AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities

Zheyue Jiang[1,¶], Yuan Zhang[1,¶], Jun Xu[2], Xinqian Sun[1], Zhuang Liu[1], Min Yang[1]
1: Fudan University, 2: University of Utah, ¶: co-first authors

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179286

***Abstract*—This paper studies the problem of cross-version exploitability assessment for Linux kernels. Specifically, given an exploit demonstrating the exploitability of a vulnerability on a specific kernel version, we aim to understand the exploitability of the same vulnerability on other kernel versions. To tackle cross-version exploitability assessment, automated exploit generation (AEG), a recently popular topic, is the only existing, applicable solution. However, AEG is not well-suited due to its template-driven nature and ignorance of the capabilities offered by the available exploit.**

In this work, we introduce a new method, *automated exploit migration* (AEM), to facilitate cross-version exploitability assessment for Linux kernels. The key insight of AEM is the observation that the strategy adopted by the exploit is often applicable to other exploitable kernel versions. Technically, we consider the kernel version where the exploit works as a reference and adjust the exploit to force the other kernel versions to align with the reference. This way, we can reproduce the exploiting behaviors on the other versions. To reduce the cost and increase the feasibility, we strategically identify execution points that truly affect the exploitation and only enforce alignment at those points. We have designed and implemented a prototype of AEM. In our evaluation of 67 cases where exploit migration is needed, our prototype successfully migrates the exploit for 56 cases, producing a success rate of 83.5%.

I. Introduction

Understanding the exploitability of a vulnerability is a standard method to assess its severity, which can benefit many downstream applications such as scheduling of patching. As of today, the primary approach to exploitability understanding is to craft an exploit against the target vulnerability manually. A common issue of manually-crafted exploits is that they often target specific software versions, failing to provide information about the other but also buggy versions.

Take the Linux kernel as an example. There are thousands of versions and derivatives of Linux kernel [1]. However, according to our preliminary study on Linux kernel exploits released in the past five years, around 80% of the exploits were designed for one particular version. This lack of cross-version capability can lead to many security consequences. For instance, downstream kernel vendors may turn down or delay the patch if the exploit cannot prove the exploitability on their customized kernels [2], [3], [4]. Motivated by such observations, this paper concerns cross-version exploitability assessment for Linux kernels: *given an exploit working on a specific kernel version, how to understand the exploitability on other versions*?

To tackle the problem of cross-version exploitability assessment, an existing line of solutions, automatic exploit generation (AEG) [5], [2], [3], [6], [7], [8], [4], [9], [10], is in principle applicable. Given a proof-of-concept (PoC) that can trigger a vulnerability, AEG attempts to extend the PoC to perform exploiting operations. In essence, AEG aims to solve a search problem: finding an execution context where the PoC turns the program into a controlled state. However, the searching space is often enormous, incurring a high computational complexity. To reduce the searching space, existing AEG solutions primarily follow templates summarized from historical attacks. For instance, AEG against use-after-free (UAF) [2], [8], [11] typically attempts to spray data after the free such that the use leads the execution to malicious behaviors like hijacked control flow.

To be applied to cross-version exploitability assessment, AEG can consider the available exploit as a PoC and convert it to a working exploit on the non-targeted versions. However, AEG, at its current stage, is probably not the best methodology for cross-version exploitability assessment because of two limitations. First, AEG solutions tend to only reuse the pieces in the exploit that are essential to activate the vulnerability and then generate all other pieces required for exploitation from scratch. This is undesired. On the one hand, generating the other pieces is not always feasible in particular when they require certain human intelligence. On the other hand, the generation of those pieces can involve highly complex operations, incurring a high cost. Second, due to its template-driven nature, whenever the vulnerability cannot be exploited following the expected templates, AEG fails. This is a crucial drawback for cross-version exploitability assessment because it entirely overlooks the exploitation strategies carried by the available exploit.

In this paper, we propose a new methodology, *automated exploit migration* (AEM), to support cross-version exploitability assessment for Linux kernels. The key insight of AEM is the observation that the strategy adopted by an exploit is generally applicable to various exploitable versions. Following this insight, AEM works by adjusting the exploit to migrate its exploitation capabilities to non-targeted versions. To realize the idea of AEM, we proposed a set of techniques driven by alignment-based analysis. Specifically, we consider the kernel version where the exploit succeeds as a reference, and adjust the exploit to force the execution on other kernel versions to align with the execution from reference. Once successful, the exploitation behaviors shall also emerge in those kernel versions.

DOI 10.1109/SP46215.2023.00034

To reduce the cost and increase the feasibility, we identify execution points pertaining to the exploitation goals and only enforce alignment at those points. Moreover, instead of aligning the identified execution points altogether, we recursively handle them one by one.

We have designed and implemented a prototype of AEM. In our evaluation with 90 cases where the original exploit fails to prove the exploitability, AEM can successfully migrate the exploit in 56 cases. Among the 34 failed cases, 23 are because the given exploit is in principle non-migratable. Excluding those cases, AEM achieves a success rate of 83.5%. On average, AEM only needs a few hours to migrate an exploit, presenting an efficiency comparable to or higher than human analysts. Moreover, AEM shows a qualitatively higher utility than existing AEG tools (which cannot handle any of the cases without human efforts).

In summary, we make the following contributions.

- We introduce the problem of cross-version exploitability assessment and propose the AEM methodology for Linux kernels. Compared to AEG, the principle of AEM is more suited for cross-version exploitability assessment.
- We design a set of new techniques to enable AEM for Linux kernels. The techniques are elaborately designed to accommodate AEM with high accuracy and low cost.
- We develop a prototype of AEM and our evaluation shows that the prototype is highly effective and efficient in migrating real-world Linux kernel exploits.

II. Overview

A. Motivation

As the most widely used open-source OS kernel, Linux kernel has evolved into various versions [12] and derivatives (e.g., Ubuntu and Android). Meanwhile, due to its rapid development and large codebase, thousands of vulnerabilities have been unveiled in the Linux kernel each year [13], [14]. Given limited resources, exploitability assessment plays a central role in prioritizing patch development and deployment for the more severe vulnerabilities.

When a vulnerability is assessed to be exploitable or not for Linux kernel, it is beneficial to test its exploitability on all affected versions and derivatives. However, due to the nature that the exploitation process heavily relies on knowledge about the kernel internals (e.g., structures of kernel objects, control flows of kernel functions), an exploit that succeeds on one kernel may not always succeed on other kernels. According to our preliminary study on Linux kernel exploits released in the past five years, around 80% of the exploits were only designed for one particular version (see §VI-A). This calls for automatic techniques to facilitate the exploitability assessment of vulnerabilities on cross-version kernels.

Target Users. Motivated by the above observation, this paper presents the first study on the cross-version exploitability assessment problem for Linux kernel. We believe our research shall benefit broad users in our community. First, with cross-version exploitability assessment, the defenders could check whether a reported exploit also works on other kernel versions/variants and alert the end-users of exploitable kernels to mitigate emerging attacks. Furthermore, based on the migrated exploits, defenders could leverage the migrated exploits to defend against attacks (e.g., synthesizing firewall signatures to stop exploit propagation). Moreover, no matter the users are security experts or not, they both benefit from our work. ❶ For security researchers who have kernel exploitation expertise, they could be benefited by saving their time in assessing the exploitability of different kernel versions. On the one hand, it is a time-consuming and labor-intensive process to manually craft exploits for different versions even with a working exploit for one version as reference. On the other hand, there are usually a lot of kernel versions to assess when a new exploit is reported. ❷ For those people lacking the expertise in kernel exploitation (e.g., security administrators, end-users), our techniques could enable them to assess the cross-version exploitability of a vulnerability on their target kernel versions. For example, downstream kernel vendors could prioritize the patch deployment based on the exploitability of a vulnerability on their kernel.

B. Running Example

We use a running example to illustrate how an exploit that succeeds on a kernel version but fails on another version and explain why existing works are ill-suited for cross-version exploitability assessment.

The Vulnerability: The vulnerability, shown in Figure 1(a), is a UAF affecting Linux no later than version 4.11. After the execution of line 19, a newly created netlink socket, `sock`, shall carry a reference counter of 2. If the follow-up attaching operation by `netlink_attachskb` fails, `sock`'s reference counter will be decreased twice by functions `netlink_attachskb` (line 21) and `netlink_detachskb` (line 26). This leads `sock` to be freed in `netlink_detachskb`. However, `sock` can still be accessed via invoking `sys_setsockopt` (line 8), causing a UAF. Specifically, the linked list with `wait` in `sock` as the head is dereferenced to pick the `next` node, and fields in that node are then used. For instance, the `func` field is called as a function pointer at line 14.

The Exploit for v4.1: Figure 1(b) demonstrates an exploit against the above vulnerability, targeting Linux v4.1. The exploit first invokes a free of `sock` (line 4) and then sprays the heap with data to override `sock` (line 6). After these operations, the `next` field in the `wait` member of `sock` points to a fake `__wait_queue` object where the function pointer `func` is picked by the adversary. Finally, the exploit activates the reuse of `sock` and hijacks the control flow to the address specified by the adversary in `func` (line 14).

Replay the Exploit on v4.4: We use the above exploit to test Linux v4.4 which is also affected by the vulnerability. However, we find that the exploit fails on v4.4. In particular, we observe the following two common causes for the failure of the exploitation that is prevalent among cross-version kernels.

- *Changes in Code:* Code changes happen frequently from version to version. When these changes get involved in

```
1  struct netlink_sock {..., wait_queue_head_t wait;}
2  struct wait_queue_head_t {spinlock_t lock; list_head task_list;}
3  struct __wait_queue { wait_queue_func_t func; struct list_head task_list;}
4  typedef int (*wait_queue_func_t)(...);
5  // heap spray
6  int sys_sendmsg(...) {...}
7  // dangerous use function
8  int sys_setsockopt(int fd, ...) {
9     sock = sockfd_lookup_light(fd, ...);
10    // sock is an alias to the sock freed at line 26, nlk is an occupied object by heap spray
11    struct netlink_sock *nlk = nlk_sk(sock->sk);
12    spin_lock_irqsave(&nlk->wait.lock, flags);
13    struct wait_queue_t *curr = container_of(
            nlk->wait.task_list.next, __wait_queue, task_list);
14    curr->func(curr, ..); // dangerous use => control flow hijack
15 }
16 // vulnerable function
17 void sys_mq_notify(... , const struct sigevent *notification) {
18    f = fdget(notification->sigev_signo);
19    sock = netlink_getsockbyfilp(f.file); //inc. refcount of sock
20    // attach skb to sock, if failed dec. refcount of sock
21    ret = netlink_attachskb(sock, ...);
22    if (ret == 1) goto out; // if failed, goto out
23    ...
24 out:
25    // dec. refcount of sock, sock->sk becomes a dangling pointer
26    if (sock) netlink_detachskb(sock, ...);
27 }
```

(a) Illustration of `CVE-2017-11176` (affecting Linux no later than v4.11).

```
1  int main(){
2     fd = sys_socket(AF_NETLINK, SOCK_RAW, 2);
3     // trigger the vulnerability => create a dangling pointer
4     sys_mq_notify((mqd_t)0x666, fd);
5     // occupy the freed netlink_sock object
6     heap_spray();
7     // trigger the use of dangling pointer => control flow hijack
8     sys_setsockopt(fd, ...);
9  }
10 void heap_spray(){
11    int sfd = sys_socket(AF_UNIX, SOCK_DGRAM, 0);
12    // forge a __wait_queue object in userland
13    struct u_wait_queue_t uwq;
14    uwq.func = 0xdeadbeef; // assign the code pointer
15    uwq.next = &(uwq.next);
16    uwq.prev = &(uwq.next);
17
18    char buf[1024]; // to occupy a netlink_sock object
19    memset(buf, 0x41, 1024);
20    // modify netlink_sock->wait.tasklist.next
21    *(unsigned long*)((char*)buf+0x308) = &(uwq.next);
22
23    struct msghdr msg;
24    msg.msg_control = buf;
25    // spray to occupy the freed netlink_sock object
26    for(i=0; i<10; i++)
27       sys_sendmsg(sfd, &msg, 0);
28 }
```

(b) Exploit against `CVE-2017-11176` [15] (designed for Linux v4.1).

