replaced by *leak_obj.size*.

- ***Transition Functions***. After extracting all the states and actions from artifacts, LE constructs the state transition function $F : S \times A \rightarrow S$. For an action a that runs state s_i to s_j, it adds an entry $F_{ij} : (s_i, a) \rightarrow s_j$ to F. The action is a sequence of operations $\{op_1, op_2, \ldots, op_n\}$ generalized by Algorithm 1.

Remark: For programs with security protections, their exploits inherently encompass the necessary steps for bypassing, which can be extracted from the artifacts and generalized in the ESM. For example, to bypass ASLR, the exploit in Figure 4 (a) leaks information via a *read* operation on a chunk's *fd* pointer, which is generalized as op_4 in Figure 4 (b). Such *read* operations can also be employed to bypass PIE by leaking the program's load address. Moreover, to bypass RELRO, which marks certain memory sections as read-only, op_{16} in Figure 4 (b) utilizes *malloc_hook*, an unprotected function pointer as the stepping-stone target. This guidance is embedded in the parameters of the operation and stored in the ESM. Lastly, to bypass NX, which marks data segments as non-executable, op_{16} further overwrites the *malloc_hook* with a code gadget in glibc, then triggers it in op_{17} to get a shell.

3) ESM for Multiple Exploits: The above methods enable LE to generate an ESM for one exploit. Given multiple exploit artifacts, LE can further construct a composite ESM for all of them. Suppose there are two ESMs (i.e., $ESM_1 = \langle S^1, A^1, F^1, s_0^1, s_e^1 \rangle$ and $ESM_2 = \langle S^2, A^2, F^2, s_0^2, s_e^2 \rangle$) from two exploits. For each state transition $F_{ij}^2 : (s_i, a) \rightarrow s_j$ in ESM_2, LE adds s_i and s_j to S^1 if they are new in S^1. LE then adds a to A^1 if it is not there. At last, LE adds an entry $F_{ij}^1 : (s_i, a) \rightarrow s_j$ to F^1 to connect s_i and s_j in ESM_1.

During our research, we found that merging ESMs helps a lot in exploitation. Sometimes, the experience from one exploit is insufficient to exploit a new program. But through the combination of multiple exploits, the exploit can succeed (see Section V-C). Also interesting is that the combination of ESMs can help *AutoPwn* find different ways to attack, similar to a human expert who learns more with multiple exploits.

C. Using Experience to Generate Exploits

Given a new vulnerable target program P_t, UE uses ESM to guide its exploit generation. The basic idea is to see whether there is a path in the ESM that can let P_t run towards the final state s_e. If s_e is reachable, then it indicates a successful attack. In this process, the critical variables in P_t are constrained to match with the states in the ESM. Here, we first show how to identify heap operations in P_t and apply ESM actions to P_t, then present the algorithm for generating exploits.

1) Identifying Heap Operations: To apply ESM actions to a new program, UE needs to find corresponding operations in it. However, the set of heap operations in a program can be quite large, and trying them one by one is very time-consuming. Therefore, UE tries to find the operations that are more likely to be used in the exploit. Concretely, it uses the following metrics to prioritize the selection of heap operations. (1) *Degree of Freedom (DOF)*. It measures the attacker's ability to control the size of an operation. For example, an *alloc* call of arbitrary size has the highest DOF, while a call with constant size has the lowest DOF. During exploration, UE prefers operations with higher DOF. (2) *Depth of the Call-site (DOC)*. It measures the lowest number of function calls between the program's entry point and the heap operation. UE prefers to use the operation with a low DOC to reduce the complexity of path exploration. (3) *Pairing State* of *alloc* and *dealloc*

Algorithm 1 Operation Generalization

```
   Input: $operations = \{op_1, op_2, \ldots, op_n\}$
   Result: $Q_{action}$ (a queue of generalized operations)
1  begin
2      $Q_{action} \leftarrow \phi$
       /* resolve leak & victim objects */
3      for $op_i$ in $operations$ do
4          if $op_i.addr$ is not NPVar then
5              if $op_i.type = read$ then
6                  $op_i.symval \leftarrow$ "leak_obj"
7                  Generalize($Q_{action}, op_i$)
8              else if $op_i.type = write$ then
9                  $op_i.symval \leftarrow$ "victim_obj"
10                 Generalize($Q_{action}, op_i$)
11             $Q_{action}.append(op_i)$
       /* mark remaining objects as
          placeholders */
12     for $op_i$ in $Q_{action}$ do
13         if $op_i.symval = \phi$ then
14             $op_i.symval \leftarrow$ "placehold_obj"
15         if $op_i.symsize = \phi$ then
16             $op_i.symsize \leftarrow$
               victim_obj.size or leak_obj.size
17     return $Q_{action}$
   /* forward correlation analysis */
18 Function Generalize ($Q_{action}, op_{sym}$):
       Input: $Q_{action}$ and a generalized operation $op_{sym}$
19     $size_{sym} \leftarrow \phi$
20     for $op_k$ in $Q_{action}$ do
21         if $op_k.addr = op_{sym}.addr$ then
22             $op_k.symval \leftarrow op_{sym}.symval$
23             if $op_k.type = alloc$ then
24                 if $size_{sym} = \phi$ then
25                     $size_{sym} \leftarrow range(op_k.addr)$
26                     $op_k.symsize \leftarrow size_{sym}$
27                 else
28                     $op_k.symsize \leftarrow op_{sym}.symval.size$
29     return
```

invokes. If the chunk allocated in previous *alloc* calls can be released by a *dealloc* call, UE marks these calls as *paired*. UE prefers paired invocations as they can be used to dig holes in memory to maintain a desired memory layout.

