SUNFLOWER: Enhancing Linux Kernel Fuzzing via Exploit-Driven Seed Generation

Qiang Zhang*, Yuheng Shen†, Jianzhong Liu†, Yiru Xu†, Heyuan Shi‡, Yu Jiang† and Wanli Chang*
* College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
† KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
‡ School of Electronic Information, Central South University, Changsha, China
*{zhangqiang9413@126.com, wanli.chang.rts@gmail.com},†jiangyu198964@126.com,
†{shenyh20,liujz21,xuyr21}@mails.tsinghua.edu.cn, ‡shiheyuan@csu.edu.cn

***Abstract*—The Linux kernel is the foundation of billions of contemporary computing systems, and ensuring its security and integrity is a necessity. Despite the Linux kernel's pivotal role, guaranteeing its security is a difficult task due to its complex code logic. This leads to new vulnerabilities being frequently introduced, and malicious exploits can result in severe consequences like Denial of Service (DoS) or Remote Code Execution (RCE). Fuzz testing (fuzzing), particularly Syzkaller, has been instrumental in detecting vulnerabilities within the kernel. However, Syzkaller's effectiveness is hindered due to limitations in system call descriptions and initial seeds. In this paper, we propose SUNFLOWER, an initial corpus generator that leverages existing exploits and proof-of-concept examples. SUNFLOWER is specifically designed to meet the critical requirements of industry deployments by facilitating the construction of a high-quality seed corpus based on bugs found in the wild. By collecting and analyzing numerous real-world exploits responsible for kernel vulnerabilities, the tool extracts essential system call sequences while also rectifying execution dependency errors. This approach addresses a pressing industry need for more effective vulnerability assessment and exploit development, making it an invaluable asset for cybersecurity professionals. The evaluation shows that, with the help of SUNFLOWER, we find a total number of 25 previously unknown vulnerabilities within the extensively tested Linux Kernel, while by augmenting Syzkaller with SUNFLOWER, we achieve a 9.5% and 10.8% improvement on code coverage compared with the Syzkaller and Moonshine.**

I. INTRODUCTION

The Linux kernel is a vital component in modern computing systems, powering millions of servers and devices worldwide. As the last line of defense for computer systems, the operating system's kernel demands robustness and resilience, which are critical to preserving the system's integrity and safety. If any kernel vulnerability is exploited maliciously, it can lead to detrimental results, such as Denial of Service (DoS) or Remote Code Execution (RCE).

Given the size and complexity of a modern OS kernel, it is difficult to ensure that bugs are not introduced with new features or fixes for other bugs, notwithstanding existing or dormant legacy bugs. Currently, many projects aim to test the Linux kernel continuously to mitigate this issue. Fuzz testing [1], [2], a.k.a. "fuzzing", is a dynamic software testing technique known for its exceptional ability to identify vulnerabilities. It has successfully detected numerous vulnerabilities across various software applications, including operating system kernels. As of current, Google's Syzkaller kernel fuzzer [3] is the state-of-the-art in kernel fuzzing. Its inner workings are similar to many other state-of-the-art fuzzers, where it uses system call descriptions (in Syzlang [4]) written by kernel experts and pseudo-randomly generates a series of system calls, a.k.a. a system call sequence, then runs the generated inputs during kernel execution, and uses code coverage as guidance to identify inputs that trigger new kernel behavior and save them for further mutation. To date, Syzkaller has successfully uncovered thousands of critical vulnerabilities within the Linux kernel and is widely adapted into different kernel vendors' CI/CD pipelines. Given Syzkaller's popularity, many research works aim to augment specific aspects of its workflow. Taking Moonshine [5] as an example, it extracts execution traces from real-world programs and converts them into Syzlang, thereby improving the overall fuzzing efficiency.

Despite Syzkaller's perceived effectiveness, we find that a typical kernel fuzzing campaign tends to stagnate in coverage statistics and bug detection within a few days. Its overall effectiveness is limited from further exploration and exploitation mainly due to the limited information that system call descriptions provide. The current pool of Syzlang is written by the Linux kernel developers and maintainers, totaling around four thousand Syzlangs. These descriptions mainly convey the syntax information of system calls to the fuzzer, whereas the semantic information is partial and qualified at best. Information such as particular argument combinations or system call chains to trigger a complex kernel state are left to the fuzzer to blindly try and find, with only code coverage as guidance and randomized methods for system call and argument generation.

The quality and quantity of the Syzlang have a profound influence on the Syzkaller's fuzzing efficiency, as it determines the code exploration ability of Syzkaller. Some works [6], [7] aim towards extracting more detailed and specific system call descriptions for Syzkaller; due to the immense complexity of the Linux kernel, no amount of effort can produce a set of system call descriptions that can entirely encompass the kernel's state space. In contrast, a more pragmatic approach is to provide Syzkaller with a set of high-quality initial seeds, which extend the initial execution traces of the fuzzer broad and deep within the kernel's code, allowing for the

fuzzer to easily overcome difficult situations and increase its effectiveness easily. Previous research in this field, such as Moonshine, uses real-world system call traces to produce an initial seed set. The drawback of doing so is that the produced seeds generally do not add much meaningful information to the fuzzing process and are not well-suited to be integrated into continuous testing scenarios. We observe that using system call traces from real-world *Proof-Of-Concept* (PoC) exploits program obtained from Common Vulnerabilities and Exposures (CVE) vulnerabilities allows one to produce a seed set with system call sequences and argument values that actually trigger a complex kernel state, therefore allowing the fuzzer to cover more code effectively, and potentially detect more bugs. This approach is well-suited for integration into a continuous testing environment, as CVE vulnerabilities are produced actively. However, to apply such an approach, we need to address the following challenges.

First, exploits programs do not provide sufficient information regarding their runtime requirements, nor do they run out-of-the-box. To be able to extract the relevant system call sequences from the exploits programs, we need to design a unified initial corpus construction mechanism. By far, these programs in the wild are fragmented and separate across multiple open-source platforms like Google Forum and GitHub. This widespread distribution complicates the collection process. Furthermore, these exploits need to be converted to Syzlang; however, given that these exploits stem from different kernel versions and vendors, they often feature operations or system calls that either lack support in mainstream Linux kernels or demand specific configurations, adding to the intricacy of kernel fuzzing.

Second, the extracted sequences need to be matched to their respective kernel versions and runtime environments, therefore requiring mechanisms that automatically prune, test, and integrate an incoming system call sequence into a running fuzzing campaign. In reality, the extracted corpus contains operations across different kernel versions and vendors. Therefore, some operations may no longer be compatible with the current Linux kernel. As a result, during the testing, the kernel may encounter frequent crashes, severely hampering the overall fuzzing performance. To this end, we need a mechanism that is capable of refining the provided system call sequence into a valid seed for fuzzing, thus further boosting the fuzzer's fuzzing efficiency.