Fig. 1. A running example of Linux kernel vulnerability and its exploit. Both the vulnerability and the exploit are simplified for the ease of understanding.

the exploitation process, the kernel can present inconsistent behaviors on different versions and the exploitation on some versions fails. Considering the above exploit on v4.1, `spin_lock_irqsave` won't block the execution since there is only one assignment for `slock` field, as shown in Figure 2(a). Thus, when the exploit overrides `nlk->wait.lock.slock` to `0x41` during heap spray, `spin_lock_irqsave` at line 12 of Figure 1(a) will return and the execution will activate the UAF at line 14. In contrast, as illustrated in Figure 2(a), `spin_lock_irqsave` on v4.4 hangs, given a non-zero value for the `slock` field. As a result, running the exploit against Linux v4.4, the execution won't even reach line 14 of Figure 1(a).

- *Changes in Data:* Going beyond code, the data used can also vary across kernel versions. In particular, the data structures often experience redefinition or adjustment. However, the exploit often assumes a specific memory layout that may only appear in the given kernel, which impedes the exploit from working on other kernel versions. As shown in Figure 1(a), the exploit needs to spray data to occupy the freed `netlink_sock` object such that the nested field `next` is overridden by a controlled value (see the `heap_spray` function in Figure 1(b)). On v4.1, `next` has an offset of `0x308` from the beginning of the freed `netlink_sock` object, guiding the exploit to place the fake `next` at that location. However, the offset on v4.4 becomes `0x2f8` due to the redefinition of `netlink_sock`. This leads the exploit to fail on v4.4 even the aforementioned `lock` is fixed.

Apply AEG: Automatic exploit generation (AEG) is the traditional theme for exploitability assessment. To assess the exploitability of the above vulnerability on v4.4, AEG could be

```
1  unsigned long spin_lock_irqsave(raw_spinlock_t *lock, ...) {
2     ...
3     lock->slock = 0;
4     ...
5  }
```

(a) `spin_lock_irqsave` in Linux v4.1

```
1  unsigned long spin_lock_irqsave(raw_spinlock_t *lock, ...) {
2     ...
3     if (virt_spin_lock(lock)) // lock->val == 0 ?
4        return;
5
6     while((val = smp_load_acquire(&lock->val.counter)) & 0xff)
7        // if &lock->val.counter != 0, the kernel pauses here
8        cpu_relax();
9     ...
10 }
```

(b) `spin_lock_irqsave` in Linux v4.4.

Fig. 2. Code changes affecting the exploit in Figure 1.

applied by using the exploit on v4.1 as input to generate a new working exploit for v4.4. However, we find that AEG is ill-suited for the cross-version exploitability assessment problem.

Using the above vulnerability as an example, FUZE [2] is the only AEG solution targeting UAF in Linux kernels. To work properly, FUZE needs to take the given exploit on v4.1 as a PoC and needs the PoC to follow a template where the use of the freed memory directly leads to exploiting primitives (e.g., using a piece of freed memory as a function pointer). Running the above exploit on Linux v4.4., the first use after free occurs on the nested field `lock` at line 12 of Figure 1(a). Internally, the use is only to check whether `lock` equals zero, which cannot work as a FUZE template to gain primitives. In addition, as the exploit overrides `lock` to all `0x41`, `spin_lock_irqsave`

will block the execution at line 12, preventing FUZE from catching the other uses at line 13 and line 14. As a result, FUZE will consider the PoC (the working exploit on v4.1) unusable and abandon it.

The example above demonstrates the first reason why AEG is ill-suited for cross-version exploitability assessment: the exploit may not offer the desired template. Going beyond, AEG can overlook the utilities of the exploit and incurs unessential, high complexity. Referring back to the above example, FUZE will search for proper objects to spray the heap, despite the exploit already carrying one (i.e., `msghdr`). This inevitably reduces the success rate while increasing the computational complexity.

C. Our Philosophy: AEM

In this paper, we propose a new philosophy to tackle cross-version exploitability assessment: *automated exploit migration* (AEM). Instead of "generating" a brand new exploit, AEM endeavors to migrate the given exploit to kernel versions where the exploit fails to work. Our key insight is that *the strategy employed by one working exploit shall generally apply to different kernel versions*. This insight is empirically validated in an evaluation of real-world exploits. Detailed setup of the study can be found in §VI-B. In summary, on 90 kernels where the given exploit does not work, 67 (74.4%) of them can actually be exploited in the same way as that exploit. The major cause behind the failures of the original exploits is *implementation changes across kernel versions* (see the two categories of changes summarized in §II-B) instead of the infeasibility of the exploitation strategy. More specifically, the implementation changes introduce disparity in the execution context desired by the exploit and interrupt the exploitation process. This inspires the core technical theme of AEM: *by adjusting the exploit to fit its desired execution context, AEM will enable the same exploiting primitives in other kernel versions*.

D. Problem Scope

With the same goal for exploitability assessment, AEG and AEM significantly differ in the methodology of creating a working exploit: AEG searches for an exploit from scratch, while AEM migrates a working exploit by reusing its exploitation strategy. In theory, AEG could be applied to all kinds of exploitability assessments. However, in practice, AEG is still far from perfect due to the large search space. In the context of cross-version exploitability assessment, AEM is more appropriate than AEG. Nevertheless, our AEM is not designed to be a silver bullet. Instead, it also has a bounded application scope, which we describe below.

Exploiting Primitive and Exploitation Strategy: In order to illustrate the problem scope, we first introduce the definition of exploiting primitive and exploitation strategy. In general, an exploiting primitive is an adversary-desired program state which provides extra capabilities beyond the original program functionality (e.g., hijacking control flows) [4], [16], [17]. Technically, the exploiting primitive usually results from security violations, e.g., memory corruption. The exploitation strategy generally represents the process of how a vulnerability is exploited to gain an exploiting primitive. In practice, the exploitation strategy usually binds to a specific exploit and consists of multiple separated techniques/steps. For example, an exploitation strategy of a UAF vulnerability includes finding the victim object to spray and finding a malicious use on the victim object to gain the primitive.

Migratable and Non-migratable Exploits: The success of AEM is built upon reusing the exploitation strategy of an exploit on a specific version and migrating it to a new version. If the strategy can be successfully reused on target kernel, we regard the exploit is *migratable* for this kernel. However, the implementation changes between two versions can be significant, making the original exploitation strategy infeasible. For example, when the data or the code that is necessary for reusing the exploitation strategy has been removed, the original exploitation strategy cannot be reused. In such cases, we call the given exploit as *non-migratable* on target kernel. Instead, a new exploitation strategy is required, which often involves new code state exploration, such as finding a new heap layout and expanding vulnerability capability. The exploitability assessment of such non-migratable cases is beyond the scope of AEM. In other words, AEM only focuses on migratable exploits.

Assumptions: This work targets cross-version exploitability assessment in the context of Linux kernels. First, we assume an exploit against one kernel version is available. Given the desired configurations, the exploit can accomplish the following stages: ❶ triggering the vulnerability; ❷ achieving exploiting primitives (e.g., control-flow hijacking and exploit-controlled memory access) that prepare the kernel to an adversary-controlled state; and ❸ performing attack behaviors (e.g., arbitrary code execution, mitigation bypass) with the primitives. Our second assumption is that the given exploit can trigger the vulnerability on a target kernel version. According to our study in §VI-B, we find that the collected 28 exploits can successfully trigger the vulnerabilities in all the affected 210 kernel versions. Besides, even if the vulnerability has not been triggered, it can be handled as an orthogonal problem [18].

To further bound our research, we focus on exploits enabled by memory corruptions, but we have no restrictions on the root causes. That is, the memory corruptions can be directly caused by memory errors like buffer overflow or indirectly caused by other issues like logic bugs. To be specific, we target the most two common memory corruption-enabled primitives, i.e., controllable memory access and control-flow hijacking. Besides, we only aim to migrate the exploit primitive, rather than the whole exploit, as triggering the exploiting primitive is already strong evidence of exploitability.

III. APPROACH

A. Challenges and Insights

The high-level idea of AEM — adjusting the exploit to create its desired execution context on a new kernel — is straightforward. However, realizing this idea is quite challenging. The first key design consideration is the perspective of adjusting the exploit.

In practice, kernel exploits are mostly C programs. As shown in our evaluation (see Table I), real-world exploits usually have hundreds of lines of C code or even over 1,000 lines. It is undesirable to consider all code lines for adjustment as many of them are exploitation irrelevant. In this paper, we consider *syscall*s and their arguments as the perspective to adjust the given exploit. The rationale is that syscalls are how an exploit interacts with the kernel towards exploitation.

Technical Challenges: By leveraging the syscalls as the perspective to adjust the exploit, there still remains two fundamental challenges in AEM, which are described below.

Challenge-I: Which syscalls to adjust? Real-world kernel exploits usually perform a large number of syscalls. As shown in our collected exploit dataset (see Table I), there are on average 1,800 syscalls during the exploitation flow. Moreover, a syscall may take several arguments which have complex data structures, especially for those syscalls that have rich functionalities (e.g., `ioctl`). Simply considering all syscalls and arguments to adjust will lead to enormous search space, incurring a high complexity.

Challenge-II: How to adjust these syscalls? Kernel exploitation usually requires accurate memory layout and code execution context. Hence, blind attempts to modify the arguments of target syscalls are hard to guarantee the success rate. An intuitive idea is to leverage the exploitation process on the successful kernel as guidance to run the adjustment. However, it remains unclear what guidance should be used and how such guidance can overcome the disparity incurred by implementation changes across kernel versions.