2) Applying ESM Actions: Since the actions in ESM are sequences of generalized operations (*alloc*, *dealloc*, *read* and *write*) with symbolic values, they cannot be directly transferred to exploit P_t. Instead, UE finds execution paths in P_t that perform these operations and replaces their symbolic values with concrete ones to apply actions. For *alloc* and *dealloc* operations, UE selects paths that perform these operations based on the aforementioned three metrics, DOF, DOC, and pairing state. For *read* and *write* operations, since normal programs rarely touch critical variables (e.g., *read@GOT*), static analysis cannot find such operations directly from normal executions of the target program. Therefore, UE matches *read* and *write* operations on critical variables during dynamic execution. Once a path containing the operation is found in P_t, UE replaces its symbolic values with concrete ones and uses the concrete values to further constrain the target program to enforce the operation.

We apply the actions in Figure 4 (b) to the target program in Listing 2 as an example. The action from S_0 to S_1 contains 4 heap operations. The only call site for the first operation op_1 *alloc*($0 \times 78, +\infty$) in Listing 2 is at Line 4. UE monitors the program's execution to determine if there is a path that arrives at the call site, then concretizes its *size* value at the range boundary and sets it to 0×79 if the operation is matched. Otherwise, UE terminates the current execution path. The returned address is stored to *leak_obj*. Subsequently, UE matches the second *alloc* operation op_2 at Line 4 and replaces the parameter *leak_obj.size* with 0×79. For op_3, UE matches it with Line 16 of Listing 2, and replaces *leak_obj* with the returned address of op_1. The fourth operation op_4 attempts to read the contents of *leak_obj.fd*, and is matched at Line 22 by reading the value at address *leak_obj*+offset(*fd*). After the action is executed, UE checks whether the state S_1 is satisfied. Note that sometimes concretizing values at the range boundary may not work. For example, choosing 0×79 as the parameter of *alloc* does not work for the program in Listing 2, since the size of *malloc* must be a multiple of 8 (Listing 2, Line 4). In this case, UE mutates and tries other values in the range until one works or timeout. This process is similar to a human analyst trying to find a suitable heap chunk size during the attack.

3) Exploit Generation: Algorithm 2 shows the complete process of exploit generation by traversing the states in ESM. Before going into the details, we first introduce two operations on the ESM. (1) *Action Query* ($AQ(S) \rightarrow 2^A$): given a state $s_i \in S$, $AQ(s_i)$ returns a subset of A, showing all possible actions that can be processed after s_i in ESM. (2) *State Equivalence Query* ($EQ(S, S) \rightarrow \{True, False\}$): given two states $s_1, s_2 \in S$, $EQ(s_1, s_2)$ returns true if each corresponding values of the two states are identical. Otherwise, it will return false.

Suppose the current state of the target program P_t is s_i. Algorithm 2 first picks an action a from $AQ(s_i)$. Usually, it will start from the most popularly used action in $AQ(s_i)$, which can be determined by calculating the frequencies of actions used in the training exploits. Then P_t applies the action a and ends with state s_j (Line 11). If $EQ(s_j, F(s_i, a)) = True$, we say that the action a successfully lets P_t runs to the state $F(s_i, a)$. After that, the process moves on to check whether $AQ(s_j)$ can let P_t run to further states in the ESM (Line 10-15 in Algorithm 2). If $EQ(s_j, F(s_i, a)) = False$, the next action in $AQ(s_i)$ will be used for testing (Line 10). If there is no further available action in $AQ(s_i)$, the process will return to the parent state of s_i (Line 16) until the root of the recursion (Line 2). Once all the actions in a path drive P_t to the final state s_e, the exploit is successfully generated.

IV. Implementation of the Prototype

1) Dataset: We used public CTF *pwn* artifacts from CTFtime [60] to train and test *AutoPwn*. Specifically, we first searched for challenges with keywords "heap" and "pwn" to find heap-related *pwn* challenges. We then used the links

Algorithm 2 Exploit Generation P_t

```
Input: Testing binary $b$ and the learned
       $ESM = (S, A, F, s_0, s_e)$
Result: An exploit for binary $b$
1  begin
2      $\mathbb{A} \leftarrow$ DFSExplore($s_0, \phi$)
3      if $\mathbb{A} = \phi$ then
4          return $\epsilon$
5      return GenExploit($b, \mathbb{A}$)
   /* recursive dfs search function */
6  Function DFSExplore($s_i, \mathbb{A}$):
       Input: current exploit state $s_i$ and intermediate
              actions list $\mathbb{A}$
       Output: final actions list $\mathbb{A} = \{a_0, \ldots, a_n\}$
7      if $s_i = s_e$ then
8          return $\mathbb{A}$
9      $candidate\_actions \longleftarrow AQ(s_i)$
10     for $a$ in $candidate\_actions$ do
11         $s_j \longleftarrow$ ApplyAction($s_i, a$)
12         if $EQ(s_j, F(s_i, a)) = True$ then
13             $ret \leftarrow$ DFSExplore($s_j, \mathbb{A} \cup \{a\}$)
14             if $ret \neq \phi$ then
15                 return $ret$
16     return $\phi$
```

available on CTFtime to crawl the binaries and documents for those challenges. We used the challenges before 2019 to train an ESM and the challenges after 2019 for evaluation. Very occasionally, if a challenge's link is missing or expired, we manually searched for it online. If it is not available online, we removed the challenge from our training dataset. Note that all the collected artifacts have been checked for their validity. In sum, we collected 305 attack documents and 75 challenges with 103 exploits for training and 96 challenges for evaluation.