In this paper, we propose SUNFLOWER, which utilizes exploits in the wild to boost the fuzzing performance. In detail, we first collect exploits from open-source communities, especially, we focus on those exploits that were not reported by Syzkaller, as they contain sequences of system calls or argument assignments that Syzkaller does not produce. Then, we construct the initial corpus to incorporate these exploits into the Syzkaller. Specifically, we dispatch all exploits to corresponding versions of Linux kernel to extract the execution trace, and then we convert them into Syzlang. Last, during the testing, we monitor the execution of Syzkaller, identify those operations within the initial corpus that frequently introduce false-positive crashes to the fuzzer, and automatically remove them. Using SUNFLOWER, we can effectively integrate real-world vulnerabilities into traditional kernel fuzzers' testing process, enabling them to cover previously uncovered code sections and boost the overall fuzzing performance.

We implemented SUNFLOWER and applied its generated initial seed set to Syzkaller. Our evaluation shows that SUNFLOWER helps to find a total number of 25 previously unknown vulnerabilities across extensively tested versions of Linux kernel, of which 3 have been assigned CVE IDs. Furthermore, with the help of SUNFLOWER, Syzkaller can cover on average 9.5% more code, while Moonshine gains an increase in average code coverage by 10.8%. In summary, this paper makes the following contributions:

- We propose to use the exploits in the wild to boost the performance of kernel fuzzing.
- We propose SUNFLOWER, which can incorporate collected exploits with Syzkaller's Syzlang specification and can automatically update the corpus during testing.
- We found a total number of 25 previously unknown bugs within different versions of Linux kernel, with 3 CVE number assigned. To promote open-source research, we provide the source code of freely SUNFLOWER[1].

II. BACKGROUND AND MOTIVATION

A. Kernel Fuzzing

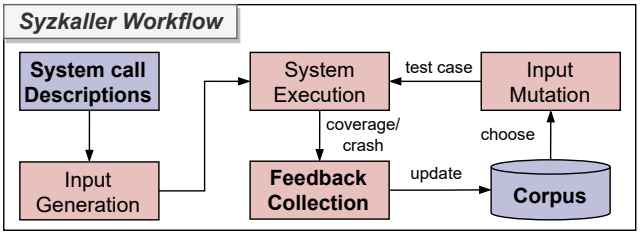


Fig. 1: Overall Workflow of Syzkaller. Using the system call descriptions as input, Syzkaller generates a set of system call sequences as input for the target kernel to execute. Then, based on the feedback information such as code coverage and crashes, Syzkaller saves those interested inputs to the corpus for further input mutation.

Given the critical importance of the Linux kernel, many works [8]–[12] have attempted to integrate fuzzing into kernel testing to enhance its security and stability. For instance, KAFL [13] leverages specific hardware features to implement binary instrumentation and a rapid snapshot recovery mechanism, thereby facilitating effective testing of the target kernel. However, a notable limitation of KAFL is its reliance on generating inputs entirely at random, which, given the kernel's stringent input specifications, results in relatively constrained fuzzing performance. Syzkaller, as the state-of-the-art kernel fuzzer, is known for its remarkable performance and has a workflow illustrated in Figure 1. Central to Syzkaller's

[1] SUNFLOWER is available at: https://github.com/zzqq0212/Sunflower.

approach is the utilization of Syzlang as its fuzzing input. Syzlang is a domain-specific language meticulously crafted to articulate system calls in a structured format, typically provided by kernel experts. This structured approach enables Syzkaller to generate sequences of system calls, emulating real-world operations and probing for unexpected system behaviors, thereby enhancing its ability to identify vulnerabilities. Upon detecting an input that triggers a system crash or uncovers new code coverage, Syzkaller saves it to a test case queue, referred to as the *corpus*. By issuing a higher mutation and execution probability to the inputs within the initial corpus, Syzkaller is able to conduct more thorough and comprehensive testing.

While Syzkaller is known for its effectiveness in kernel fuzzing, during the general fuzzing campaign, despite the complexity of the kernel's code logic, Syzkaller's fuzzing progress often plateaus within a few days. This limitation is primarily attributed to the circumscribed information provided by system call descriptions. The existing Syzlangs, approximately four thousand in number, are crafted by Linux kernel developers and maintainers. The number of Syzlang determines the Syzkaller's code exploration ability to a large extent. Some works try to boost the Syzkaller's performance. For example, Moonshine harnesses operations emitted during real-world program execution as input and employs a system call distillation algorithm to generate test cases of superior quality, thereby enhancing fuzzing efficiency.

B. Motivating Examples

The initial corpus holds significant weight in determining the overall performance of fuzzing, as it directly influences the fuzzer's ability to explore code. However, given the intricate complexity of the Linux kernel, crafting a set of Syzlang that fully encapsulates the kernel's state space is an impossible task. A more viable strategy involves equipping Syzkaller with a robust set of initial seeds. We find that a wealth of exploits programs represent some of the kernel's in-depth error states; therefore, they contain execution traces capable of triggering complex kernel states, where the Syzkaller can hardly generated based on its original Syzlang set. By leveraging such traces into the initial corpus, Syzkaller is enabled to probe more extensively into the kernel code and state space, therefore achieving more effective code coverage and potentially uncovering more unknown vulnerabilities.

Here, we demonstrate the Syzlang and the corresponding execution trace of an exploits program found by SUNFLOWER as Figure 2. The relevant repair patch can refer to Listing 1. In short, when a user attempts to remove an extent status and subsequently inserts a new one with the intention of merging it, a use-after-free bug happens. As we can see from the above figure, this bug is hidden deep within the kernel's code logic; triggering this bug requires using a set of subset operations, including creating a file, conducting write operations, and deallocating the file. Traditional kernel fuzzers like Syzkaller may find it hard to find such a bug, as randomly generated input can hardly generate such complex input that

```c
// Exploits Source Code
int main(void) {
    intptr_t res = 0;
    memcpy((void*)0x20000000, "./file0\000", 8);
    res = syscall(__NR_creat, 0x20000000ul, 0ul);
    if (res != -1)
       r[0] = res;
    memcpy((void*)0x20000040, "./file0\000", 8);
    res = syscall(__NR_creat, 0x20000040ul, 0ul);
    if (res != -1)
       r[1] = res;
    memcpy((void*)0x20000080, "threaded\000", 9);
    syscall(__NR_write, r[1], 0x20000080ul,
        0xfb3ful);
    syscall(__NR_fallocate, r[0], 0x10ul, 0xful,
        0x8000ul);
}

// Corresponding Strace
r0 = creat(&(0x7f0000000000)='./file0\x00', 0x0)
r1 = creat(&(0x7f0000000040)='./file0\x00', 0x0)
write$cgroup_type(r1, &(0x7f0000000080), 0xfb3f)
fallocate(r0, 0x10, 0xf, 0x8000)
```

Fig. 2: The Syzlang and corresponding execution traces for a previously unknown bug found by SUNFLOWER.

contains such logic. This highlights the necessity of the initial corpus. Moreover, we noticed that there are lots of kernel bugs and their corresponding PoC programs separated in different open-source platforms that are not found and reported in the by Syzkaller. By collecting and utilizing them as the initial fuzzing corpus, we can enable Syzkaller to generate test input with more semantic information and cover those parts that are hard to cover by original Syzkaller, thereby greatly augmenting the Syzkaller's fuzzing efficiency and allow test deeper into the kernel's code logic. However, to utilize such programs, we may need to face the following two challenges.