Insights: Recall that we focus on memory-related exploiting primitives and the key idea of AEM is to reuse the successful exploitation strategy on a kernel. This brings us the following insights. In principle, the exploiting primitives are an accumulative result of the sequence of memory operations leading to the primitive site. Therefore, by adjusting the memory operations in a *target kernel* (where the exploit does not work) to align with the *reference kernel* (where the exploit works), we shall trigger the same primitives in the target kernel. Inspired by this idea, we propose *primitive-centric, alignment-guided* exploit adjustment, which incorporates the following two more insights.

First, as described in *Challenge-I*, naively considering all memory operations can again cause complexity and infeasibility issues. Therefore, we propose to focus on primitive-centric memory operations for alignment based on two observations. ❶ Not every memory operation affects the exploiting primitives. Given a memory operation carrying no dependency, directly or indirectly, with the primitives, any adjustment on it won't influence the success (or not) of the primitives. Thus, we will identify and exclude such memory operations. ❷ Many memory operations are independent of user-space data, which cannot be adjusted by any means. Hence, we will also pinpoint and rule out this type of memory operation.

Second, as mentioned in *Challenge-II*, the guidance to align two memory operations from different kernels has to account for implementation disparities. Strict guidance, such as requiring the two operations to be identical (same address and same value), evidently does not work. We observe that the execution context can often sufficiently align two memory operations (i.e., verify that the two operations are the same one in different kernels). For better accuracy, we will consider both code context (e.g., the source code location) and data context (e.g., the type of the accessed data) of a memory operation.

B. Key Techniques

Following the insights, we design the first AEM method on the market. Our method is driven by two key techniques.

Technique-I: Primitive-centric Memory Abstraction: Given the reference kernel, we gather the memory accesses incurred by the exploit in the kernel space, followed by pruning those irrelevant to the exploiting primitives or independent of user-space data. The remaining memory operations are then organized as an EXPGRAPH where each vertex encodes a unique memory operation with annotated context information and the edges represent the dependency relations among different memory operations. Detailed definitions of the context information and the dependency are given in §IV-A.

Technique-II: Alignment-guided Exploit Adjustment: Once the EXPGRAPH is ready, we rerun the exploit on the target kernel and gather memory operations again. These memory operations are then traversed to identify those aligned ones to the topologically-sorted vertices of EXPGRAPH with the help of the context information. If no memory operation exists to align with an EXPGRAPH vertex, we consider the first unaligned node as the *migration target* and will adjust the exploit to achieve re-alignment. Technically, the exploit will first be concolically executed on the target kernel up to the memory operation matching the last aligned EXPGRAPH node. Then, symbolic execution will be started under the context (i.e. the data stored in memory and registers from previous execution) of the previous execution until a memory operation aligned with the migration target appears or the alignment is proven impossible. In the former case, the alignment and adjustment process will repeat all the way towards the exploiting primitives. In the latter case, we will conclude the migration is not doable and stop the exploration. To ensure fidelity of the alignment, we consider a conservative set of conditions when determining whether a memory operation and an EXPGRAPH vertex are aligned. More details are presented in §IV-B.

IV. DESIGN

The design of AEM incorporates our two key techniques and follows the workflow presented in Figure 3. Given an exploit and a reference kernel, AEM derives a new exploit to activate the same exploiting primitives in the target kernel. The rest of this section elaborates on the designs.

A. Primitive-centric Memory Abstraction

AEM starts with creating a primitive-centric memory abstraction for the reference kernel in the form of EXPGRAPH.

Instruction-level Full-system Tracing: Launching the exploit against the reference kernel, AEM records the sequence of

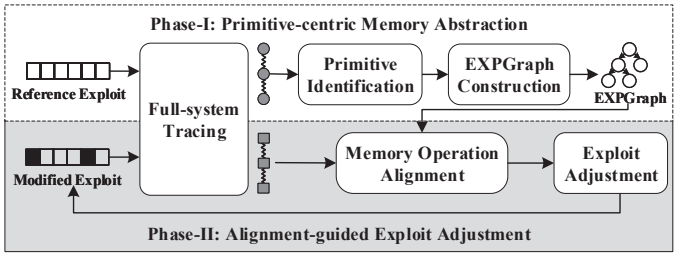


Fig. 3. The workflow of AEM consists of two phases. First, AEM analyzes the given reference exploit to summarize the key exploitation strategies into a EXPGRAPH. Second, AEM iteratively adjusts the given exploit to force its execution on the target kernel to align with the EXPGRAPH.

instructions executed in both the kernel space and the user space. To facilitate later analysis, the recorded information includes the address and the operands of each instruction. Given a memory-based operand, both the address and the value are logged. Further, the allocation and deallocation of kernel objects by standard functions (e.g., `__kmalloc`, `kfree`) are annotated at the entry and exit of the responsible functions. Instructions executed by different threads are separately recorded with their scheduling information. Once the recording is done, each instruction is further labeled with its affiliated event: which syscall, which interrupt, or user-space execution.

Exploiting Primitive Identification: On the recorded instruction trace, AEM runs a scan from the beginning. The goal is to identify instructions that represent exploiting primitives. In this paper, we focus on the two most common exploiting primitives enabled by memory corruptions: *control-flow hijacking* and *exploit-controlled memory access*.

Control-flow hijacking typically happens at an indirect control transfer whose target is obtained from a memory cell. To demonstrate the harm of the exploitation, the target often points to code capable of conducting malicious behaviors, commonly including stack pivoting gadget, payload in user space, and privilege manipulation functions like `prepare_kernel_cred`. Accordingly, AEM visits all indirect control transfer instructions in the trace and pinpoint those carrying a target falling into the above categories.

Exploit-controlled memory access has two general forms: reading from arbitrary addresses and writing arbitrary/constrained values to arbitrary addresses. When exploit-controlled memory access occurs, the data often has an inconsistent static type and run-time type. Precisely, the data type expected by the source code (*static type*) does not match the data type specified at the allocation site (*run-time type*). Details of how to obtain run-time type will be explained shortly. Inspired by the observation, AEM inspects each memory access and reports those with discrepant static type and run-time type as exploit-controlled memory accesses.

In general, an exploit may assemble multiple exploiting primitives. If detecting more than one primitive, AEM only considers the last one as it represents deeper execution and thus, more complete exploitation behaviors. Once the exploiting primitive is identified, AEM trims all the succeeding instructions.

EXPGRAPH Construction: From the instruction trace ending at the exploiting primitive, AEM extracts memory operations incurred by the kernel code and organizes them as an EXPGRAPH.

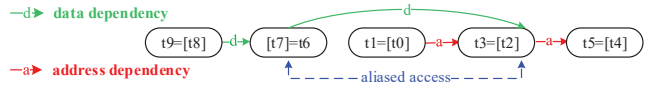
First, each memory operation is represented as a vertex with annotated context information. In the later exploit adjustment phase, the context information is used by AEM to identify aligned memory operations in the target kernel. To ensure fidelity of the alignment, we conservatively consider both code-level and data-level properties of a memory operation in the context information.

- **SYSCALL (/code)** records the syscall where the memory operation occurs, including both the syscall's number and order. Interrupts are considered special syscalls without arguments and handled in the same way.
- **Access class (/code)** indicates whether the memory operation is a read or a write.
- **Static type (/code)** refers to the object type of the accessed memory specified in the source code (e.g., `int` or `pointer`).
- **Address source (/data)** describes the source of the address to access. Possible sources include the result of an object allocation, constant, other memory operation, and user-space data (passed via syscall arguments).
- **Data source (/data)** describes the source of the data being accessed. Possible data sources are similar to address sources.
- **Address space (/data)** shows whether the memory operation falls into the kernel space or the user space.
- **Data value (/data)** is the value of the accessed data.
- **Runtime type (/data)** means the actual, runtime type of the accessed data. This is determined by the type specified to allocate the data.
- **Aliased access (/data)** tracks other memory operations that use the same address, including those happening in other syscalls.

Second, the vertices are connected with edges representing their dependency relations, which can be generally classified into *data dependency* and *address dependency*. Data dependency happens when the value loaded by a read operation propagates to the data stored by a write operation or the other way around. In contrast, address dependency arises when the value loaded by a read operation is used to form the address of another memory operation. One thing worthy of noting is that we ignore data dependency to a memory operation accessing the user space, since the data operated can be arbitrarily adjusted by the exploit without respecting any dependencies.

EXPGRAPH Pruning: Directly using the initial EXPGRAPH for alignment is often problematic as it involves many irrelevant memory operations. Accordingly, AEM further prunes two types of vertices and removes the edges from/to them.

First, many memory operations have zero effect on the exploiting primitive, which shall be excluded from consideration for later alignment. We pinpoint those memory operations by leveraging the initial EXPGRAPH. Specifically, we identify the vertices that cannot reach the exploiting primitive on the EXPGRAPH. In principle, such vertices carry no dependency



Operation / Properties	t9=[t8]	[t7]=t6	t1=[t0]	t3=[t2]	t5=[t4]
Line# in Fig.1(a)	line 6	line 6	line 11	line 13	line 14
SYSCALL	sendmsg	sendmsg	setsockopt	setsockopt	setsockopt
Static Type	__u64	__u64	socket(sk)	netlink_sock (wait.task_list.next)	__wait_queue (func)
Address Source	argument	allocation	global	t1=[t0]	t3=[t2]
Data Source	user	t9=[t8]	allocation	[t7]=t6	user
Address Space	user	kernel	kernel	kernel	user
Data Value	0x10850	0x10850	0xffff880235d77b00	0x10850	0xdeadbeef
Runtime Type	__u64	__u64, netlink_sock (wait.task_list.next)	socket (sk)	netlink_sock (wait.task_list.next)	u_wait_queue_t (func)
Aliased Access	-	t3=[t2]	-	[t7]=t6	-

Fig. 4. An example to show the EXPGRAPH for the exploit in Figure 1(b).

with the exploiting primitive, thus presenting no impact on the primitive.

Second, memory operations independent of user-space data cannot be adjusted in any way. Hence, they should also be ruled out for alignment. On the initial EXPGRAPH, this type of memory operation cannot be reached from any vertices representing user-space data, which gives us a straightforward approach to spot them.

EXPGRAPH for Running Example: To better illustrate the idea of EXPGRAPH, we construct the EXPGRAPH for our running example (see Figure 1) and show the simplified EXPGRAPH in Figure 4. The EXPGRAPH includes five key memory operations and encodes how the exploiting primitive is achieved. Specifically, the first two operations, `t9=[t8]` and `[t7]=t6`, are incurred when performing heap spray with syscall `sendmsg` (which is called by the exploit at line 27 of Figure 1(b)) and begins execution in the kernel at line 6 of Figure 1(a). The other three operations, `t1=[t0]`, `t3=[t2]`, and `t5=[t4]`, are activated by syscall `setsockopt` after the spray, which correspond to the kernel code execution at line `11`, `13-14` of Figure 1(a) respectively. Due to the UAF vulnerability, `[t7]=t6` and `t3=[t2]` access the same address, represented by their data dependency. The value written by `[t7]=t6`, propagated from and controlled by the user space, is later read by `t3=[t2]` and eventually used as the address of `t5=[t4]` to load the control transfer target. This way, the EXPGRAPH describes the exploiting trajectory until the primitive.

