CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel

Shuangpeng Bai
The Pennsylvania State University
State College, United States
shuangpengbai@psu.edu

Zhechang Zhang
The Pennsylvania State University
State College, United States
zbz5352@psu.edu

Hong Hu
The Pennsylvania State University
State College, United States
honghu@psu.edu

ABSTRACT

Kernel use-after-free (UAF) bugs are severe threats to system security due to their complex root causes and high exploitability. We find that 36.1% of recent kernel UAF bugs are caused by improper uses of reference counters, dubbed *refcount-related* UAF bugs. Current kernel fuzzing tools based on code coverage can detect common memory errors, but none of them is aware of the root cause. As a consequence, they only trigger refcount-related UAF bugs passively and coincidentally, and may miss many deep hidden vulnerabilities.

To actively trigger refcount-related UAF bugs, in this paper, we propose CountDown, a novel refcount-guided kernel fuzzer. CountDown collects diverse refcount operations from kernel executions and reshapes syscall relations based on commonly accessed refcounts. When generating user-space programs, CountDown prefers to combine syscalls that ever access the same refcounts, aiming to trigger complex refcount behaviors. It also injects refcount-decreasing and refcount-accessing syscalls to intentionally free the refcounted object and trigger invalid accesses through dangling pointers. We test CountDown on mainstream Linux kernels and compare it with popular fuzzers. On average, our tool can detect 66.1% more UAF bugs and 32.9% more KASAN reports than state-of-the-art tools. CountDown has found nine new kernel memory bugs, where two are fixed and one is confirmed.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Vulnerability scanners**.

KEYWORDS

Use-After-Free (UAF); Reference Counting; Kernel Fuzzing

ACM Reference Format:
Shuangpeng Bai, Zhechang Zhang, and Hong Hu. 2024. CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690320

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690320

1 INTRODUCTION

Linux kernel has the highest privilege to access all computing resources, and therefore, attackers like to exploit kernel vulnerabilities to launch severe attacks such as privilege escalation [2, 5, 7, 26, 50] and information leakage [25, 45, 46, 51]. Among all kernel vulnerabilities, the sophisticated use-after-free (UAF) bugs are getting more attention from security researchers and kernel maintainers, since they are challenging to detect, analyze and fix [45, 47]. UAF bugs are invalid memory accesses via dangling references where referred memory objects have been released. To mitigate UAF bugs, Linux kernel relies heavily on reference counters (*refcount* for short) for automatic object management [27, 40]. A refcounted object contains a refcount instance that represents the number of active object references. A zero-value refcount means the object has no active reference, and the memory manager should release the object automatically. Refcount helps avoid UAF bugs as long as the refcount value is consistent with the number of active object references.

However, improper uses of refcounts can bring in new UAF bugs. For each refcounted object, developers should invoke proper APIs to increase the refcount when creating a reference and decrease the refcount when destroying a reference. If developers make a mistake, the number of references will be inconsistent with refcount. The memory manager may prematurely free an object when active references still exist. If the kernel uses dangling references to visit released objects, it will trigger UAF bugs [17]. We term them *refcount-related* UAF bugs. We investigate fixed memory errors reported by Syzbot in recent three years [42] to understand the prevalence of refcount-related UAF bugs. We consider a UAF bug as refcount-related if the incorrectly accessed object is freed by the refcount mechanism. We collect 342 memory errors, including 205 UAF bugs. 74 of them are refcount-related UAF bugs, accounting for 36.1% of all UAF bugs and 21.6% of all KASAN reports. This result confirms the prevalence of refcount-related UAF bugs.