Firstly, an automated and unified mechanism is required to construct the initial corpus efficiently, enabling the execution of exploits program and extraction of the execution trace from these programs. However, publicly available exploits programs are scattered across diverse open-source platforms like Google Forum and GitHub, complicating the collection process. Moreover, converting exploits programs into Syzlang poses challenges due to the diversity in their originating kernel versions, configurations, and vendors. This may introduce system calls that are either unsupported or necessitate specific configurations in mainstream Linux kernels. Secondly, an automatic mechanism capable of checking the compatibility of each Syzlang within the constructed corpus during testing and dynamically removing Syzlang instances that are not compatible with the target kernel. Specifically, as mentioned above, the extracted corpus contains Syzlang from different kernel versions and vendors; consequently, some Syzlang and their operations may not align with the current Linux kernel. As a result, the kernel may encounter frequent crashes, introducing many false positives to the fuzzer and significantly impeding overall fuzzing performance. Therefore, a mechanism that avoids operations incompatible with the kernel and enhances fuzzing efficiency is imperative.

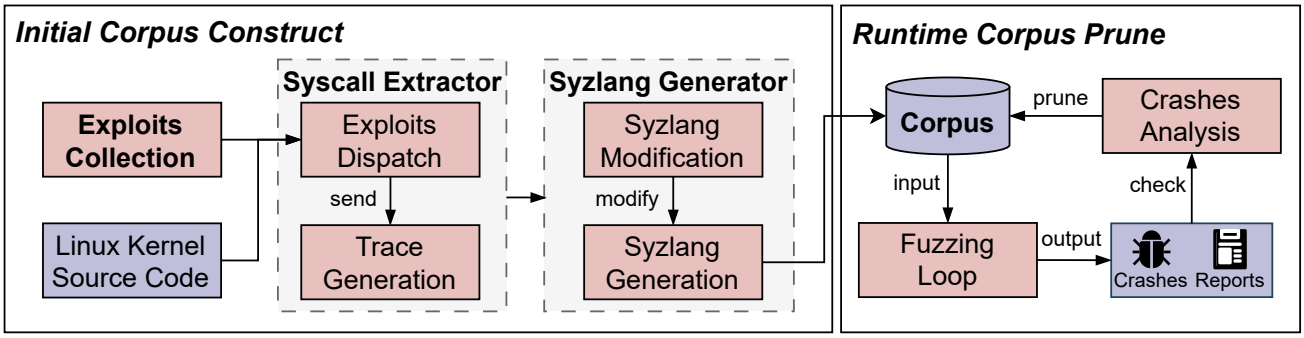


Fig. 3: Overall Diagram of SUNFLOWER. It first collects PoC programs from open-source communities, and using the syscall extractor, SUNFLOWER dispatches programs to different Linux kernels and generates execution traces for each program. Based on traces, SUNFLOWER modifies those traces that may fail to pass the semantic checks and convert them into the Syzlang as the initial corpus. Last, SUNFLOWER monitors the reported bugs and crashes, and filters those Syzlang that causes false-positive.

III. DESIGN

We proposed SUNFLOWER, an initial corpus generator that utilizes existing exploits programs. By collecting the exploits programs in the wild, SUNFLOWER can effectively convert them into Syzkaller's initial corpus, thereby improving the fuzzing performance. The detailed workflow is demonstrated as Figure 3. As we can see from the figure, SUNFLOWER consists of two phases, namely, the initial corpus construct and the runtime corpus prune. Within the initial corpus construct phase, we first collect existing PoC programs from different open-source communities. Then, using the system call extractor, we try to compile, execute, and extract their execution trace under different versions of the Linux kernel. Based on the execution trace, SUNFLOWER generates the Syzlang as the initial corpus and modifies that Syzlang that does not conform to Syzkaller's requirements. During the runtime corpus prune phase, SUNFLOWER checks the crash report and filters out those Syzlang sequences that may frequently introduce false-positive crashes during the fuzzing process, thereby improving the fuzzing efficiency.

A. Initial Corpus Construct

To extract high-quality seeds as the initial corpus, we first propose the initial corpus construction. This requires us to collect large amounts of exploits programs from diverse sources, efficiently analyze their severity, bug type, and if containing any reproduction program. Utilizing these programs, we extract their respective system call traces and generate relevant Syzlang, forming the initial corpus for Syzkaller.

*1) **Exploits Data Preparation**:* Currently, exploits programs for kernel vulnerabilities are widely distributed and come from multiple sources. At the same time, most of these programs are submitted and disclosed by individual kernel security researchers. The majority of these exploits programs are disclosed in some personal blogs, Linux kernel mailing lists, discussion Groups, etc. In most cases, the coding style and programming language of these programs also show huge differences due to the different personal styles of kernel security experts. It is difficult to quickly identify and collect these programs using unified interface specifications. Therefore, efficiently collecting and organizing these PoC programs is a challenging problem.

TABLE I: List of linux kernel exploits that collected from NVD, Github, and Google.

Exploit Types	NVD	Github	Google	Count
privilege escalation	35	22	11	68
logic error	42	5	9	56
memory corruption	7	3	3	13
out-of-bounds	2	16	10	28
use-after-free	4	7	12	23
double free	4	5	2	11
null point defer	14	7	9	30
memory leak	20	3	8	31
Total	128	68	64	260

Data Collection. To address the aforementioned challenge of collecting exploits programs, we have curated public disclosure sources from open-source communities, such as open-source sites, active security experts' personal blogs, and Linux subscription RSS sites. Our focus is directed toward vulnerabilities that can jeopardize the security and integrity of the system, particularly those within the kernel's critical modules. Table I illustrates the programs we have amassed. To better collect and analyze the exploits programs, for each collected exploits and their corresponding program, we scrutinize their execution behaviors and potential consequences. we first use a Python script to help automatically scan the designated open source platform and to acquire relevant vulnerability information, such as bug type, bug description, and exploits programs. Then, upon collecting the above information, we classify them into different bug types, such as memory corruption and memory leak issues, and we filter that collected bug that may have missing exploits program.

In detail, we have collected a total of 260 exploits programs in the wild, with a specific emphasis on the National Vulnerability Database (NVD), GitHub, and Google Forum. These platforms are widely recognized in open-source communities

and contain a substantial number of Linux kernel vulnerabilities. Additionally, based on the collected information, we summarized eight types of kernel bugs. Our attention is particularly drawn to highly dangerous bugs capable of triggering system erroneous states, such as memory error, privilege leaks, and data races. By leveraging the collected programs, we can generate high-quality inputs and cover those code sections that Syzkaller may have failed to test before, thereby improving the fuzzing effectiveness.