2) Learning From Previous Experience: We first leverage NLP to identify critical variables. Particularly, we trained a Word2vec model to generate semantic word vectors and used NLTK [4] and CoreNLP [31] to generate the parse tree of sentences and tag part-of-speeches. We calculated cosine similarity to measure the similarity between verbs. We set the thresholds $\theta_l = 0.38$ and $\theta_h = 0.48$, The choice of these thresholds helped us to identify 124 critical variables from 50 randomly selected documents. Among them, 94 are true positives and 30 are false positives, while 1,341 are true negatives and 5 are false negatives. The accuracy is 97.6% and the recall is 94.9%, indicating that the results are good enough using the two thresholds. Note that a smaller θ_l and a larger θ_h require more manual effort for identification but do not introduce errors. Also, note that the efforts are one-time.

Then we dynamically monitor the execution of the target program with the exploit to extract experience. To accomplish this job, we utilized *LD_PRELOAD* to hook function calls. In this way, when the target program calls a hooked function, our analysis code will first take control. After running our tasks (e.g., recording the parameters and return values), it gives the control back to the original function. In addition, we developed a GDB Python script to speculate the program's memory states at runtime. Finally, we implemented a Python script to generalize the operations using Algorithm 1.

3) Using Experience to Generate Exploits: When applying ESM actions on the target program P_t, we need to map the ESM's operations to similar functions in P_t. To achieve this goal, we first used Binary Ninja [54] to statically analyze P_t and extracted all the functions that perform heap operations. Then we calculated the three metrics (i.e., DOF, DOC, pairing states, see Section III-C) of each function. Such calculation is straightforward using data-flow and control-flow analysis. Finally, we sorted the functions according to the three metrics.

We implemented Algorithm 2 as a plugin of S2E, a symbolic execution engine [10], to generate exploits using ESM. The plugin starts with the PoC as input, then applies ESM actions by filtering out irrelevant program paths and focusing only on paths that perform operations matched by the ESM. Specifically, we first hooked the program's input interfaces, such as *scanf* and *recv*, to inject symbolic data. Then we prioritized the exploration of paths that involve heap operations for applying *alloc* and *dealloc* operations, based on the three metrics. Additionally, we utilized S2E's signal handlers to capture memory access and apply *read, write* operations. If an ESM action cannot be applied at the current state, the plugin terminates the execution of this path. Finally, the symbolic engine outputs the constraints for input variables, and variables that depend on heap or library addresses are annotated with their offsets, which are calculated using the leaked information at runtime. Sometimes the program executes for a long time, so we set up a timeout of 24 hours.

V. Evaluation

In this section, we evaluate the effectiveness of *AutoPwn* on CTF challenges and real-world CVEs. All experiments were conducted on a Ubuntu 18.04 server with 8 Intel(R) Xeon(R) Silver 4110 CPU@2.10GHz cores and 128 GB memory.

A. Effectiveness on CTF Challenges

We used CTF challenges after 2019 (average size is 14.1 KB) for evaluation. Table I shows the evaluation results: *AutoPwn* managed to generate full or partial exploits for 35 challenges (average size is 12.3 KB). Among them, 22 challenges have full exploits, while 13 have partial exploits. A partial exploit achieves some key steps in the attack, e.g., information leakage or control flow hijacking. For example, facing the challenge *Hard heap* in *HSCTF 6* [71], *AutoPwn* successfully leaked the program's heap address, which is critical for bypassing ASLR. This allows human participants to partially solve a CTF challenge using *AutoPwn* and manually complete the remaining steps, saving them time and effort.

We then analyzed why *AutoPwn* failed to generate the exploits for some challenges. The two main reasons are: (1) *New defenses*. New defenses in the newer version of the allocator can make past exploit methods ineffective. For example, we observe that several exploits in our training dataset (before 2019) exploit *tcache* for double-free. However, Glibc 2.29 adds security checks to patch the hole [57]. Therefore, *AutoPwn* cannot bypass this check with only experience in Glibc<2.29. Once *AutoPwn* learns the experience, more challenges could be solved. (2) *Path explosion.* We leverage S2E for symbolic execution and limit the execution time to 24 hours. However, some programs have many path constraints, which impede S2E's execution. For example, the challenge *securenote* in *Balsn CTF 2019* [69] uses AES-CTR to encrypt data on the heap, which contains too many path constraints for S2E to analyze. If S2E is given enough time or optimized for such a situation, more challenges could be automatically solved.

1) Comparison With the State-of-the-Art AEG Tool: To the best of our knowledge, no prior work has explored the automatic summarization of exploit patterns for AEG. Nevertheless, we tried to collect state-of-the-art AEG tools for comparison. However, most of them cannot handle heap vulnerabilities or are not publicly available. Finally, we identified the available tool Rex [58] for comparison, which is an auto-exploit kit framework that achieved third place in the DARPA CGC competition [59]. Rex uses symbolic execution to analyze a crash and identifies exploitable states, then generates specific exploits for the target program. Similar to *AutoPwn*, it takes a PoC as input. Therefore, we ran Rex on the same dataset with the same input[2] to check if it can generate exploits for heap CTF challenges. Table I gives the results: Rex failed to produce exploits for any of the CTF challenges (with a time limit of 24 hours). The primary reason is that Rex depends on a limited set of manually summarized rules. The result demonstrates that our automatically summarized ESM is more effective than the exploit patterns currently supported by Rex. While Rex can potentially perform better with expert refinement, it requires significant manual work, whereas *AutoPwn* automates this process by analyzing existing artifacts.