B. Alignment-guided Exploit Adjustment

Using the pruned EXPGRAPH as a reference, AEM then adjusts the exploit to reproduce the exploiting primitives in the target kernel. At the high level, AEM follows three steps.

- **Step-1:** AEM runs the exploit again on the target kernel and collects the instruction trace, using the full-system tracing described in §IV-A. Memory operations are then extracted from the trace and sorted based on their execution time.
- **Step-2:** AEM attempts to align the vertices on the EXPGRAPH with memory operations in the target kernel. After topologically sorting the EXPGRAPH, AEM picks the first vertex and scans the target kernel's memory operations to find an aligned point. For accuracy, the determination of an aligned point follows a conservative group of constraints on their context information (which will be explained shortly). If an aligned point is found, AEM moves to another vertex. Otherwise, it switches to the next step.
- **Step-3:** Given an unaligned EXPGRAPH vertex, AEM identifies the syscalls and their arguments in the exploit that affect the execution between the previous aligned point and the current unaligned vertex. This is followed by adjusting the arguments via symbolic execution to incur a memory operation in the target kernel to realign with the vertex. If the realignment fails, AEM quits and reports no successful migration. Otherwise, AEM checks whether the primitive emerges in the target kernel, and if not, AEM jumps back to *Step-2* to continue the alignment process.

1) Memory Operation Alignment

Given a vertex on the EXPGRAPH, AEM seeks an aligned memory operation in the target kernel. The idea is to visit each memory operation starting from the previous aligned point. If a memory operation satisfies the following constraints, we consider it as a new aligned point.

- **Constraint-1:** The memory operation and the vertex locate in the same syscall.
- **Constraint-2:** The memory operation and the vertex carry the same access class (both read or both write).
- **Constraint-3:** The memory operation and the vertex have identical static type and runtime type. In principle, two aligned memory operations should access the same piece of data. Using the concrete address to measure this property is intuitive but undesired because of the disparity in memory layout on different kernel versions. Alternatively, we consider runtime type as the indicator, which is more abstract and shall be stable across kernel versions.
- **Constraint-4:** The memory operation and the vertex share the same address source and data source. In the case where the address/data source is other memory operations, those memory operations must be also aligned. This constraint essentially implies that the dependency relations are aligned.

2) Exploit Adjustment

Given an EXPGRAPH vertex for which we cannot identify the aligned point in the target kernel, AEM first determines why the alignment fails. In the broad sense, there are two reasons:

- **R1:** the same memory operation happened in the target kernel but carried slightly different data-level properties;
- **R2:** the same memory operation is never executed in the target kernel.

To figure out the exact reason, AEM visits each memory operation succeeding the last aligned point in the target kernel. If a memory operation satisfies all the above alignment

constraints *except for the runtime type in Constraint-3*, we conclude reason R1 and otherwise R2. The rationale is that this set of constraints is in general conservative to tracking down the same memory operation. The exclusion of runtime type reflects the observation that the failure of alignment under R1 is typically a result of accessing unintended memory locations.

R1 Exploit Adjustment: This type of adjustment handles an unaligned EXPGRAPH vertex due to reason R1 (where the associated memory operation is executed in the target kernel). For simplicity of presentation, we denote the memory operation tied to the EXPGRAPH vertex as op_{ref} and the counterpart in the target kernel as op_{tar}. Technically, AEM needs to adjust the exploit such that op_{tar} carries the same runtime type as op_{ref}, which is completed in two phases.

Phase-I: Syscalls and arguments that affect op_{tar} are identified. This is done by backward and recursively tracing the data/address sources of op_{tar} until the entry points of syscalls. The syscalls and their arguments captured during the tracing are considered as targets.

Phase-II: The target syscall arguments, including the user-space data which the pointer arguments point to, are symbolized to support the concolic execution of the exploit on the target kernel. During this concolic execution, the runtime type of op_{ref} is encoded as an extra constraint on op_{tar}. Specifically, AEM first identifies all the alive objects by tracing the object allocations and deallocations. Then, the alive objects with the same runtime type as op_{ref} are selected, and the constraint requires that op_{tar} accesses one such object. Moreover, if op_{ref} represents the exploiting primitive, we add another constraint mandating the data read/written by op_{tar} and op_{ref} to have the same value. If the concolic execution can arrive at op_{tar} with the extra constraint(s) solved, it will derive a new exploit which can align op_{tar} and op_{ref}, and AEM will continue with the alignment process. Otherwise, AEM will report that the alignment cannot be completed.

R2 Exploit Adjustment: The second type of adjustment deals with an unaligned EXPGRAPH vertex due to reason R2 (where op_{ref} was never executed in the target kernel). The idea is to adjust the exploit to incur op_{ref} in the target kernel with all the alignment constraints satisfied, spanning three phases below.

Phase-I: AEM determines the counterpart of op_{ref} in the target kernel, notated as op_{tar}, by finding memory operations that share similar properties with op_{ref}. Specifically, it considers any memory operation carrying the following features as a candidate op_{tar}: (i) the memory operation appears after the previous aligned point on the control flow; (ii) the memory operation has the same access class and static type as op_{ref}; (iii) the source code corresponding to the memory operation is similar to that of op_{ref}.[1] Each of the candidate op_{tar} will be then processed in the next phase. For efficiency, we start with the candidate op_{tar} with the maximal source-code similarity to op_{ref} and stop once a candidate op_{tar} is successfully aligned. If no candidate could be aligned, the migration fails.

Phase-II: Given a candidate op_{tar}, AEM concolically executes the exploit on the target kernel until reaching the aligned point right before the op_{tar}. In this process, AEM fixes the syscalls and their arguments that are needed to satisfy the constraints on the execution path. Afterward, AEM symbolizes all the remaining arguments and switches to the last phase.

Phase-III: Under the context prepared by the above step, AEM continues executing the exploit on the target kernel in a symbolic manner. Once a path reaching the candidate op_{tar} is found, AEM stops the symbolic execution and concretizes the symbolized syscall arguments. At this point, AEM creates an exploit matching the need of *R1 Exploit Adjustment*, which can be done as described above.

Exploit Adjustment for Running Example: To clearly illustrate how the two types of adjustment facilitate exploit migration, we apply them to our running example in Figure 1. Referring back to the EXPGRAPH presented in Figure 4, the reference kernel (v4.1) and target kernel (v4.4) have two unaligned memory operations: `t3=[t2]` and `t5=[t4]`. AEM will align them in turn as follows.

Step-1: realigning `t3=[t2]` with R2 Exploit Adjustment: As pointed out in §II-B, `t3=[t2]` is not incurred in the target kernel. To align this operation, AEM first identifies the counterpart of `t3=[t2]` by locating all memory operations that (i) appear after `t1=[t0]`; (ii) access the `func` field of `__wait_queue` object; (iii) have source code similar to line 14 in Figure 1(a). Eventually, AEM picks the memory operation at line 14. In the follow-up step, AEM concolically executes the exploit until `t1=[t0]` (line 11), during which it fixes the arguments of all syscalls except for `sendmsg` to maintain the encountered conditions. Finally, AEM symbolizes the arguments of syscall `sendmsg` and enumerates the paths from line 11 to line 12. Once finding the path where `spin_lock_irqsave` returns, AEM concretizes the symbolic arguments to make the exploit reach line 14.

Step-2: realigning `t5=[t4]` with R1 Exploit Adjustment: After the realignment of `t3=[t2]`, AEM will still have to align `t5=[t4]`. In this case, AEM could locate operation `t5=[t4]` in the execution trace of the target kernel, but the data-level properties carried by it do not match the ones from EXPGRAPH. Though backward and recursively tracing the data/address source of `t5=[t4]`, AEM figures out that `t5=[t4]` is affected by syscall `sendmsg` and its arguments. Thus, AEM will symbolize those arguments to concolically execute the exploit on the target kernel. In this process, AEM adds two more constraints when `t5=[t4]` is exercised: (i) `t4` points to the `func` field of an alive, user-provided `u_wait_queue_t` object, and (ii) the data read by `t5=[t4]` equals to the one read in the reference kernel (i.e., `0xdeadbeef`). Solving those two constraints, AEM will adjust the exploit to load a hijacked control-flow-transfer target at `t5=[t4]`.

[1]Given two memory operations, we locate their source code lines from the debug information. We use edit distance to measure their similarity and set a threshold of 0.9 to identify potentially aligned memory operations.

TABLE I
DETAILED INFORMATION OF 28 EXPS FROM 2016 TO 2021.

CVE	EXP ID	Vulerability Type	Exploiting Primitive	Malicious Behaviour	CVSS Score	C LoC	Syscall Count	Ref. Ver.
CVE-2016-4557	exp1 [19]	UAF	Control Flow Hijacking	ROP	7.8	154	195*	v4.4
CVE-2016-4557	exp2 [20]	UAF	Control Flow Hijacking	ROP	7.8	136	272	v4.5
CVE-2016-6187	exp1 [21]	Heap OOB Write	Control Flow Hijacking	-	7.8	259	353	v4.5
CVE-2016-8655	exp1 [22]	Race Condition	Control Flow Hijacking	SYSCTL hijacking	7.8	1,043	14,344	v4.4
CVE-2016-9793	exp1 [23]	Heap OOB Write	Control Flow Hijacking	Ret2User	7.8	176	76	v4.8
CVE-2017-2636	exp1 [24]	Race Condition	Arbitrary Address Read/Write	Write Cred	7.8	722	2,935*	v4.10
CVE-2017-5123	exp1 [25]	Logic Bug	Control Flow Hijacking	Ret2User	8.8	124	35	v4.13
CVE-2017-6074	exp1 [26]	Double Free	Control Flow Hijacking	-	7.8	747	604	v4.8
CVE-2017-6074	exp2 [27]	Double Free	Control Flow Hijacking	ROP	7.8	744	756	v4.4
CVE-2017-7184	exp1 [28]	Heap OOB Write	Restricted Address Write Restricted Value	Write Cred	7.8	394	37*	v4.10
CVE-2017-7184	exp2 [29]	Heap OOB Write	Restricted Address Write Restricted Value	Write Cred	7.8	342	41	v4.10
CVE-2017-7308	exp1 [30]	Heap OOB Write	Control Flow Hijacking	ROP	7.8	545	675	v4.10
CVE-2017-7308	exp2 [31]	Heap OOB Write	Control Flow Hijacking	JOP	7.8	592	958	v4.8
CVE-2017-7308	exp3 [32]	Heap OOB Write	Control Flow Hijacking	ROP	7.8	545	689	v4.10
CVE-2017-8824	exp1 [33]	UAF	Control Flow Hijacking	Ret2User	7.8	92	13	v4.10
CVE-2017-8890	exp1 [34]	Double Free	Control Flow Hijacking	Ret2User	7.8	547	1,084	v4.10
CVE-2017-10661	exp1 [35]	UAF	Control Flow Hijacking	-	7.0	242	64*	v4.10
CVE-2017-11176	exp1 [15]	UAF	Control Flow Hijacking	ROP	7.8	271	676	v4.1
CVE-2017-15649	exp1 [36]	UAF	Control Flow Hijacking	Ret2User	7.8	405	414*	v4.13
CVE-2017-16995	exp1 [37]	Logic Bug	Arbitrary Address Write Arbitrary Value	Write Cred	7.8	454	541	v4.10
CVE-2017-16995	exp2 [38]	Logic Bug	Arbitrary Address Write Arbitrary Value	Write Cred	7.8	260	59	v4.4
CVE-2017-18344	exp1 [39]	Global OOB Read	Arbitrary Address Read	Dump physmap Memory	5.5	1,692	9,645	v4.14
CVE-2017-1000112	exp1 [40]	Heap OOB Write	Control Flow Hijacking	ROP	7.0	672	59	v4.8
CVE-2018-5333	exp1 [41]	NULL Ptr. Dereference	Control Flow Hijacking	ROP	5.5	574	138	v4.4
CVE-2018-6555	exp1 [42]	UAF	Control Flow Hijacking	-	7.8	276	189	v4.8
CVE-2018-9568	exp1 [43]	Type Confusion	Control Flow Hijacking	-	7.8	243	731*	v4.8
CVE-2019-15666	exp1 [44]	Heap OOB Write	Restricted Address Write Restricted Value	Write Cred	4.4	545	18,529	v4.10
syzbot#6a039858	exp1 [45]	UAF	Control Flow Hijacking	-	-	598	1,105	v4.14