*2) **Syscall Extractor**:* Once we have collected the exploits programs, to construct the initial corpus, we first need to obtain their `strace` log and extract the corresponding execution trace. However, collected programs are distributed across different kernel versions with different compilation and execution dependencies, such as glibc version requirements, kernel configurations, and compile options. It is hard to simply compile and execute a collected program at a randomly given Linux kernel version and get its execution trace. Therefore, we need an automatic approach that can help SUNFLOWER find the most suitable Linux kernel for the collected programs to execute and extract the execution trace.

To efficiently bridge the gap between incompatible kernel runtime environments with the exploit execution requirements, we propose to use a dispatcher that manages multiple targets Linux kernels with different versions and configurations to extract the execution trace. Specifically, we choose different kernel versions from Linux 3.x to the latest Linux 6.x as the extraction target. Furthermore, Syzkaller will conduct certain kernel configuration checks before testing. All target kernels are compiled with Syzkaller's check configurations enabled. If a binary is compilable and executable, we will get the relevant `strace` log and send it back to the collector at the host.

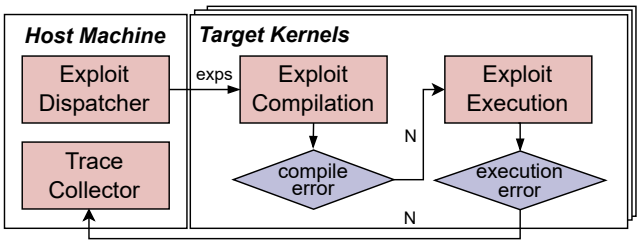


Fig. 4: Diagram of Trace Generation. The trace generation works at both the host machine and different versions of target kernels. The dispatcher sends collected exploits to different target kernels for compilation and execution till the execution trace can extracted and collected by the collector.

The overall workflow is demonstrated as Figure 4. As we can see, the extraction process works at both the target kernel and the host machine. In the host machine, the dispatcher is in charge of sending collected exploits programs to the target kernel, and the collector is in charge of collecting the execution trace. The dispatcher manages multiple Linux kernel, each with a different kernel version and configuration. After initiating the dispatcher, we boot the target kernel. Within the kernel, once it receives the programs, it first tries to compile the collected programs; if the compilation is successful, it then executes the program and monitors the execution trace with `strace`; if the program has any compile error or execution error, SUNFLOWER will simply drop them and send to the next target kernel, till we can compile and run the program.

*3) **Syzlang Generation**:* After obtaining the strace output log from the exploits program using SUNFLOWER, the next step is to construct the corpus by extracting and converting the execution trace into Syzlang. This process is divided into two primary steps. Initially, the output log is scrutinized to eliminate any undesired information that could cause the trace to fail Syzkaller's semantic checks. This is crucial for correcting malformed traces. Then, we can convert the execution trace into desired Syzlang.

Algorithm 1: Syzlang Generation Algorithm

Input: $SyscallTable$: System Call Table
Input: $SyscallRegex$: Regex For System Call
Input: $TracesLog$: Original Strace Output Log
Output: $SeqSpec$: Extracted Trace Log

Procedure *IsVaildTrace(SyscallTable, log)*
  $Flag \leftarrow$ FALSE
  for $syscall \in SyscallTable$ **do**
    if $log.contain(syscall)$ & $log.match(SyscallRegex)$ **then**
      $Flag \leftarrow$ TRUE
      return TRUE
  return $Flag$
Procedure *GenSPEC(SyscallTable, TraceLog)*
  TraceSeq $\leftarrow \emptyset$
  SeqSpec $\leftarrow \emptyset$
  for $line \in TraceLog$ **do**
    if *IsVaildTrace*$(SyscallTable, line)$ **then**
      $TraceSeq$.append$(line)$
    else
      $TraceLog$.remove$(line)$
  $SeqSpec \leftarrow$ ConvertToSpec$(TraceSeq)$
  return $SeqSpec$

We employ the Syzkaller's `syz2trace` toolset. This toolset is adept at transforming real-world execution traces into an instantiated version of Syzlang, complete with arguments. However, a significant challenge arises during this conversion. Some system calls do not pass the Syzkaller's semantic checks successfully. The root of this issue lies in the strace output, which contains extensive debug information, including details about error operations or return values. While this information aids in manual inspection, it hinders the Syzkaller's trace lexer's ability to process the traces correctly. To overcome this hurdle, we remove incompatible statements prior to generation. A set of regular expressions (regex) is applied to represent the execution information. When traces containing elements incompatible with our regex expressions are encountered, we omit the problematic parts. This step ensures that all execution traces adhere to Syzkaller's semantic standards. Following this rigorous filtering and conversion process, the final corpus is generated. This approach enables the acquisition of an initial

corpus that maintains a balance between clarity and comprehensive detailing, crucial for exploring kernel vulnerability exploits with SUNFLOWER.

The detailed description of our approach can be referred to as Algorithm 1. Specifically, in the first phase of trace conversion, we inspect each line in the execution trace log, as shown in lines 11 to 14. In Particular, we focus on two types of flags: whether the log contains any system call and whether the log matches our system call regular expression, as shown in lines 3 to 4. If we check that the current line of the log is indeed an execution trace, we will return a true; else, we return a false, and we will add the current execution log to the *TraceSeq*; else, we remove that from the *TraceLog* as shown in lines 3 to 6, and line 12 to 15. Once we have obtained the TraceSeq, we can call `syz2trace` to convert them into Syzlang.

B. Runtime Corpus Prune

Upon obtaining the Syzlang, we augment it to Syzkaller as the initial fuzzing corpus. However, since the collected exploits programs originated from different versions of Linux kernels, some of the extracted Syzlang may be incompatible with the target kernel due to incompatible glibc requirements or unsupported system calls, thereby introducing many meaningless crashes during the fuzzing and significantly affecting the fuzzing performance. Hence, we need to monitor the execution, update the corpus, and remove those Syzlang that are not supported on the target kernel.

Algorithm 2: Corpus Prune Algorithm

```
   Input: new_crash
   Input: crash_count_map
   Input: crash_list
   Output: corpus
1  id = hash(new_crash)
2  if id ∉ crash_count_map then
3      crash_count_map[id] = 1
4  else
5      crash_count_map[id] += 1
6  if crash_count_map[id] >
     MAX_CRASH_COUNT then
7      if new_crash.description ∈ crash_list then
8          corpus.remove(new_crash.prog)
```

The overall corpus prune process is depicted in Algorithm 2. Specifically, upon the acquisition of a new crash, we first generate a unique identifier, id, utilizing a hash function. If id is absent in the `crash_count_map`, it is added with an initial count of 1 (line 1). Conversely, if it is already present, the count is incremented (line 2). When the count for an id exceeds a predefined threshold, denoted as `MAX_CRASH_COUNT`, verification is conducted to ascertain if the description of the new crash resides in the `crash_list`. If it does, the corresponding program of the new crash is excised from the `corpus`. The `MAX_CRASH_COUNT` is summarized from our empirical practice and can dynamically adjust during testing.