2) Performance: We measured the time that *AutoPwn* spent to generate exploits. The results are shown in the last column of Table I. On average, it took *AutoPwn* 69.3 minutes to generate a complete exploit for the 22 successful challenges, 18.82 minutes to generate an information leakage exploit for 5 challenges, and 31.2 minutes to generate a control-flow hijacking exploit for 8 challenges. The fastest one (#18) only used 13 seconds, while the longest one (#9) took 13.1 hours. We analyzed the logs of #9 and found that its exploit covered 8 different states (including 25 operations) in the ESM and contained more input constraints than challenges #8,#10, which is more complex than others. Since most CTF events last more than 24 hours (92.16% in 2021 [61]), Table I shows that *AutoPwn* is efficient in generating exploits.

B. Effectiveness on Real-World CVEs

Currently, *AutoPwn* cannot generate exploits independently for real programs due to its reliance on symbolic execution (Section VI). Nevertheless, we found that *AutoPwn*'s guidance can help human analysts to write exploits. To evaluate this, we conducted a human study with security analysts and asked them to write exploits for real-world vulnerabilities using *AutoPwn*. Specifically, we recruited 8 skilled CTF players that achieved similar scores in heap *pwn* challenges and randomly divided them into two groups. For each vulnerability, we randomly provide one group (Group A) with access to *AutoPwn*, while the other group does not (Group B).

In our experiment, we prefer vulnerabilities with public exploits to ensure their exploitability. Therefore, we searched ExploitDB [63] and GitHub to find heap vulnerabilities with publicly available exploits. Ultimately, we identified 22 such vulnerabilities. However, some exploits cannot be reproduced in our environment. Such work of reproduction is known to be hard [34]. We tried our best and spent 12 hours for each exploit to tune its environment and parameters for reproduction. Finally, we successfully reproduced 3 vulnerabilities in the CVE database (Table II) for the experiment. We measured the time taken by each group and assessed their progress.

Table II gives the result: given access to *AutoPwn*, Group A successfully produced exploits for all CVEs, with an average time of 2.77 hours; Group B managed to exploit 2 out of 3 CVEs without *AutoPwn*, but spent significantly more time (+1.16 hours) than Group A. The result indicates that *AutoPwn* can improve security analysts' efficiency in writing exploits.

To understand how *AutoPwn* provided assistance, we observed and analyzed the players' behaviors. Specifically, for CVE-2021-3345, Group A used *AutoPwn*'s state equivalence query (EQ, see Section III-C) to check the program's state on the execution path of the PoC and found a matching state where a function pointer on the heap was overwritten in the ESM. By manipulating the input, they were able to overwrite the function point with the address of a code gadget. Subsequently, guided by *AutoPwn*'s action query (AQ), they used a *read* operation to hijack the control flow and complete the exploit. On the other hand, Group B had to manually analyze the PoC to identify exploitable states and devise their exploit strategies, which took more time. In particular, for CVE-2021-3156, Group A quickly located the key heap operations with the help of *AutoPwn*, which provided clues about two functions (*setlocale* and *get_user_info*). Consequently, they successfully manipulated the heap layout with the two functions and wrote the exploit. Meanwhile, Group B struggled when analyzing numerous heap operations in its PoC. Consequently, Group B failed to generate the exploit within 12 hours.

Additionally, we let *AutoPwn* learn from the three CVEs' artifacts. The key steps were automatically extracted and stored into the ESM. In the LE stage, more pointers (669 on average) were monitored than in CTF challenges (14 on average). But it only took *AutoPwn* slightly more time for analysis, indicating that *AutoPwn*'s LE stage is scalable to real-world exploits.

In conclusion, our experiments show that *AutoPwn* can improve analysts' efficiency in producing exploits for real-world vulnerabilities. Also, *AutoPwn* can efficiently learn the experience from real-world exploit artifacts.

C. Case Studies

In this section, we present two case studies to demonstrate the effectiveness of *AutoPwn* from two aspects: the portability of ESM, and the effectiveness of merged ESMs.