* For the infinite loops in the exploit, number of SYSCALLs in the loop is only calculated once.

V. IMPLEMENTATION

We implemented a prototype of AEM on top of Angr [46] and S2E [47]. It consists of (i) 4,698 lines of C++ code to enhance S2E for tracing the execution of an exploit and performing the exploit adjustment with its capability of symbolic execution and (ii) 6,141 lines of Python code which use Angr to build an efficient instruction trace analysis, including context information extraction for memory operations, EXPGRAPH construction from the reference exploit, unaligned points identification in the target kernel, etc. We describe some important implementation details below.

Strategies to Ease Exploit Adjustment: During exploit adjustment, AEM often needs to symbolize syscall arguments which are essentially user-space data. However, due to randomness introduced by the runtime and ASLR, the arguments can locate at a different place every time the exploit is executed, making it inconvenient to identify the data to be symbolized. We adopt the following strategies to force the exploit to use a fixed layout for the user-space memory. First, we `mmap` a large region of memory at a fixed place and pivot the user-space stack to that region. Second, we replace the `Glibc` heap allocator with a linear allocator to fix the locations of heap objects. Finally, we disable PIE. Note that these strategies are only used to ease the exploit adjustment process and do no modify the kernel. Thus, they do not hurt the capability and the generality of the migrated exploit.

Extraction of Static/Runtime Type Information: The static type information is obtained from the debug information. Specifically, given an instruction that accesses a memory unit, AEM first decides which variable this memory unit corresponds to by using the information in the `.debug_loc` section and the `.debug_info` section. Then, AEM gets the type of the variable according to information in the `.debug_info` section. To obtain the runtime type of a memory unit, AEM relies on the data type specified at the memory allocation site. This is done by a self-crafted S2E plugin which tracks all allocations and deallocations and determines which memory belongs to which allocation site.

VI. EVALUATION

A. Experimental Setup

To facilitate the evaluation, we gather a set of real-world exploits and a group of diverse Linux kernels.

Exploit Dataset: We considered all Linux kernel vulnerabilities released on CVE database [48] and syzbot [14] (a Syzkaller-based fuzzing system for Linux kernel) between 2016 and 2021. For each vulnerability, we exhaustively searched for memory corruption enabled exploits included in public databases [49], [50], [51], technical articles [52], [53], [30], [15], and existing AEG projects [2], [3]. In total, we collected 78 exploits, among which only 6 exploits are not enabled through memory corruptions.

To understand the target versions of these exploits, we performed a study on the collected 78 Linux kernel exploits. Specifically, we manually inspected the technical articles, and the comments in the source code of these exploits to check how many target versions have the authors explicitly declared. If no target version is declared in an exploit, we regard it only targets one kernel version. The results show that around 80% of the exploits are only designed for one particular version, which calls for cross-version exploitability assessment.

TABLE II
MIGRATION RESULTS OF AEM.

	CVE	EXP ID	Vul. Type	Reference	Results of Exploit Migration on Target Versions[1]												
					v4.1	v4.4	v4.5	v4.6	v4.8	v4.10	v4.13	v4.14	v4.16	A4.4.0	A4.9.44	U4.4.0-21	U4.8.0-34
No Migration Needed	CVE-2016-4557	exp1	UAF	v4.4	✓	☞	✓	-	-	-	-	-	-	-	-	-	-
	CVE-2016-6187	exp1	Heap OOB Write	v4.5	-	-	☞	✓	-	-	-	-	-	-	-	-	-
	CVE-2016-9793	exp1	Heap OOB Write	v4.8	✓	✓	✓	✓	☞	-	-	-	-	-	-	✓	-
	CVE-2017-5123	exp1	Logic Bug	v4.13	-	-	-	-	-	-	☞	-	-	-	-	-	-
	CVE-2017-6074	exp2	Double Free	v4.4	-	☞	✓	✓	✓	-	-	-	-	-	-	✓	✓
	CVE-2017-7184	exp1	Heap OOB Write	v4.10	✓	✓	✓	✓	✓	☞	-	-	-	-	-	✓	✓
	CVE-2017-7184	exp2	Heap OOB Write	v4.10	✓	✓	✓	✓	✓	☞	-	-	-	-	-	✓	✓
	CVE-2017-8824	exp1	UAF	v4.10	✓	✓	✓	✓	✓	☞	-	-	-	-	-	✓	✓
	CVE-2017-8890	exp1	Double Free	v4.10	✓	✓	✓	✓	✓	☞	-	-	-	-	-	-	✓
	CVE-2017-15649	exp1	UAF	v4.13	-	-	-	-	-	✓	☞	-	-	-	-	-	-
	CVE-2017-16995	exp1	Logic Bug	v4.10	✓	✓	✓	✓	✓	☞	✓	✓	-	✓	✓	✓	✓
	CVE-2017-1000112	exp1	Heap OOB Write	v4.8	-	-	✓	✓	☞	✓	-	-	-	-	-	✓	✓
	CVE-2019-15666	exp1	Heap OOB Write	v4.10	-	-	-	-	✓	☞	✓	✓	✓	-	-	-	✓
Migration Needed	CVE-2016-4557	exp2	UAF	v4.5	✓	✓	☞	-	-	-	-	-	-	✓	-	✓	-
	CVE-2016-8655*	exp1	Race Condition	v4.4	×	☞	✓	✓	✓	-	-	-	-	-	-	×	-
	CVE-2017-2636*	exp1	Race Condition	v4.10	×	×	×	×	×	☞	-	-	-	-	-	×	×
	CVE-2017-6074	exp1	Double Free	v4.8	-	✓	✓	✓	☞	-	-	-	-	-	-	✓	✓
	CVE-2017-7308	exp1	Heap OOB Write	v4.10	✓	✓	✓	✓	✓	☞	-	-	-	-	-	✓	✓
	CVE-2017-7308	exp2	Heap OOB Write	v4.8	×	✓	✓	✓	☞	✓	-	-	-	-	-	✓	✓
	CVE-2017-7308	exp3	Heap OOB Write	v4.10	×	✓	✓	✓	✓	☞	-	-	-	-	-	✓	✓
	CVE-2017-10661*	exp1	UAF	v4.10	×	×	✓	✓	✓	☞	-	-	-	-	-	✓	✓
	CVE-2017-11176	exp1	UAF	v4.1	☞	✓	✓	✓	✓	×	-	-	-	✓	-	✓	✓
	CVE-2017-16995	exp2	Logic Bug	v4.4	✓	☞	✓	✓	✓	×	×	×	-	✓	×	✓	✓
	CVE-2017-18344	exp1	Global OOB Read	v4.14	✓	✓	✓	✓	✓	✓	✓	☞	-	✓	✓	✓	✓
	CVE-2018-5333	exp1	NULL Ptr. Dereference	v4.4	✓	☞	✓	✓	✓	✓	✓	✓	-	✓	×	✓	✓
	CVE-2018-6555	exp1	UAF	v4.8	×	✓	✓	✓	☞	×	×	×	×	✓	×	✓	✓
	CVE-2018-9568	exp1	Type Confusion	v4.8	×	✓	✓	✓	☞	×	×	-	-	-	-	✓	✓
	syzbot#6a039858	exp1	UAF	v4.14	×	×	×	×	✓	✓	✓	☞	×	-	-	×	✓

[1] - means vulnerability is patched or the vulnerable code does not exist; ☞represents reference version; ✓ indicates the exploit can natively work on the target version; ✓ and × respectively show that AEM succeeds or fails when migrating the exploit to the target version.
* AEM cannot work on the exploits because S2E does not support booting kernel with symmetric multiprocessing (smp>2), which is required by those EXPs.

After analysis for at least 24 total human-hours on each case, we reproduced 28 exploits against 22 kernel vulnerabilities in the provided/suggested environment, as summarized in Table I. The vulnerabilities span various types, including UAF, race condition, heap overflows, global out-of-bound access, null pointer dereferences, and logic bugs. On average, the exploits contain over 400 lines of C code and issue about 2,000 system calls in the exploiting process. To our knowledge, our dataset represents the largest corpus of exploits considered by research on Linux kernel exploitability assessment.

Kernel Dataset: For each vulnerability, we gathered a group of Linux kernels, including both versions targeted by the exploit and versions reported as affected by the vulnerability. As summarized in Table V (in Appendix), the entire set consists of 13 kernels, ranging from mainstream versions between v4.1 and v4.16 to downstream variants supporting Android/Ubuntu.

Evaluation Environment: All our evaluations are performed on a machine with Intel Xeon Gold CPU 6242 2.80GHz and 192 GB RAM, running Ubuntu 18.04.5 LTS.

B. Native Cross-Version Capability of Exploits

Our evaluation starts with understanding the capability of wild exploits in supporting multiple kernel versions. Given an exploit, we first exclude kernel versions that are not affected by the vulnerability. The approach is to manually verify whether the vulnerable code is absent or the patch is present in a kernel. Either condition being true leads us to consider the kernel unaffected. Initially, our dataset includes 364 <exploit, kernel> pairs. 154 of them are excluded in this step and 210 remain.