IV. IMPLEMENTATION

We implemented SUNFLOWER using Golang and Python, in addition to some modifications on Syzkaller. First, SUNFLOWER uses a Python script to extract exploits from open-source communities. Then, to collect the execution trace, SUNFLOWER implements a dispatcher that operates on both the host and guest machines. The host machine is responsible for starting different versions of the Linux kernel, while the guest machine is tasked with executing each exploit and collecting the execution trace during the process. Later, based on the execution trace, SUNFLOWER modified parts of the Syzkaller component to generate Syzlang, adapting it to older versions of the Linux kernel. Finally, we modified the mutation module of Syzkaller to automatically filter out seeds that frequently trigger false-positive crashes during the fuzzing process.

SUNFLOWER by far has been integrated into Shuimuyulin's continuous fuzz testing pipeline, the *Wfuzz* Robot, to provide fuzzing with high-quality test cases. Alterations within a particular segment of the kernel instigate the CI/CD (Continuous Integration/Continuous Deployment) process, thereby serving as a pivotal enhancement in the kernel fuzzing procedure. This integration has been employed in the OS header vendor, UOS, to facilitate continuous fuzzing of its product.

V. EVALUATION

To thoroughly assess the effectiveness of SUNFLOWER in augmenting kernel fuzzing capabilities, we conducted a set of experiments with an emphasis on whether SUNFLOWER can facilitate kernel fuzzer in achieving higher code coverage and detecting more previously unknown vulnerabilities. We begin by showcasing previously unknown bugs discovered by SUNFLOWER. Subsequently, we delve into specific case studies of these unique bugs and discuss potential associated risks. Lastly, we compare SUNFLOWER's code coverage performance against Syzkaller and Moonshine, highlighting its proficiency in exploring a broader range of execution paths and deeper kernel states.

A. Experiment Setup

The experiments were conducted on a server with a 128-core CPU and 32 GiB of memory running Linux as the host kernel. We chose Linux kernel v5.15, v6.1, v6.3.4, and v6.5 as our test kernel targets. In detail, the Linux v6.5 is the latest release version when we were conducting experiments. Each version of the kernel uses the same compilation configuration, while KCOV [14] and KASAN [15] options are enabled to collect code coverage and detect memory errors. When setting up the KCSAN [16] configuration, the same configuration is used in the control test.

We used the original Syzkaller and Moonshine for comparison against Syzkaller augmented with SUNFLOWER. Each experiment maintained consistent parameters, including QEMU [17] setups, system call descriptions, and more. Specifically, to strictly control the computational resources, we started all experiments simultaneously and distributed the resources evenly, including 2 cores and 2 GiB of memory

TABLE II: SUNFLOWER has discovered 25 previously unknown vulnerabilities, among which are 3 CVEs. The first column shows the module in the Linux kernel that contains the vulnerability, and the rest of the columns illustrate the kernel versions, the locations of the bugs, the type of the bugs, and their description, respectively.

Modules	Versions	Locations	Bug Types	Bug Descriptions
fs/ext4	v6.5	ext4_es_insert_extent	use-after-free	incorrect read task access causes use-after-free error
arch/x86/kvm	v6.3.4	kvm_vcpu_reset	logic error	kvm virtual cpu reset process causes error
net/8021q	v6.5	unregister_vlan_dev	logic error	invalid opcode at net/8021q/vlan.c causes error
fs/dcache	v6.3.4	__d_add	data race	contention with read operation at __d_add function
net/ipv4	v6.3.4	__netlink_create	memory leak	unreleased memory objects causes leaks
net/ipv6	v6.5	ip6_tnl_exit_batch_net	logic error	unregistering process of network devices results error
mm/slab	v6.3.4	cache_grow_begin	memory leak	unreferenced object causes memory leak
net/can	v6.5	raw_setsockopt	deadlock	circular lock acquisition results in a deadlock
fs/proc	v6.3.4	proc_pid_status	data race	data race invoking tasks causes system hang
mm/memory	v6.3.4	copy_page_range	data race	unsynchronized access to shared data by threads results in error
fs/dcache	v6.3.4	dentry_unlink_inode	data race	file unlinking operations results error
fs/proc	v6.3.4	task_dump_owner	data race	unsynchronized thread access to shared data leads to error
fs/f2fs	v6.3.4	f2fs_truncate_data_blocks_range	out-of-bounds	incorrect read operation results out-of-bounds error
fs/buffer	v6.3.4	submit_bh_wbc	logic error	incorrect write operation causes invalid opcode error
fs/xfs	v6.3.4	xfs_btree_lookup_get_block	logic error	invalid memory access results error
drivers/block/aoe	v6.3.4	aoecmd_cfg	logic error	jump labels operation causes kernel hang error
mm/mmap	v6.3.4	do_vmi_munmap	logic error	incorrect instruction execution causes kernel panic
fs/udf	v6.3.4	udf_finalize_lvid	use-after-free	invoking deprecated mand mount option results use-after-free bug
drivers/block	v6.5	sock_xmit	use-after-free	incorrect memory deallocation causes the use-after-free error
kernel/sched	v6.3.4	run_rebalance_domains	logic error	incorrect scheduling operation causes RCU (Read-Copy-Update) error
block/bdev	v6.3.4	blkdev_flush_mapping	dead lock	incorrect filesystem operation causes error
mm/swap	v6.3.4	folio_batch_move_lru / folio_mark_accessed	data race	unsynchronized thread access to shared data leads to error
lib/find_bit	v6.3.4	_find_first_bit	data race	unsynchronized thread access to shared data causes error
mm/filemap	v6.3.4	filemap_fault / page_add_file_rmap	data race	inconsistent read and write operations results data race
fs/ext4	v6.3.4	ext4_do_writepages / ext4_mark_iloc_dirty	data race	unsynchronized thread access to shared data causes race contention

for each virtual machine. All tools use the same version of the Syzlang description. To accurately assess SUNFLOWER's effectiveness and negate any architectural biases in the results, each set of experiments is repeated five times, and each experiment is executed over a period of 48 hours, and we calculate the average values as the results.

B. Bug Finding

To demonstrate the effectiveness of SUNFLOWER in finding real-world vulnerabilities, we deploy SUNFLOWER, Syzkaller, Moonshine to conduct fuzzing testing on the selected mainstream kernels.