[2]The PoCs we utilized are available at https://github.com/0xddhub/data.git

TABLE I
EVALUATION RESULTS OF *AutoPwn*

No.	CTF Event Name	Challenge Name	Vuln. Type[1]	Glibc Version	IL[2]	CFH[2]	Exploit Created	Rex	ESM States[3]	ESM OPs[3]	Time (min)
1	Security Fest 2019	Baby5	UAF	2.27	✓	✓	Yes	No	5	10	83.6
2	UTC-CTF 2019 Teaser	Newton	UAF	2.27	✓	✓	Yes	No	5	10	7.2
3	UTC-CTF 2019 Teaser	Mixtape	UAF	2.27	✓	✓	Yes	No	5	15	32.9
4	TJCTF 2019	House of Horror	UAF	2.23	✓	✓	Yes	No	5	11	15.6
5	Codegate CTF 2019	god-the-reum	UAF	2.27	✓	✓	Yes	No	5	13	119.4
6	tsg-live-ctf-4	ShyEEICtan	UAF	2.27	✓	✓	Yes	No	5	15	42.0
7	BSides Delhi CTF 2019	message_saver	UAF	2.23	✓	✓	Yes	No	5	9	28.7
8	SECCON 2019 Online CTF	one	UAF	2.27	✓	✓	Yes	No	8	27	14.7
9	TJCTF 2019	Halcyon Heap	UAF	2.23	✓	✓	Yes	No	8	25	785.8
10	RedpwnCTF 2019	penpal world	UAF	2.27	✓	✓	Yes	No	8	30	60.8
11	DEF CON Qualifier 2019	speedrun-010	UAF	2.27	✓	✓	Yes	No	4	6	4.1
12	westlake 2019	noinfoleak	UAF	2.23	✓	✓	Yes	No	6	10	100.9
13	FireShell CTF 2019	babyheap	UAF	2.26	✓	✓	Yes	No	6	8	8.0
14	KipodAfterFree CTF	CookShow	UAF	2.23	✓	✓	Yes	No	5	11	8.0
15	picoCTF 2019	messy-malloc	UAF	2.23	✓	✓	Yes	No	2	4	0.6
16	HackPack CTF 2020	bookworm	UAF	2.27	✓	✓	Yes	No	5	8	7.1
17	HackPack CTF 2020	ToddlerCache	UAF	2.26	✓	✓	Yes	No	4	10	21.7
18	UMDCTF 2020	Jump Not Found	HO	2.23	✓	✓	Yes	No	2	1	0.22
19	Zh3r0 CTF 2020	command-2	UAF	2.27	✓	✓	Yes	No	5	9	142.4
20	COMPFEST CTF 2020	Binary Exploitation	HO	2.23	✓	✓	Yes	No	3	2	15.3
21	b01lers CTF bootcamp	There is no spoon	HO	2.23	✓	✓	Yes	No	2	1	1.8
22	NACTF 2020	Covid tracker	UAF	2.27	✓	✓	Yes	No	2	4	24.1
23	TJCTF 2019	House of Horror 2	UAF	2.23	✓		No	No	3	5	1.3
24	UTCTF 2019	Encryption Service	UAF	2.23	✓		No	No	3	5	20.4
25	starCTF 2019	girlfriend	UAF	2.29	✓		No	No	3	5	2.5
26	Hgame 2020	Roc826	UAF	2.23	✓		No	No	3	4	62.8
27	m0leCon CTF 2020	pheappo	UAF	2.27	✓		No	No	3	5	7.1
28	BackdoorCTF 2019	babyheap	UAF	2.23		✓	No	No	4	6	60.8
29	N1CTF 2019	Warmup	DF	2.27		✓	No	No	4	6	36.3
30	De1CTF 2019	Weapon	UAF	2.23		✓	No	No	4	6	72.8
31	HSCTF 6	Aria Writer	DF	2.27		✓	No	No	4	7	10.2
32	HSCTF 6	Aria Writer v3	DF	2.27		✓	No	No	4	7	13.6
33	Rooters CTF	USER ADMIN	UAF	2.27		✓	No	No	4	8	5.7
34	N1CTF 2019	BabyPwn	UAF	2.23		✓	No	No	4	10	8.6
35	redpwnCTF 2020	four-function-heap	UAF	2.27		✓	No	No	4	7	41.3

[1] Vulnerability type. **UAF**=**U**se-**A**fter-**F**ree, **HO**=**H**eap **O**verflow, **DF**=**D**ouble **F**ree.
[2] Partial exploit goals achieved. **IL**=**I**nformation **L**eakage, **CFH**=**C**ontrol-**F**low **H**ijacking.
[3] The number of ESM states and **OP**eration**s** (**OPs**) used to generate the exploit.

TABLE II
HUMAN STUDY RESULTS OF *AutoPwn*

CVE ID	Application	Time (hours)	
		Group A	Group B
2016-10190	ffmpeg	2.37	3.75 (+1.38)
2021-3156	sudo	5.18	Timeout
2021-3345	libgcrypt	0.76	1.69 (+0.93)

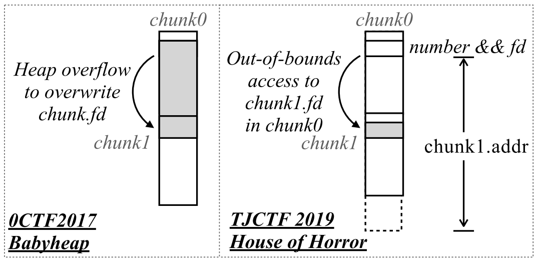

chunk0
Heap overflow to overwrite chunk.fd
chunk1
0CTF2017 Babyheap
chunk0
Out-of-bounds access to chunk1.fd in chunk0
number && fd
chunk1.addr
chunk1
TJCTF 2019 House of Horror

Fig. 5. (Case 1) Different methods to overwrite $victim\_obj.fd$ in *House of Horror* and *Babyheap*.

1) Portability of ESM: The ESM that *AutoPwn* learns from previous exploits can be used to exploit new programs. For example, *AutoPwn* can use the experience from *Babyheap* [65] to attack a new vulnerability in *House of Horror* [73]. Although the two programs are different in vulnerability types, data structures, and interfaces, *AutoPwn* finds that they can reach the same exploitation state, where the *victim_obj.fd* is overwritten. Interestingly, the two exploits use different ways to overwrite the victim pointer. *House of Horror* overwrites the victim pointer using an out-of-bounds vulnerability. In contrast, *Babyheap* has to overflow a previous buffer to overwrite the *fd* pointer in the next chunk, as shown in Figure 5.

This case demonstrates that although the programs and vulnerabilities are different, the exploitation experience from *Babyheap* can still effectively guide the generation of exploits for *House of Horror*. We checked other cases and found similar situations. Such migration is enabled by ESM's reduction of states and generalization of operations, which reduced the impact of irrelevant operations while ensuring their transferability to new programs with different contexts.

2) Effectiveness of Merged ESMs: *AutoPwn* can further combine the experience from multiple artifacts to generate new

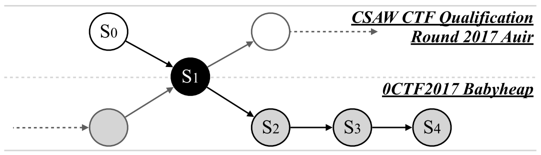


Fig. 6. (Case 2) Exploit for *House of Horror* combines *Babyheap* (heap overflow) and *Auir* (UAF).

exploits. For example, the ESM in Figure 6 is combined from two exploits: the states in gray color are from *Babyheap* [65] (heap overflow), while the states in white color are from *Auir* [67] (UAF). We show how to use the combined ESM to generate a new exploit for *House of Horror* [73].