In the follow-up step, we run each exploit on all affected kernels and observe the exploitation result. Technically, manual analysis is performed to learn the primitives that the exploit can achieve in the reference kernel. Given a target kernel, we consider the exploit successful if the same primitives arise. Take `exp1` of `CVE-2017-8824` as an example. This exploit creates a control-flow hijacking primitive which redirects the control flow to a user-space address. By checking whether such control-flow hijacking occurs or not, we can determine the success status of the exploit on a target version. Note that the manual analysis here is just to confirm whether an exploit succeeds on a target kernel and the primitive identification step is still automated.

Among the 210 <exploit, kernel> pairs, 28 correspond to the reference version (i.e., the target version specified by the exploit) and the remaining 182 represent cross-version exploitation. The wild exploits succeed in 92 of the cross-version pairs and fail in the other 90 (spanning 15 exploits and 13 CVEs). This demonstrates that real-world exploits have imperfect built-in support for different versions.

C. Effectiveness of AEM

Applied on the 90 cross-version <exploit, kernel> pairs, AEM successfully migrates the exploit for 56 pairs. As we will point out shortly, in 23 out of the failed 34 pairs, the primitives carried by the exploit are infeasible to achieve in the target kernels (aka non-migratable exploits). Excluding the 23 cases, AEM achieves a success rate of 83.5%. Detailed results are presented in Table II.

```
1  void rds_atomic_free_op(struct rm_atomic_op *ao){
2     struct page *page = sg_page(ao->op_sg);
3     struct address_space *mapping = page_mapping(page);
4     if (likely(mapping)) {
5        int (*spd)(struct page *) = mapping->a_ops->set_page_dirty;
6        ...
7        return (*spd)(page);
8     }
9  }
10 struct address_space *page_mapping(struct page *page){
11    unsigned long mapping;
12    if (unlikely(PageSlab(page)))
13       return NULL;
14    if (unlikely(PageSwapCache(page))) {
15       ...
16       return swap_address_space(entry);
17    }
18    return page->mapping;
19 }
20 #define TESTPAGEFLAG(uname, lname)                    \
21 static inline int Page##uname(const struct page *page)  \
22    { return test_bit(PG_##lname, &page->flags); }
```

Fig. 5. Code related to the exploitation of CVE-2018-5333.

1) How AEM Succeeds

To understand how AEM "migrates" the exploits exactly, we inspected the 56 successful cases. These cases can be classified into two categories, respectively corresponding to the two types of changes described in §II-B. In 7 cases, the migration encounters both types of changes and thus, they are double-counted in both categories.

Category-1: Amending Effects of Code Changes (10 cases). Target kernels in this category carry code changes compared to their reference counterparts, which impedes the exploit from working. The EXPGRAPH constructed by AEM captures the discrepancy created by the code changes in the execution context and guides AEM to adjust the exploit.

Figure 5 shows an example in this category. The vulnerability, CVE-2018-5333, is a null-pointer dereference that occurred in function `rds_atomic_free_op`. At line 2, `sg_page` can return a null pointer to `page`, which is passed to `page_mapping` at line 3. If the `flags` field in the data structure pointed to by `page` dissatisfies all the checks in lines 12-17, `page_mapping` returns the `mapping` field in that data structure at line 17. Afterward, the field `mapping` is dereferenced at line 5 to retrieve a function pointer, which is called at line 7. Given a kernel where `mmap_min_addr`[2] is set to 0, the exploit can map and dereference the null address to fill the memory with arbitrary data. Thus, by writing the memory at the null address with a fake `page` whose `flags` does not satisfy any checks in lines 12-17, the exploit will be able to control the retrieval of the function pointer at line 5 and eventually hijack the function call at line 7.

In the above example, the control-flow hijacking primitive depends on the context that `flags` in `page` invalidates the checks enforced by `PageSlab` and `PageSwapCache` (line 12-17). However, the code changes across kernel versions and the semantic of `flags` evolves. The flag `PG_Slab` and `PG_SwapCache` are defined at the 8th bit and 16th bit in Linux v4.4, which change to the 9th bit and the 28th bit in v4.10. The exploit, designed for v4.4, does not set the desired bits for v4.10, failing to incur the execution of line 18 towards the primitive.

When migrating the exploit for Linux v4.10, AEM builds the EXPGRAPH on top of v4.4 (the reference version), which catches the dependency chain of `spd←mapping←page←op_sg`. In contrast, execution on v4.10 only presents `page←op_sg`. This inspires AEM to realign the dependency of `mapping←page`. The first thing to do so is to activate the retrieval of `mapping` on v4.10, for which line 18 is the candidate code picked by AEM. Comparing the EXPGRAPH and the execution trace of v4.10, AEM finds that the last aligned point locates at line 2. Thus, it runs symbolic execution from there, setting unrestricted user-space data, including `page->flags` as symbolic values. Through constraint solving, AEM can easily find a feasible path to line 18, which automatically enables the exploit primitive.

Category-2: Amending Effects of Data Changes (53 cases). Kernels in this category use different definitions/structures for data objects, when compared with the reference versions. This often leads to disparity in the memory layout and makes the exploit unsuccessful. The EXPGRAPH built by AEM illustrates the layout disparity and how that affects the exploit, intelligently guiding AEM on making the desired adjustment.

Figure 6 demonstrates an example in this category. The vulnerability, CVE-2017-7308, is an out-of-bound write. Upon a new network packet, `tpacket_rcv` calls `packet_current_rx_frame` (line 14) to get an active block at `pkc->nxt_offset`, followed by saving the packet data to that block through `skb_copy_bits` (line 16). However, the birth of `pkc->nxt_offset` (line 4) involves a user-controlled component `pkc1->blk_sizeof_priv`. As a result, `pkc->nxt_offset` can be modified to an address out of the block. The exploit against the vulnerability first sprays a `packet_sock` object to occupy the memory that can be overflown. It then activates the overflow to manipulate the `xmit` field (line 1), a function pointer, in the `packet_sock` to a desired code location. By eventually triggering a call to `xmit`, the exploit can hijack the control flow.

The exploit is designed for Linux v4.10, where `xmit` has an offset of 1,296 to the start of `packet_sock`. Switching to v4.4, the offset is 1,304. Thus, the exploit cannot properly locate `xmit`, failing to create the control-flow hijacking primitive. When migrating the exploit to v4.4, AEM finds that all memory operations on EXPGRAPH are aligned on v4.4. The only exception happens at the final reading of the hijacked control-flow-transfer target. On the EXPGRAPH, the memory has runtime type of `xmit` and is manipulated to an adversary-desired value by the overflow, which differs on v4.4. This inspires AEM to set the constraint that the address of the final reading on v4.4 should point to the location of a `xmit` and the data read from `xmit` should be the same as v4.10. By solving this constraint, AEM adjusts the exploit to amend the final

[2]https://wiki.debian.org/mmap_min_addr

```
1  struct packet_sock {..., int (*xmit)(struct sk_buff *skb);}
2  static void prb_open_block(struct tpacket_kbdq_core *pkc1, ...){
3      ...
4      pkc1->nxt_offset = pkc1->pkblk_start +
                  BLK_PLUS_PRIV(pkc1->blk_sizeof_priv);
5      ...
6  }
7  static void *packet_current_rx_frame(struct packet_sock *po, ...){
8      ...
9      pkc = GET_PBDQC_FROM_RB(&po->rx_ring);
10     return pkc->nxt_offset;
11     ...
12 }
13 static int tpacket_rcv(struct sk_buff *skb, ...){
       ...
14     h.raw = packet_current_rx_frame(po, ...);
15     ...
16     skb_copy_bits(skb, 0, h.raw + macoff, snaplen);
17 }
```

Fig. 6. Code related to the exploitation of `CVE-2017-7308`.

reading to load the desired value stored in the sprayed `xmit`, thus achieving the exploiting primitive.

2) Why AEM Fails

To understand the capability boundary of AEM, we further inspected the 34 cases where AEM failed. In general, the cases also belong to two categories.

Category-1: Engineering Issues (11 cases). AEM cannot run on 3 exploits (see Table II), corresponding to 11 cases in need of cross-version migration. This is because the S2E engine does not support booting kernel with symmetric multiprocessing (smp>2), which is, however, required by the exploits.

Category-2: Non-migratable Ones (23 cases). AEM aims to migrate the primitives carried by the available exploit. However, the primitives are often infeasible on the target versions.

- *Barriers in configuration (6 cases):* Many exploits require the presence/absence of certain configurations to work. The conditions for such configurations may not hold in the target versions, which in principle prevents the exploits from succeeding. For example, `exp1` on `CVE-2018-6555` gains a control-flow-hijack primitive by manipulating a global function pointer `ptmx_fops`. However, on Linux after v4.9, `ptmx_fops` is marked read-only after initialization. Thus, the primitive is infeasible.
- *Barriers in functionality (2 cases):* A group of exploits leverages special functionality in the reference version to create primitives. Unquestionably, the primitives can never be fulfilled in target versions where the functionality is unavailable. Consider the previously discussed `CVE-2018-5333` as an example. It requires the functionality of mapping the null address to be exploited. However, kernel A4.9.44 does not enable this functionality.
- *Barriers in data structures (5 cases):* The success of exploits also often depends on particular data structures (or sub-structures). For target versions without the needed data structures, the exploits certainly won't work. When exploiting `CVE-2017-16995`, `exp2` creates a privilege-escalation primitive by manipulating the `cred` in the `task_struct` field of a `thread_info` object. However, `thread_info` no longer maintains a `task_struct` field after Linux v4.9, making the exploit unusable.
- *Barriers in memory management (10 cases):* Heap-spray-based exploits need compliance with memory management. Target kernels using different memory management are inevitably incompatible with those exploits. In the case of `syzbot#6a039858`, a UAF occurs on a `route4_filter` object. To exploit the UAF vulnerability, `exp1` sprays the heap with a `msg_msg` object to override the freed `route4_filter`. This works on Linux v4.6-4.14 because `route4_filter` and `msg_msg` are placed in the same cache (`kmalloc-96`). On other versions, `route4_filter` is managed in cache `kmalloc-128` or `kmalloc-192`. As a result, the spray does not work. Unlike the other three scenarios, the primitives in the scenario can be re-enabled by finding another object to spray and use. This, however, requires generation capabilities like those offered by AEG [2], [7], which is beyond the migration AEM aims to provide.

TABLE III
AVERAGE TIME NEEDED BY AEM TO MIGRATE AN EXPLOIT.