In our experiments, SUNFLOWER uncovered over 131 vulnerabilities in total. This includes 25 previously unidentified vulnerabilities, of which 3 were assigned CVE IDs(CVE-2023-40791, CVE-2023-40792, CVE-2023-40793). Even with extensive testing by Syzkaller, Moonshine, and other kernel fuzzers using substantial computing resources, these vulnerabilities remained undetected. Notably, the majority of these vulnerabilities are situated within the core logic of the Linux kernel, encompassing modules like file systems, network, and memory management. Detailed descriptions can be found in Table II, which includes the specific kernel module, kernel version, location, bug type, and bug description.

The ability of SUNFLOWER to identify previously unknown vulnerabilities is primarily due to the fact that the system call sequences in the corpus are sourced from previously verified real-world kernel exploits, which were not discovered by any automated kernel fuzzing tools. Leveraging Syzkaller's mutation and scheduling strategies, SUNFLOWER is capable of mutating system call sequences that are more valuable than those generated by automated approaches. This enhances code coverage and directs the exploration toward discovering vulnerabilities in more linux kernel error-prone modules.

C. Case Study

In this section, we'll explore the root cause and analyze the potential implications of these detected vulnerabilities.

Case Study 1. Listing 1 depicts a use-after-free vulnerability discovered in the ext4 file system module of Linux kernel v6.5, which has been fixed by corresponding maintainers.

This bug is in function `ext4_es_insert_extent()` within the ext4 file system, responsible for adding information to an inode's extent status tree. When this function is invoked, memory is allocated to the es1 and es2 variables. As the kernel program progresses, it reaches the `__es_remove_extent()` function (Line 4), which calls `ext4_es_insert_extent()` function to insert es1 to es tree. However, a problem arises at Line 10, which calls `ext4_es_insert_extent()` to insert es2 to es tree. Internally, this action prompts the `ext4_es_try_to_merge_right()` function to free es1 variable by `ext4_es_free_extent()`. Consequently, by the time the program reaches Line 24, where it attempts to check if `es1->es_len` is zero, it triggers the use-after-free, as the memory space of es1 has been freed before but is reaccessed. The patch addresses this issue by checking the status of es1 or es2 immediately after invoking either the `__es_remove_extent()` or `__es_insert_extent()`. This ensures that the use-after-free doesn't arise if either es1 or es2 is freed.

Case Study 2. Listing 2 illustrates a logic vulnerability discovered in the memory management module of Linux kernel v6.5. This vulnerability is triggered by SUNFLOWER and has been fixed.

```
--- a/fs/ext4/extents_status.c
+++ b/fs/ext4/extents_status.c
@@ -878,23 +878,21 @@ void
     ext4_es_insert_extent(struct inode *inode,
     ext4_lblk_t lblk,
  err1 = __es_remove_extent(inode, lblk, end,
      NULL, es1);
  if (err1 != 0)
    goto error;
+ if (es1 && !es1->es_len)
+   __es_free_extent(es1);

  err2 = __es_insert_extent(inode, &newes, es2);
  if (err2 == -ENOMEM &&
      !ext4_es_must_keep(&newes))
    err2 = 0;
  if (err2 != 0)
    goto error;
+ if (es2 && !es2->es_len)
+   __es_free_extent(es2);

  if (sbi->s_cluster_ratio > 1 &&
      test_opt(inode->i_sb, DELALLOC) &&
     (status & EXTENT_STATUS_WRITTEN ||
      status & EXTENT_STATUS_UNWRITTEN))
    __revise_pending(inode, lblk, len);
-
- /* es is pre-allocated but not used, free it.
    */
- if (es1 && !es1->es_len)
-   __es_free_extent(es1);
- if (es2 && !es2->es_len)
-   __es_free_extent(es2);
```

Listing 1: This patch fixes a use-after-free vulnerability in ext4 file system. Line 24 attempts to access `es1->es_len` after this variable's memory space has been freed in Line 10.

```
struct vm_area_struct *vma_merge(struct
    vma_iterator *vmi, struct mm_struct *mm,
    struct vm_area_struct *prev, ...)
{
      ...
  /* Verify some invariant that must be enforced
      by the caller. */
  VM_WARN_ON(prev && addr <= prev->vm_start);
  VM_WARN_ON(curr && (addr != curr->vm_start ||
      end > curr->vm_end));
  VM_WARN_ON(addr >= end);
      ...
}
static int mbind_range(struct vma_iterator *vmi,
    struct vm_area_struct *vma, struct
    vm_area_struct **prev, ...)
{
      ...
  merged = vma_merge(vmi, vma->vm_mm, *prev,
      ...);
  if (merged) {
    *prev = merged;
    return vma_replace_policy(merged, new_pol);
  }
      ...
  *prev = vma;
  return vma_replace_policy(vma, new_pol);
}
SYSCALL_DEFINE4(set_mempolicy_home_node, ...)
{
      ...
  for_each_vma_range(vmi, vma, end) {
    old = vma_policy(vma);
              /* fix: update prev pointer here:
                  prev = vma */
    if (!old)
                    continue;
          ...
          err = mbind_range(mm, vmstart, vmend,
              new);
          ...
  }
  ...
}
```

Listing 2: The code snippet of a logic error. The function `set_mempolicy_home_node()` inconsistently bypasses `mbind_range()`, leading to potential issues in updating VMA pointers when no VMA policy is present.

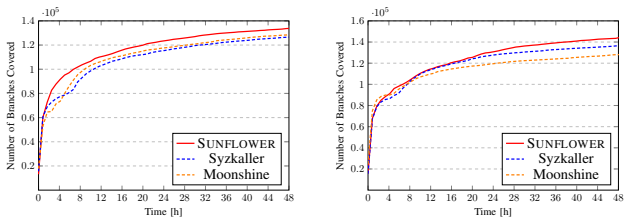
In the Linux kernel's memory management subsystem, SUNFLOWER identified an inconsistency in the behavior of the `mbind_range()` function. Specifically, the current implementations that utilize `mbind_range()` anticipate that it will not update the pointer referencing the preceding Virtual Memory Area (Line 19) through its internal sanity check (Line 5-7). Nevertheless, during our fuzzing, we observed that the `set_mempolicy_home_node()` function (Line 28-29) bypasses the invocation of `mbind_range()` in scenarios where a VMA policy is absent. To rectify this discrepancy, kernel maintainers propose the corresponding fix: the pointer to the preceding VMA should be updated before proceeding with the iteration over the VMAs in cases where the policy is not present (Line 27, `prev = vma`). This adjustment ensures consistency and meets the expectations of the functions relying on `mbind_range()`. The Syzlang SUNFLOWER extracted contains deep code logic and, therefore, allows Syzkaller to cover such error states.

D. Coverage Improvement

To demonstrate the effectiveness of SUNFLOWER in exploring a broader range of execution paths and deeper kernel states, we present the code coverage statistics in comparison with existing state-of-the-art kernel fuzzers, including Syzkaller and Moonshine. Four versions of the Linux kernel are chosen for the experiment, including v5.15, v6.1, v6.3.4, and v6.5. We selected Linux v6.5 because it is the latest kernel version at the time of our experiments. At the same time, we choose v6.3.4 to compare the coverage improvement effect that exists between different state kernel versions, while Linux v6.1 and Linux v5.15 serve as the two featured release versions widely adopted by numerous distributions. The Qemu simulation environments utilized by all fuzzer tools maintain the same parameter configurations, such as CPU core number, memory storage size, and so on. Finally, each testing campaign undergoes five repetitions, and we present the ultimate average value obtained over a forty-eight-hour period.