We find that *House of Horror* contains a UAF vulnerability. Using the states from *Auir* can help to leak the base address of glibc (state S_1 in Figure 6). Subsequently, the attack is carried on by overwriting an entry in GOT. However, in the case of *House of Horror*, the GOT entry is not writable, so the further states from *Auir*'s are not applicable. Fortunately, the state S_2 from *Babyheap* works. It exploits the fast bin and manages to overwrite the function pointer *malloc_hook*, which drives the attack forward to a successful exploit. From the example, we can see neither of the two ESMs can work separately. Merging ESMs from different exploits can greatly expand *AutoPwn*'s capability to exploit new vulnerabilities. This is similar to human experts who learned a lot from previous artifacts and combined the experience to solve complex problems.

VI. Discussion

1) Ethics: Using *AutoPwn* over CTFs does not bring ethical concerns, since CTFs are for experimental purposes. The human experiment in Section V-B is performed in the local environment and the CVEs are disclosed and fixed vulnerabilities with public exploits. All participants are voluntary.

2) Limitations: Currently, we assume deterministic allocators (same as [13], [14], [24], [44]), i.e., the same sequence of heap operations will result in the same memory layout, thus *AutoPwn* can apply operations from previous artifacts to new programs. Without this assumption, the operation sequences may be different for each run of an attack. In this case, even for human experts, additional efforts need to be spent to tune an exploit according to the program's runtime context.

Additionally, in the LE stage, the extracted memory operations may contain noises, reducing the ESM's effectiveness. Specifically, a complex program may involve irrelevant logical operations (e.g., format checking), which introduces noise operations to the LE stage, leading to a large ESM with less relevant states and actions. Also, the expertise of CTF players varies, resulting in differences in the quality of artifacts. For instance, an experienced player may produce a succinct and clear artifact, while a less experienced player may produce an artifact with vague and redundant operations. This also brings additional noise, affecting the effectiveness of the ESM.

Furthermore, when applying actions in the UE stage, we prioritize the selection of paths with heap operations based on three static metrics (see Section III-C), which may not be very effective for large programs. Smarter metrics can be evaluated at runtime for path selection, especially when a path has multiple malloc/free calls. In the current implementation, we use the symbolic execution tool S2E to apply ESM actions. The inherent path explosion problem of symbolic execution has restricted our testing on complex programs.

3) Future Research: Despite the preliminary effectiveness of *AutoPwn*, we find it still has room for improvement. For example, as stated in this paper, the CTF documents provide a rich set of exploit knowledge. Currently, we only identify critical variables to reduce the space of program memory to be monitored. More knowledge could be extracted to improve the effectiveness of ESM. Another direction is to expand the training dataset for *AutoPwn*. Currently, we used exploits before 2019 for training and failed to solve some new problems after 2019. If more artifacts are learned, *AutoPwn* could be more effective at a broader range of problems. Also, when *AutoPwn* fails to produce an exploitable memory state with existing ESM actions, one can integrate previous works on heap layout manipulation [13], [24], [44], [52] to perform the task. Finally, *AutoPwn* only targets heap corruption vulnerabilities in CTF competitions. It can be further extended to real-world systems. Below we briefly discuss its extension.

4) Extension to Real Systems: When migrating the methodology of *AutoPwn* for automation in real systems, there are several practical issues that need to be addressed. Particularly, since real-world programs often involve numerous memory operations, new techniques are required to reduce the noise brought by abundant operations in both the LE and UE stages. Specifically, in the LE stage, reduction techniques such as C-Reduce [36] can be employed to minimize the size of the exploit code and the extracted operations while ensuring exploitability. In the UE stage, a combination of static analysis, fuzzing techniques, and symbolic execution can be utilized to efficiently identify a path that corresponds to ESM's operation, improving the efficiency of $ApplyAction$ in Algorithm 2. Moreover, when the automated process struggles to produce an exploit in time, one can adopt the human-assisted approach [39] to leverage human knowledge to advance the process. There are typically two scenarios. First, when the ESM contains the exploitation steps but gets stuck while exploring execution paths in complex programs, *AutoPwn* can present the current states to humans and seek human assistance in specifying additional path constraints to prevent it from exploring irrelevant paths. For example, when exploring CVE-2016-10190, *AutoPwn* struggles to construct a valid HTTP data packet that can pass FFmpeg's format checks. Through human intervention, the user can take control and manually provide the constraints that satisfy the checks to advance the exploration. Second, when certain exploitation steps are missing in the ESM, users can manually investigate the vulnerability, pinpoint a potentially exploitable path within the program, integrate it into the ESM, and subsequently let the UE stage execute along this path and generate an exploit. In this way, *AutoPwn* can be extended to handle many complex scenarios, helping penetration testers and security researchers in their timely assessment and response to new security threats.

VII. Conclusion

In this paper, we introduce the first artifact-assisted AEG solution, *AutoPwn*, which automates the extraction of exploit patterns to address the flexibility issues inherent in existing AEG literature. Compared to previous solutions that heavily rely on human experts' experience to summarize exploit patterns, *AutoPwn* harnesses the observation that public attack artifacts offer key insights into the exploits. To this end, *AutoPwn* adopts a novel Exploitation State Machine (ESM), characterized by well-defined, generic states and transitions, and then effectively recovers it by automatically analyzing the artifact collection, including both code and documents. We implement and evaluate *AutoPwn* on CTF *pwn* competitions. The evaluation results show that after learning from the experience in passed CTF artifacts, *AutoPwn* generates 22 exploits and 13 partial exploits for 96 new binaries, preliminarily demonstrating its efficacy.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Oct. 2009.