CVE	EXP	Time Cost (*min*)				
		Tracing	EXPGraph	Alignment	Adjustment	**Total**[1]
CVE-2016-4557	exp2	40.75	5.00	6.17	0.80	93.47
CVE-2017-6074	exp1	67.97	6.07	9.47	0.88	152.35
CVE-2017-7308	exp1	141.33	7.74	10.69	1.28	302.37
CVE-2017-7308	exp2	160.73	2.00	-	-	-
CVE-2017-7308	exp3	160.85	22.95	26.60	2.70	373.95
CVE-2017-11176	exp1	51.24	9.24	14.26	5.48	211.70
CVE-2017-16995	exp2	14.14	3.68	5.37	2.05	39.38
CVE-2017-18344	exp1	1,242.02	53.37	79.29	3.97	5,377.97
CVE-2018-5333	exp1	53.72	8.75	12.05	0.72	128.95
CVE-2018-6555	exp1	31.17	15.15	18.32	3.17	98.98
CVE-2018-9568	exp1	122.67	1.28	1.48	2.73	379.00
syzbot#6a039858	exp1	51.18	2.97	4.32	4.77	114.42
Avg.	-	178.15	11.52	17.09	2.60	661.14

[1] **Total** is the time cost of adjusting the exploit for n times, calculated as $T_{total} = (n+1) * t_{Tracing} + t_{EXPGraph} + n * (t_{Classification} + t_{Adjustment})$.

D. Efficiency of AEM

The analysis of AEM involves complex operations to identify migration targets and adjust the exploit, which can lead to high time cost. Accordingly, we also measure the efficiency of AEM when handling the testcases. As shown in Table III, AEM needs an average time of 661 minutes to migrate an exploit. In cases like the migration of `exp2` on `CVE-2016-4557`, the needed time is only 1.5 hours. We envision the efficiency of AEM is comparable to, if not higher than, human analysis.

Where the Cost Comes From: To better understand the sources of the time cost, we break down the operations of AEM and separately measure their efficiency. The results in Table III illustrate that AEM only needs a few minutes to generate the EXPGRAPH ($< 12m$), identify aligned point ($< 18m$), and adjust the EXP ($< 3m$). In contrast, most of the time (84.15%+) AEM spends is on tracing the execution. The reason is that S2E, used by AEM for tracing, runs expensive dynamic binary translation, whose cost dramatically increases with the number of instructions. In this regard, the majority of AEM's cost is rooted in the tool we use, instead of its designs.

How Our Design Improves Efficiency: To migrate an exploit,

TABLE IV
DETAILED STATISTICS FOR EXPLOIT UNDERSTANDING AND EXPLOIT MIGRATION FOR THE 12 EXPLOITS WHERE AEM WORKS.

CVE	EXP Name	# of vertices on EXPGRAPH	Adjustment Times	Adjustment Type
CVE-2016-4557	exp1	11	1	I
CVE-2017-6074	exp1	15	1	I
CVE-2017-7308	exp1	13	1	I
CVE-2017-7308	exp2	9	-	-
CVE-2017-7308	exp3	29	1	I
CVE-2017-11176	exp1	21	2	II, I
CVE-2017-16995	exp2	28	1	I
CVE-2017-18344	exp1	83	3	I, I, I
CVE-2018-5333	exp1	5	1	II
CVE-2018-6555	exp1	11	1	I
CVE-2018-9568	exp1	9	2	I, I
syzbot#6a039858	exp1	7	1	I
Avg.	-	19.36	1.36	-

the key is to reason the failure and properly adjust an exploit. However, the search space is in general large. AEM introduces EXPGRAPH to reduce the search space and thus, improve the overall efficiency. We run two experiments to evaluate this matter. First, we measure the utility of EXPGRAPH in identifying memory operations to align. As summarized in Table IV, EXPGRAPH, after filtering out irrelevant memory operations, only involves less than 20 vertices on average. In contrast, a single system call can incur over 3,400 memory operations. This demonstrates AEM's capability in identifying memory operations tied to the exploit. Second, we inspect the complexity of EXPGRAPH on adjusting the exploit. On average, AEM only needs to adjust the given exploit 1.36 times to make it work on the target kernel. This further illustrates that AEM's design facilitate the efficiency.

E. Comparison with AEG

In the last evaluation, we performed an experiment to compare AEM with existing AEG solutions. To the best of our knowledge, only two solutions (FUZE [2] and KOOBE [3]) support vulnerabilities covered by our dataset.

FUZE: We experimented with the public version of FUZE [54] on the UAF vulnerabilities included in our dataset. However, FUZE could not work on these cases because the released source code is incomplete. Specifically, as confirmed with the developers, the released source code of FUZE does not include two critical components: (1) a dynamic tracing module to extract required information (e.g., spatial and temporal metadata) of a vulnerability on the target kernel and (2) an under-context kernel fuzzing module to find sensitive operations on a vulnerable object.

KOOBE: In our evaluation dataset, three exploits work against kernel OOB vulnerabilities, which were considered as PoCs to test KOOBE. In summary, KOOBE cannot generate working exploits for any of them. The generation even broke at the first step of understanding the capability of the OOB vulnerabilities. More specifically, the OOB behaviors were not detected by either standard detection tools (e.g., KASAN [55]) or KOOBE's built-in detector. In contrast, AEM neither relies on such external tools nor needs to build an understanding of the vulnerabilities. We further manually adjusted the exploit such that KOOBE can understand the vulnerability's capability. However, KOOBE still failed to automatically generate a working exploit for the target kernels. First, KOOBE requires manual annotations of the syscalls and arguments needed for exploration, preventing the tests from being completed automatically. Second, KOOBE involves a large number of constraints, including many of those irrelevant to the exploitation process. The solvers often fell short of solving those constraints. In comparison, AEM requires no manual annotations and it simplifies the constraints by focusing on the ones connected to the exploiting primitives.

VII. DISCUSSION

Generality of AEM: Though AEM is designed to facilitate cross-version exploitability assessment for Linux kernels, the basic idea of AEM, including its core designs (e.g., the memory operation alignment algorithm and the exploit adjustment methodology), is also applicable to user-space applications. To support user-space applications, AEM needs to be extended from two aspects. First, AEM adopts syscalls as the perspective to adjust the kernel exploit. However, the exploit for user-space programs is usually raw bytes from program input (e.g., files or standard input). Thus, their exploit payloads require a new perspective to adjust. Second, the instruction tracing system needs to be adapted to capture user-space behaviors needed by our analysis (e.g., allocations and deallocations of user-space heap objects).

Limitations of AEM: As described in §II-B, the major obstacles in cross-version exploitability assessment are the implementation changes across different kernels. AEM mitigates these obstacles by designing a primitive-centric, alignment-guided exploit migration approach. However, our approach cannot adapt to all degrees of change. First of all, we acknowledge that AEM only applies to *migratable cases*, as defined in §II-D. To support those non-migratable but exploitable cases, AEM could be extended with AEG to incorporate certain state exploration capability. Second, our alignment-guided design may meet difficulties in some migratable cases when there are excessive implementation changes. For example, we rely on the type information of the accessed memory to align memory operations. However, when the type is redesigned or re-organized, our current design may fail to align them. Besides, we rely on source code similarity to search a candidate op_{tar} in *R2 Exploit Adjustment*, which may have false positives. When a wrong op_{tar} is selected, our current design can fail to migrate the exploit. In theory, there would be many other cases that make AEM fail. However, we do not encounter such cases in our evaluation. We envision the failures will not happen a lot in general as cross-version code changes of a long-term maintenance project such as Linux kernel are typically less significant. Third, our prototype of AEM does not support booting kernels with symmetric multiprocessing, which we hope to address in our future work.

VIII. RELATED WORK

Automatic Exploit Generation: Starting around 15 years ago, a series of works have been proposed to automate the generation

of exploits. Brumley *et al.* [56] first introduce automatic patch-based exploit generation (APEG), which automatically creates an exploit for an unpatched program based on patch changes. In the follow-up development, Thanassis *et al.* [5] model the AEG problem as a program-verification task and develop a pioneering, end-to-end AEG system based on preconditioned symbolic execution. Later on, Mayhem [57] and Revery [11] are created to leverage concolic execution, memory modeling, and layout-oriented fuzzing to improve the scalability of AEG.

Despite the above efforts, the community gradually realizes that general, full AEG for modern software is less feasible. As a result, recent research narrows down the scope. One common strategy is to focus AEG on a specific family of vulnerabilities. To date, there have been AEG solutions designated for user-after-free [2], [8], out-of-bound heap access [3], uninitialized variables [6], [58], and data races [59]. The other popular strategy opts to facilitate specific tasks in creating exploits. [60], [61], [62], [63], [64], [7] propose to identify desired objects and create correct memory layouts to help exploitation of memory corruptions; [10], [65] explore new forms of primitives to expand the possibility of the exploitation; [66], [67], [68] present solutions for enabling exploits to bypass mitigation mechanisms (e.g., ASLR and CFI).

As we have pointed out in §II, AEG at the current stage is not a well-suited methodology for cross-version exploitability assessment. On the one hand, AEG prevalently relies on templates to create exploits. However, not every vulnerability can be exploited following those templates. On the other hand, AEG builds every component from scratch, although the existing exploit already carries many reusable components. Thus, AEG can incur an unnecessarily high burden and present a lower-than-expected success rate when applied to cross-version exploitability assessment.

Cross-Version Assessment of Vulnerabilities: This line of research understands whether a disclosed vulnerability affects different software versions. In principle, two approaches are applicable. The first approach detects the presence of affected code [69], [70], [71] or available patches [72], [73], [74], [75], [76]. Most solutions using this approach are based on features obtained via static code analysis (e.g., code syntax [69], [73], data constants [70], and normalized code [71]). While offering scalability and coverage, these solutions cannot guarantee the validity of vulnerabilities even their detection shows a positive result, considering that they cannot provide proof showing the vulnerabilities can be indeed triggered. The second approach, which is more intuitive, aims to produce testcases to trigger the vulnerabilities [77], [78], [79], [80], [18], [81], [82], [83]. AFLGo [77] and Hawkeye [78] propose directed fuzzing, which derives inputs to gradually approach the target code and activate the vulnerabilities. Savior [79] follows a similar philosophy but uses more intelligent techniques like concolic execution.

Our work aims to understand the cross-version exploitability of vulnerabilities, which can be deemed as a follow-up step to complement the above research.

IX. Conclusion

This paper introduces a new methodology, AEM, to tackle the problem of cross-version exploitability assessment for Linux kernels. The philosophy of AEM is to force the strategy adopted by a given exploit to work on all applicable kernel versions. Technically, AEM leverages the successful exploitation on the reference kernel as guidance and adjusts the exploit to reproduce the same exploiting behaviors on other kernels. Our evaluation with real-world datasets shows a success rate of 83.5%, demonstrating the validity of our philosophy and the utility of AEM.

X. Acknowledgment

We thank the anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (U1836210, U1836213, 62172105, 61972099, 62172104, 62102091, 62102093), the National Key R&D Program of China (2021YFB3101200), and the Natural Science Foundation of Shanghai (19ZR1404800). Yuan Zhang was supported in part by the Shanghai Rising-Star Program 21QA1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] Q. Wu, Y. Xiao, X. Liao, and K. Lu, "OS-Aware Vulnerability Prioritization via Differential Severity Analysis," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, 2022.

[2] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[3] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[4] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[5] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic Exploit Generation," *Communications of the ACM*, vol. 57, no. 2, 2014.