We evaluated code coverage statistics for the Linux kernel using three seed scenarios: generated exploit seeds by SUNFLOWER, no seeds, and seeds produced by Moonshine based on Syzkaller. Syzkaller can test the Linux kernel through a definition of a set of pseudo-system calls. These pseudo-system calls can connect the information interaction between user mode and kernel mode, thereby collecting coverage data of the accessed kernel modules. To reduce the randomness of the experiment, we ran five groups of experiments for each kernel version and set the coverage statistics to be sampled every ten seconds during each experiment. The average value of the sampled data in several executions of each Syzkaller was calculated as comparative experimental data.

Figure 5 illustrates the comparison of branch coverage among Syzkaller, Moonshine, and SUNFLOWER. Specifically, all tools show growth in the first 24 hours. SUNFLOWER outperforms Syzkaller and Moonshine by varying degrees in terms of code coverage over the same timeframe. However, post this period, while Syzkaller and Moonshine's coverage



(a) Coverage on Linux v5.15 (b) Coverage on Linux v6.1

(c) Coverage on Linux v6.3.4 (d) Coverage on Linux v6.5

Fig. 5: Branch coverage of Syzkaller, Moonshine, and SUNFLOWER on the Linux versions 5.15, 6.1, 6.3.4, and 6.5. After 48-hours of testing, SUNFLOWER consistently demonstrated the highest coverage for all versions.

growth slows and nears saturation, SUNFLOWER continues to progress at a faster rate. Throughout the forty-eight-hour experiment, we observed that SUNFLOWER consistently achieved the highest coverage rate. This is because SUNFLOWER utilizes a corpus initialized with kernel vulnerability exploits, allowing the fuzzer to reach deeper kernel code more efficiently. In contrast, Syzkaller starts with a random sequence of system calls, often missing deeper vulnerability logic. Although Moonshine leverages system call sequences from real programs, many of these sequences fail to trigger kernel vulnerabilities. Consequently, SUNFLOWER performs better in improving coverage and exploring the kernel code space.

TABLE III: Coverage statistics of SUNFLOWER, Syzkaller, and Moonshine on Linux over 48 hours. SUNFLOWER achieves higher coverage on all kernel versions.

Version	SUNFLOWER	Syzkaller	Moonshine
v5.15	**133744**	126564(+5.7%)	128546(+4.0%)
v6.1	**144336**	136442(+5.8%)	128120(+12.7%)
v6.3.4	**136948**	127333(+7.6%)	126635(+8.1%)
v6.5	**154365**	129709(+19.0%)	130576(+18.2%)
Overall	**142348**	**130012(+9.5%)**	**128469(+10.8%)**

Table III provides a detailed breakdown of the branch coverage statistics for Syzkaller, Moonshine, and SUNFLOWER over a forty-eight hours period. Compared to Syzkaller, SUNFLOWER achieves coverage improvements of 5.7%, 5.8%, 7.6% and 19.0% on experiments with versions 5.15, 6.1, 6.3.4 and 6.5, respectively. Against Moonshine, SUNFLOWER shows enhancements of 4.0%, 12.7%, 8.1% and 18.2% for the same versions. Overall, SUNFLOWER achieves 9.5% and 10.8% higher coverage than Syzkaller and Moonshine, respectively. In essence, our data underscores that SUNFLOWER, by leveraging past kernel vulnerability exploits as corpus seeds, consistently outperforms in achieving higher coverage.

VI. Related Works

A. Fuzz Testing

Fuzz testing [1], [18], [19], also referred to as fuzzing, stands out as an automated program testing technique, has located a wide range of vulnerabilities among operating systems, communication protocols, and diverse libraries. A fuzzer is designed to navigate through the code space of target programs using randomly generated test cases and employing a variety of sanitization techniques. This approach enables it to adeptly identify a range of bug types, from memory corruption to concurrency issues, thereby enhancing software reliability and security. Fuzzers can generally be categorized into two primary types: generation-based fuzzers and mutation-based fuzzers [20], [21]. Generation-based fuzzers, such as the well-known protocol fuzzer Peach, leverage predefined input specifications to guide the generation of high-quality test cases. For instance, Peach [22] utilizes protocol specifications (pit files) that encapsulate detailed data and state transition descriptions, enabling it to generate highly structured input to thoroughly test various protocol implementations. Conversely, mutation-based fuzzers, like the most famous and state-of-the-art fuzzer AFL [23], utilize different metrics as guidance, such as code coverage, to steer the fuzzing process. For instance, it prioritizes inputs that trigger new code coverage, allocating them a higher likelihood of being executed and mutated in subsequent iterations. Numerous works have explored various mutation strategies and execution methods to enhance the efficacy of this fuzzing approach [24], [25].

Given the proven effectiveness of fuzzing and the paramount importance of kernel security, numerous researchers have ventured into amalgamating fuzzing with kernel testing to fortify kernel robustness [26]. Syzkaller, a state-of-the-art kernel fuzzer, employs Syzlang as fuzzing input to emulate authentic execution workloads, thereby testing various kernel modules, and has successfully unearthed thousands of kernel bugs over the years. The complexity of kernel logic has spurred innovations like HFL [27], which melds symbolic execution and fuzzing. By calculating paths that are challenging to reach, HFL can more comprehensively cover code sections that are seldom executed. Moonshine, on the other hand, collects real-world execution traces from different programs and proposes a distillation algorithm to construct the initial corpus, enabling Syzkaller to generate a higher-quality payload. Moreover, Healer [28] utilizes relation learning to enhance testing efficiency, analyzing the relationship between two adjacent system calls within a system call sequence to generate input that probes deeper into the kernel's code logic. KSG [29] uses the ebpf technique to extract kernel system call's argument type and value constraint, using this extract information to generate corresponding Syzlang as the initial corpus to test kernel's specific modules. Additionally, there have been concerted

efforts to augment Syzkaller's testing efficiency. For instance, Horus [30] enhances data transfer processes by using the shared memory to transfer fuzzing-related data like coverage and test input, thereby eliminating the transmission overhead inherent in Syzkaller's RPC communication mechanism.

Different from previous work in terms of kernel fuzzing, SUNFLOWER proposes using previously detected bugs as the initial corpus; this allowed Syzkaller to generate more complex input to test deep into the kernel's code logic and be capable of testing those error-prone code sections more frequently, thereby improving the overall fuzzing efficiency.