[2] T. Avgerinos, S. K. Cha, B. Lim, and D. Brumley, "AEG: Automatic exploit generation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011, pp. 74–84.

[3] T. Balon and I. Baggili, "Cybercompetitions: A survey of competitions, tools, and systems to support cybersecurity education," *Educ. Inf. Technol.*, vol. 28, no. 9, pp. 11759–11791, Sep. 2023.

[4] S. Bird, E. Klein, and E. Loper, *Natural Language Processing With Python*, 1st ed. Sebastopol, CA, USA: O'Reilly, 2009.

[5] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. IEEE Symp. Secur. Privacy*, May 2008, pp. 143–157.

[6] T. J. Burns, S. C. Rios, T. K. Jordan, Q. Gu, and T. Underwood, "Analysis and exercises for engaging beginners in online CTF competitions for security education," in *Proc. ASE USENIX Secur. Symp.*, 2017, pp. 1–9.

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 380–394.

[8] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1165–1184.

[9] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1707–1722.

[10] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[11] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 125–136.

[12] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 531–548.

[13] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 763–779.

[14] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1689–1706.

[15] N. Huang, S. Huang, and C. Chang, "Analysis to heap overflow exploit in Linux with symbolic execution," *IOP Conf. Ser., Earth Environ. Sci.*, vol. 252, no. 4, 2019, Art. no. 042100.

[16] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations," in *Proc. IEEE 6th Int. Conf. Softw. Secur. Rel.*, Jun. 2012, pp. 78–87.

[17] P. Hulin et al., "AutoCTF: Creating diverse pwnables via automated bug injection," in *Proc. WOOT*, 2017, pp. 1–10.

[18] D. Kennedy, J. O'gorman, D. Kearns, and M. Aharoni, *Metasploit: The Penetration Tester's Guide*. San Francisco, CA, USA: No Starch Press, 2011.

[19] S.-K. Kim, E.-T. Jang, H. Park, and K.-W. Park, "Pwnable-sherpa: An interactive coaching system with a case study of pwnable challenges," *Comput. Secur.*, vol. 125, Feb. 2023, Art. no. 103009.

[20] S.-K. Kim, E.-T. Jang, and K.-W. Park, "Toward a fine-grained evaluation of the pwnable CTF," in *Proc. Int. Conf. Inf. Secur. Appl.* Jeju Island, (South) Korea: Springer, 2020, pp. 179–190.

[21] K. Kujanpää, W. Victor, and A. Ilin, "Automating privilege escalation with deep reinforcement learning," in *Proc. 14th ACM Workshop Artif. Intell. Secur.*, Nov. 2021, pp. 157–168.

[22] D. Lea and W. Gloger, "A memory allocator," 1996. [Online]. Available: https://gee.cs.oswego.edu/dl/html/malloc.html

[23] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *Programming Languages and Systems*. Berlin, Germany: Springer, 2019, pp. 244–265.

[24] R. Li, B. Zhang, J. Chen, W. Lin, C. Feng, and C. Tang, "Towards automatic and precise heap layout manipulation for general-purpose programs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–15.

[25] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Shellcode_IA32: A dataset for automatic shellcode generation," 2021, *arXiv:2104.13100*.

[26] P. Liguori et al., "EVIL: Exploiting software via natural language," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2021, pp. 321–332.

[27] D. Liu et al., "Pangr: A behavior-based automatic vulnerability detection and exploitation framework," in *Proc. 17th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun./12th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2018, pp. 705–712.

[28] D. Liu et al., "From release to rebirth: Exploiting thanos objects in Linux kernel," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 533–548, 2023.

[29] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "VuRLE: Automatic vulnerability detection and repair by learning from examples," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2017, pp. 229–246.

[30] R. Maeda and M. Mimura, "Automating post-exploitation with deep reinforcement learning," *Comput. Secur.*, vol. 100, Jan. 2021, Art. no. 102108.

[31] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics, Syst. Demonstrations*, 2014, pp. 55–60.

[32] M. Miller. (2018). *Microsoft Remote Code Execution Vulnerability Statistics: 2016–2017.* [Online]. Available: https://t.co/qCtoDrTufj

[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

[34] D. Mu et al., "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 919–936.

[35] S. Mumtaz, C. Rodriguez, B. Benatallah, M. Al-Banna, and S. Zamanirad, "Learning word representation for the cyber security vulnerability domain," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2020, pp. 1–8.

[36] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2012, pp. 335–346.

[37] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Proc. Workshop Program. Lang. Anal. for Secur.*, Oct. 2017, pp. 25–35.

[38] J. Roney, T. Appel, P. Pinisetti, and J. Mickens, "Identifying valuable pointers in heap data," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2021, pp. 373–382.

[39] Y. Shoshitaishvili et al., "Rise of the HaCRS: Augmenting autonomous cyber reasoning systems with human assistance," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 347–362.