[6] K. Lu, M. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes, "Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[7] Y. Chen and X. Xing, "SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[8] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[9] Y. Chen, Z. Lin, and X. Xing, "A Systematic Study of Elastic Objects in Kernel Exploitation," in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[10] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[11] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From Proof-of-Concept to Exploitable," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[12] L. Torvalds, "Linux kernel source tree," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/, 2021.

[13] T. M. Corporation, "Linux kernel: Vulnerability statistics," https://www.cvedetails.com/product/47/?q=Linux+kernel, 2022.

[14] Google, "syzbot dashboard," https://syzkaller.appspot.com/upstream, 2018.

[15] N. FABRETTI, "CVE-2017-11176: A step-by-step Linux Kernel exploitation," https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html, 2021.

[16] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to "weird machines" and theory of computation," *login Usenix Mag.*, vol. 36, no. 6, 2011.

[17] M. Miller, "Modeling the exploitation and mitigation of memory safety vulnerabilities," Breakpoint 2012, 2012.

[18] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating Vulnerability Assessment through PoC Migration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[19] jannh, "Issue 808: Linux: UAF via double-fdput() in bpf(BPF_PROG_LOAD) error path," https://bugs.chromium.org/p/project-zero/issues/detail?id=808, 2016.

[20] W. Wu, "Exploit for CVE-2016-4557," https://github.com/ww9210/Linux_kernel_exploits/tree/master/cve-2016-4557-exp1, 2018.

[21] V. Nikolenko, "CVE-2016-6187: Exploiting Linux kernel heap off-by-one," https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit, 2016.

[22] bcoles, "Linux AF_PACKET race condition exploit for CVE-2016-8655," https://github.com/bcoles/kernel-exploits/tree/master/CVE-2016-8655, 2016.

[23] SecWiki, "Local root exploit for the vulnerability in the SO_SNDBUFFORCE and SO_RCVBUFFORCE socket options implementation CVE-2016-9793," https://github.com/SecWiki/linux-kernel-exploits/tree/master/2016/CVE-2016-9793, 2016.

[24] A. Popov, "CVE-2017-2636: Exploit the race condition in the n_hdlc Linux kernel driver," https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html, 2017.

[25] C. Salls, "Exploiting CVE-2017-5123 with full protections. SMEP, SMAP, and the Chrome Sandbox!" https://salls.github.io/Linux-Kernel-CVE-2017-5123/, 2017.

[26] wjbsyc, "Exploit for CVE-2017-6074," https://github.com/wjbsyc/syzbot-exploit/blob/main/CVE-2017-6074-exploit.c, 2021.

[27] A. Konovalov, "Local root exploit for the vulnerability in the DCCP protocol implementation CVE-2017-6074," https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-6074, 2017.

[28] snorez, "Exploit for CVE-2017-7184," https://github.com/snorez/exploits/tree/master/cve-2017-7184, 2017.

[29] p4nda, "Exploit for Linux xfrm module OOB write vulnerability CVE-2017-7184," https://github.com/ret2p4nda/kernel-pwn/tree/master/CVE-2017-7184, 2017.

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

[31] Y. Chen, "Exploiting the Linux kernel via file struct," https://github.com/chenyueqi/osok_kernel/blob/master/poc/2017-7308/, 2017.

[32] A. Konovalov, "Local root exploit for the vulnerability in the AF_PACKET sockets implementation CVE-2017-7308," https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-7308, 2017.

[33] W. Wu, "Exploit for CVE-2017-8824," https://github.com/ww9210/Linux_kernel_exploits/tree/master/cve-2017-8824-exp1, 2017.

[34] thinkycx, "Exploit for CVE-2017-8890," https://github.com/thinkycx/CVE-2017-8890, 2017.

[35] Anonymous, "Linux kernel < 4.10.15 - Race Condition Privilege Escalation," https://www.exploit-db.com/exploits/43345, 2017.

[36] W. Wu, "A LPE exploit for cve-2017-15649," https://github.com/ww9210/Linux_kernel_exploits/tree/master/cve-2017-15649, 2017.

[37] R. Larabee, "Linux Kernel < 4.13.9 (Ubuntu 16.04 / Fedora 27) - Local Privilege Escalation," https://www.exploit-db.com/exploits/45010, 2017.

[38] B. Leidl, "Linux Kernel < 4.4.0-116 (Ubuntu 16.04.4) - Local Privilege Escalation," https://www.exploit-db.com/exploits/44298, 2017.

[39] A. Konovalov, "Linux Kernel 4.14.7 (Ubuntu 16.04 / CentOS 7) - (KASLR & SMEP Bypass) Arbitrary File Read," https://www.exploit-db.com/exploits/45175, 2017.

[40] ——, "Local root exploit for the vulnerability in the UFO Linux kernel implementation CVE-2017-1000112," https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-1000112, 2017.

[41] bcoles, "Local root exploit for Linux RDS rds_atomic_free_op NULL pointer dereference," https://github.com/bcoles/kernel-exploits/blob/master/CVE-2018-5333, 2018.

[42] S. Disclosure and O. Nimron, "SSD Advisory – IRDA Linux Driver UAF," https://ssd-disclosure.com/ssd-advisory-irda-linux-driver-uaf/, 2018.

[43] QuestEscape, "Exploit for CVE-2018-9568_WrongZone," https://github.com/QuestEscape/exploit/tree/master/CVE-2018-9568_WrongZone, 2018.

[44] B. Cyber, "Exploiting CVE-2019-15666 by reversing the binary PoC," https://blog-cyber.riskeco.com/en/exploiting-cve-2019-15666-by-reversing-the-binary-poc/, 2019.

[45] wjbsyc, "Exploit for syzbot bug 6a039858238a38cbc7f372607fc5d49f4469cf2c," https://github.com/wjbsyc/syzbot-exploit/blob/main/syzbot-6a039858238a38cbc7f372607fc5d49f4469cf2c-control-pc.c, 2021.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[47] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[48] U. N. I. of Standards and Technology, "National Vulnerability Database," https://nvd.nist.gov/home.cfm, 2021.

[49] O. Security, "Exploit database," https://www.exploit-db.com/, 2021.

[50] Rapid7, "Vulnerability & exploit database," https://www.rapid7.com/db/, 2021.

[51] SecWiki, "Linux-kernel-exploits," https://github.com/SecWiki/linux-kernel-exploits, 2021.

[52] A. Konovalov, "kernel-exploits," https://github.com/xairy/kernel-exploits, 2021.

[53] S. S. Disclosure, "Ssd advisory – linux kernel af_packet use-after-free," https://ssd-disclosure.com/archives/3484, 2021.

[54] W. Wu, "Fuze project," https://github.com/ww9210/Linux_kernel_exploits, 2021.

[55] T. kernel development community, "The kernel address sanitizer (kasan)," https://www.kernel.org/doc/html/latest/dev-tools/kasan.html, 2021.

[56] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, 2008.

[57] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[58] H. Cho, J. Park, J. Kang, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[59] Y. Lee, C. Min, and B. Lee, "ExpRace: Exploiting Kernel Races through Raising Interrupts," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[60] S. Heelan, T. Melham, and D. Kroening, "Automatic Heap Layout Manipulation for Exploitation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[61] D. Repel, J. Kinder, and L. Cavallaro, "Modular Synthesis of Heap Exploits," in *Proceedings of the 12th Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017.

[62] I. Yun, D. Kapil, and T. Kim, "Automatic Techniques to Systematically Discover New Heap Exploitation Primitives," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[63] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[64] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards Automated Heap Feng Shui," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[65] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[66] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, 2011.

[67] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2011.

[68] E. Bosman and H. Bos, "Framing Signals - A Return to Portable Shellcode," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[69] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[70] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[71] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[72] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "PDiff: Semantic-based Patch Presence Testing for Downstream Kernels," in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[73] H. Zhang and Z. Qian, "Precise and Accurate Patch Presence Test for Binaries," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[74] Q. Feng, R. Zhou, Y. Zhao, J. Ma, Y. Wang, N. Yu, X. Jin, J. Wang, A. M. Azab, and P. Ning, "Learning Binary Representation for Automatic Patch Detection," in *Proceedings of 16th IEEE Annual Consumer Communications & Networking Conference, CCNC*, 2019.

[75] J. Dai, Y. Zhang, Z. Jiang, Y. Zhou, J. Chen, X. Xing, X. Zhang, X. Tan, M. Yang, and Z. Yang, "BScout: Direct Whole Patch Presence Test for Java Executables," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[76] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying Open-Source License Violation and 1-day Security Risk at Large Scale," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[77] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[78] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-Box Fuzzer," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[79] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: Towards Bug-Driven Hybrid Testing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[80] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided Greybox Fuzzing," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[81] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "MemLock: Memory Usage Guided Fuzzing," in *Proceedings of 42nd International Conference on Software Engineering (ICSE)*, 2020.

[82] H. Liang, L. Jiang, L. Ai, and J. Wei, "Sequence Directed Hybrid Fuzzing," in *Proceedings of 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020.

[83] M. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities," in *Proceedings of 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[84] Google, "Android kernel source tree," https://android.googlesource.com/kernel/common/+refs, 2021.

[85] C. Ltd., "Packages of ubuntu xenial for amd64," https://launchpad.net/ubuntu/xenial/amd64/, 2021.

XI. Appendix

A. Kernel Dataset

Table V shows the details of the 13 Linux kernels that we used to evaluate AEM. These kernels are from three vendors (Linux mainstream, Android and Ubuntu) and run on different kinds of devices, e.g., desktops, servers, and mobile phones.

TABLE V
DETAILED INFORMATION OF LINUX KERNEL DATASET.

Kernel Tag	Vendor[1]	Commit Id
v4.1	Torvalds	b953c0d234bc72e8489d3bf51a276c5c4ec85345
v4.4	Torvalds	afd2ff9b7e1b367172f18ba7f693dfb62bdcb2dc
v4.5	Torvalds	b562e44f507e863c6792946e4e1b1449fbbac85d
v4.6	Torvalds	2dcd0af568b0cf583645c8a317dd12e344b1c72a
v4.8	Torvalds	c8d2bc9bc39ebea8437fd974fdbc21847bb897a3
v4.10	Torvalds	c470abd4fde40ea6a0846a2beab642a578c0b8cd
v4.13	Torvalds	569dbb88e80deb68974ef6fdd6a13edb9d686261
v4.14	Torvalds	bebc6082da0a9f5d47a1ea2edc099bf671058bd4
v4.16	Torvalds	f8bae1feaa568b165612f0c5073268671f3a11ba
A4.4.0	Android	d9b927362a860ba48984913600cec51c09cbac47
A4.9.44	Android	34803e7c1c92f53603f6aa11235915afc2589290
U4.4.0-21	Ubuntu	linux-source-4.4.0_4.4.0-21.37
U4.8.0-34	Ubuntu	linux-source-4.8.0_4.8.0-34.36

[1] Torvalds kernel is collected from [12]. Android kernel is collected from [84]. Ubuntu kernel is collected from [85].