B. Corpus Generation

The corpus, embodying a collection of high-quality test cases, holds paramount importance in fuzzing, serving as a pivotal element that can be utilized both before and during the fuzzing process to facilitate more thorough testing of the target program [31]. Constructing a corpus before fuzzing typically involves collecting payloads that adhere to the input specifications of the target program [32], [33].

In the realm of compiler fuzzing, Csmith [34] stands out as a notable tool that generates random C programs, which are utilized to test compilers. by ensuring that generated programs are well-defined according to the C standard, providing a robust mechanism for identifying compiler bugs without manual triage of crashes to determine whether they expose genuine compiler bugs or merely undefined behaviors. Furthermore, the EMI [35], [36] strategically utilizes real-world C programs, transforming them into equivalent variants to facilitate differential testing across various compilers. Specifically, by generating these semantically identical but syntactically varied programs, EMI effectively exposes discrepancies and potential vulnerabilities in compiler behaviors, thereby enhancing compiler testing methodologies.

Despite that, both SUNFLOWER and the above work are designed to provide fuzzers with high-quality input. However, different from the compiler fuzzing that generates C programs as the initial corpus, SUNFLOWER mainly uses the collected C program as input and extracts their corresponding execution traces to form the Syzlang for Syzkaller.

VII. LESSON LEARNED

A. Initial Corpus Construction

Currently, the initial corpus constructed by SUNFLOWER, particularly when derived from actual vulnerabilities observed in the wild, holds substantial significance in kernel fuzzing. This corpus, enriched with practical and historically problematic data, serves a dual purpose: it acts as a robust input reservoir and guides the kernel fuzzer toward exploring and scrutinizing deeper, more complex logical structures that have a higher propensity to trigger vulnerabilities. Moreover, it can help the fuzzer to avoid exploring some superficial logic, thereby optimizing its path and conserving valuable computational resources. However, the current corpus we constructed by SUNFLOWER still suffers from version incompatibility, as PoC programs and their corresponding system call traces from older versions of Linux can no longer be compatible with current versions of the Linux kernel. Consequently, an approach to collecting exploits and converting them into a valid initial corpus becomes paramount, especially focusing on those that have eluded detection by tools like Syzkaller. In the future, we can convert these incompatible traces by extracting their execution trace, transferring the traces to newer versions of Linux, and then generating the corresponding corpus.

B. Kernel PoC program reproduction

In the process of using SUNFLOWER to construct the corpus, it is not an easy task to reproduce and analyze the kernel vulnerabilities, particularly those discovered in real-world scenarios, which are often fraught with complexities. This process requires identifying vulnerabilities and analyzing their callback traces. In our empirical research, we observed that reproducing vulnerabilities is made even more difficult by the diverse hardware configurations used in real-world deployments, each introducing unique variables. Moreover, the wide range of software environments and usage scenarios further complicates this already intricate task. Furthermore, diverse PoC programs showcase varying behaviors, and some provoke non-deterministic actions during the reproduction phase. Consequently, this complicates the consistent recreation of specific conditions that trigger particular bugs. At the same time, the lack of detailed information and context about these vulnerabilities also exacerbates the complexity of reproducing vulnerabilities. In future research, the work on reproducing and analyzing vulnerabilities can not only involve more technical research from the perspective of software and hardware environment adaptation but also detailed vulnerability disclosure documents, comprehensive version control, and robust industry collaboration. These factors will greatly reduce the complexity of reproducing and analyzing these vulnerabilities.

VIII. CONCLUSION

In this paper, we introduce SUNFLOWER, an initial corpus generator designed to boost kernel fuzzer performance by leveraging existing exploits programs as its corpus. Sourcing exploits from open-source communities, SUNFLOWER features a unique automatic version dependency dispatcher and a runtime corpus prune mechanism, allowing it to compile and execute enhanced seeds across varying version dependencies. Using SUNFLOWER, we can effectively integrate real-world vulnerabilities into traditional kernel fuzzers' testing process, enabling them to cover previously uncovered code sections and improve the fuzzing performance.

We evaluated the performance of SUNFLOWER on several Linux kernel versions. Compared to Syzkaller and Moonshine, SUNFLOWER enhances branch coverage by 9.5% and 10.8%, respectively. Furthermore, SUNFLOWER successfully found 25 previously unknown vulnerabilities, among which are 3 CVEs. The above results demonstrate the ability of SUNFLOWER to explore kernel code space, thus helping fuzzers discover previously unknown vulnerabilities.

REFERENCES

[1] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[2] L. McDonald, M. I. U. Haq, and A. Barkworth, "Survey of software fuzzing techniques."

[3] D. Vyukov and A. Konovalov, "Syzkaller: an unsupervised coverage-guided kernel fuzzer," 2015, https://github.com/google/syzkaller.

[4] ——, "Syzlang: System Call Description Language," 2015, https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[5] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 729–743. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[6] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 3262–3278.

[7] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3133956.3134069

[8] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 754–768.

[9] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *Proceedings of the 13th USENIX Conference on Offensive Technologies*, ser. WOOT'19. USA: USENIX Association, 2019, p. 8.

[10] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang, and W. Chang, "Rtkaller: State-Aware Task Generation for RTOS Fuzzing," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021. [Online]. Available: https://doi.org/10.1145/3477014

[11] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1643–1660.

[12] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning," in *USENIX Security Symposium*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:237057141

[13] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[14] SimonKagstrom, "Kcov," 2013, https://github.com/SimonKagstrom/kcov.

[15] Google, "Kernel address sanitizer," 2013, https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[16] ——, "Kernel concurrency sanitizer," 2013, https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[17] F. Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[18] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[19] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[20] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[21] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[22] M. Eddington. (2023) Peach fuzzer platform. [Online; accessed 07-October-2023]. [Online]. Available: http://www.peachfuzzer.com/products/peach-platform/

[23] lcamtuf, "American fuzzy lop," 2013, https://lcamtuf.coredump.cx/afl/.

[24] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 861–875.

[25] J. Fell, "A review of fuzzing tools and methods," *PenTest Magazine*, 2017.

[26] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. A. Sani, "Demystifying the dependency challenge in kernel fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 659–671.

[27] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.

[28] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, *HEALER: Relation Learning Guided Kernel Fuzzing*. New York, NY, USA: Association for Computing Machinery, 2021, p. 344–358. [Online]. Available: https://doi.org/10.1145/3477132.3483547

[29] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "{KSG}: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[30] J. Liu, Y. Shen, Y. Xu, H. Sun, and Y. Jiang, "Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures," *ACM Transactions on Software Engineering and Methodology*, 2023.

[31] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[32] X. Zhu, X. Feng, T. Jiao, S. Wen, Y. Xiang, S. Camtepe, and J. Xue, "A feature-oriented corpus for understanding, evaluating and improving fuzz testing," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 658–663.

[33] V. Vikram, R. Padhye, and K. Sen, "Growing a test corpus with bonsai fuzzing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 723–735.

[34] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[35] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[36] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, 2016, pp. 849–863.