[40] H. Sidhpurwala. (2019). *Hardening ELF Binaries Using Relocation Read-Only (RELRO).* [Online]. Available: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro

[41] I. Takaesu, "Deepexploit: Fully automatic penetration test tool using machine learning," 2018. [Online]. Available: https://github.com/13o-bbr-bbq/machine_learning_security/tree/master/DeepExploit
[42] P. Team, "PaX non-executable pages design & implementation," 2003. [Online]. Available: http://pax.grsecurity.net/
[43] Y. Wang et al., "Revery: From proof-of-concept to exploitable," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1914–1927.
[44] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards automated heap Feng shui," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1647–1664.
[45] Z. Wang, H. Wang, H. Hu, and P. Liu, "Identifying non-control security-critical data in program binaries with a deep neural model," 2021, *arXiv:2108.12071*.
[46] S. Wi, J. Choi, and S. K. Cha, "Git-based CTF: A simple and effective approach to organizing in-course attack-and-defense security competition," in *Proc. USENIX Workshop Adv. Secur. Educ.*, 2018, pp. 1–9.
[47] Wikipedia. (2023). *Address Space Layout Randomization*. [Online]. Available: https://en.wikipedia.org/wiki/Address_space_layout_randomization
[48] Wikipedia. (2023). *Position-Independent Code*. [Online]. Available: https://en.wikipedia.org/wiki/Position-independent_code
[49] S. Xu and Y. Wang, "BofAEG: Automated stack buffer overflow vulnerability detection and exploit generation based on symbolic execution and dynamic analysis," *Secur. Commun. Netw.*, vol. 2022, pp. 1–9, Jun. 2022.
[50] G. Yang, X. Chen, Y. Zhou, and C. Yu, "DualSC: Automatic generation and summarization of shellcode via transformer and dual learning," 2022, *arXiv:2202.09785*.
[51] Y. Younan, W. Joosen, and F. Piessens, "Efficient protection against heap-based buffer overflows without resorting to magic," in *Proc. ICICS*. Cham, Switzerland: Springer, 2006, pp. 379–398.
[52] B. Zhang, J. Chen, R. Li, C. Feng, R. Li, and C. Tang, "Automated exploitable heap layout generation for heap overflows through manipulation distance-guided fuzzing," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 4499–4515.
[53] Z. Zhao, Y. Wang, and X. Gong, "HAEPG: An automatic multi-hop exploitation generation framework," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2020, pp. 89–109.
[54] (2021). *Binary Ninja*. [Online]. Available: https://binary.ninja/
[55] (2019). *Empire*. [Online]. Available: https://github.com/EmpireProject/Empire
[56] *The GNU C Library (GLIBC)*. [Online]. Available: https://www.gnu.org/software/libc/
[57] (2019). *Changelog for GLIBC 2.29*. [Online]. Available: https://abi-laboratory.pro/index.php?view=changelog&l=glibc&v=2.29
[58] (2021). *Shellphish's Automated Exploitation Engine, Originally Created for the Cyber Grand Challenge*. [Online]. Available: https://github.com/angr/rex
[59] (2016). *DARPA Cyber Grand Challenge (CGC)*. [Online]. Available: https://www.darpa.mil/program/cyber-grand-challenge
[60] (2021). *CTFtime: All About CTF*. [Online]. Available: https://ctftime.org/
[61] (2021). *CTFtime: CTF Events in 2021*. [Online]. Available: https://ctftime.org/event/list/?year=2021&online=1&format=1&archive=true&restrictions=-1
[62] (2023). *Common Vulnerabilities and Exposures*. [Online]. Available: https://cve.mitre.org
[63] (2021). *Exploit Database*. [Online]. Available: https://www.exploit-db.com/
[64] (2021). *CVE-2021–3156: Sudo Privilege Escalation Vulnerability*. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156
[65] (2017). *0CTF 2017: BabyHeap2017*. [Online]. Available: https://ctftime.org/task/3584
[66] (2017). *ASIS CTF Quals: CaNaKMgF*. [Online]. Available: https://ctftime.org/task/3831
[67] (2017). *CSAW CTF 2017: Auir*. [Online]. Available: https://ctftime.org/task/4639
[68] (2018). *PicoCTF 2018: Sword*. [Online]. Available: https://github.com/Mithreindeir/ctf-writeups/tree/master/pico-ctf2018/sword
[69] (2019). *Balsn CTF 2019: Securenote*. [Online]. Available: https://ctftime.org/task/9387
[70] (2019). *FireShell CTF 2019: Babyheap*. [Online]. Available: https://ctftime.org/task/7482
[71] (2019). *HSCTF 6: Hard Heap*. [Online]. Available: https://ctftime.org/task/8739
[72] (2019). *SECCON 2019 Online CTF: One*. [Online]. Available: https://ctftime.org/task/9535
[73] (2019). *TJCTF 2019: House of Horror*. [Online]. Available: https://bitbucket.org/ptr-yudai/writeups/src/master/2019/TJCTF_2019/House_of_Horror/
[74] (2019). *TokyoWesterns CTF 5th 2019: MI*. [Online]. Available: https://ctftime.org/task/9150

Dandan Xu received the B.E. degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2018. She is currently pursuing the Ph.D. degree in information security with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. Her research interests include vulnerability detection and software security.

Kai Chen (Member, IEEE) received the Ph.D. degree from the University of Chinese Academy of Sciences in 2010. He joined the Chinese Academy of Sciences in January 2010. He became an Associate Professor in September 2012 and became a Full Professor in October 2015. His research interests include software analysis and testing, smartphones, and privacy.

Miaoqian Lin is currently pursuing the Ph.D. degree in information security with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. Her research interests include software security.

Chaoyang Lin received the B.S. degree from the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China, in 2022. His research interests include the IoT security, mobile security, and system security.

XiaoFeng Wang (Fellow, IEEE) is the Associate Dean for Research and a James H. Rudy Professor of Luddy School of Informatics, Computing and Engineering, Indiana University at Bloomington. His research focuses on systems security and data privacy with a specialization on security and privacy issues in mobile and cloud computing, cellular networks and intelligent systems, and privacy issues in dissemination and computation of human genomic data.