Recent research efforts adopt coverage-guided fuzzing to test Linux kernel and detect thousands of bugs [42], including many UAF issues. The idea is to utilize code coverage to guide the generation of user-space programs, which invoke various syscalls to test the kernel code. For example, Syzkaller [43], the most popular kernel fuzzer, defines syscall grammars and utilizes relations among different syscalls to guide the program generation. It takes two factors to build syscall relations, specifically, (1) data flows across syscall arguments and return values, and (2) frequencies that two syscalls are used in the same program. When mutating a program, Syzkaller randomly selects one existing syscall and inserts a new syscall that is strongly related to the chosen one. If the new program triggers previously unseen kernel code, Syzkaller will save it into a queue for future mutations. Recent efforts [32, 37] adopt

```
int llcp_sock_bind(...) {
  // reference += 1; counter += 1;
  llcp_sock->local = nfc_llcp_local_get(local);
  // ???;            counter -= 1;
  nfc_llcp_local_put(llcp_sock->local);
+ // reference -= 1;
+ // llcp_sock->local = NULL; <-- patch
 }
```

Figure 1: Buggy reference counting in kernel

```
void main(void) {          // reference (1)    counter (1)
  int sock1 = socket(...); //       +0 => 1        +0 => 1
  int sock2 = socket(...); //       +0 => 1        +0 => 1
  bind(sock1, &addr, ...); //       +1 => 2     +1,-1 => 1
  bind(sock2, &addr, ...); //       +1 => 3     +1,-1 => 1
  close(sock1);            //       -1 => 2       -1 => 0 (free)
  close(sock2);            //  use-after-free bug
}
```

Figure 2: User-space program triggering a kernel bug

```
void main(void) {
  int sock1 = socket(...);
  bind(sock1, &addr, ...);
  close(sock1);
}
// simple program in corpus
// that tests the function
// of socket, bind and close
```

Figure 3: A simple program

static and dynamic program analyses to refine syscall relations, aiming to generate unique syscall combinations to maximize the code coverage. Meanwhile, researchers try to construct accurate grammars to generate more valid syscalls, like through manual efforts [43], static analysis [16, 21, 38], or dynamic analysis [9].

However, current feedback focuses on code coverage [32, 37, 43] and only detects kernel UAF bugs passively and coincidentally during the process of code exploration, whereas many UAF bugs are irrelevant to reaching new code. Especially for refcount-related UAF bugs, triggering them requires to execute various refcount operations many times to reduce the refcount to zero before the kernel frees a vulnerable object. These necessary repetitions usually do not trigger new code or branch, and thus are not favored by current feedback mechanisms. Figure 1 shows the vulnerable code of `bind` related to the UAF bug CVE-2021-23134 [3]. In general, `llcp_sock_bind` forgets to destroy an obsolete reference but correctly reduces the refcount. Figure 2 provides a proof-of-concept (PoC), which invokes different syscalls at least two times before kernel AddressSanitizer (KASAN) [22] captures the UAF bug.

To address the limitation of code coverage-based feedback, in this paper, we propose CountDown, a novel technique that leverages refcount operations to guide program generation to promptly expose kernel temporal memory bugs, such as use-after-free and double free. Our insight is that the refcount mechanism is a substantial root cause of many UAF bugs (see Figure 5), and therefore, diversifying refcount operations will have higher opportunities to expose refcount-related UAF bugs. Specifically, we collect refcounts accessed by different syscalls, and utilize the number of commonly accessed refcounts (or *shared refcounts*) to build syscall relations. We encourage combining syscalls that ever operate on more shared refcounts, in the hope of triggering more sophisticated refcount operations that will ultimately lead to refcount-related UAF bugs.

CountDown contains four key components. First, it will dynamically learn refcount-based relations between different syscalls. We instrument Linux kernel to record all accessed refcounts for each executed syscall, and identify commonly visited refcounts among different syscalls to calculate their relations. Second, it utilizes the refcount-based relation to help generate new programs. At the early stage of fuzzing, we utilize previous coverage-based relation to boost the code coverage. As time goes on, we set higher priority to refcount-based relations such that the code generation will focus on exploring new operations on refcounts. Third, when measuring the quality of generated programs, we take newly visited refcounts by each syscall into consideration. Such unseen (syscall,refcount) pairs indicate new access patterns and we will update refcount-based relations accordingly. At last, we use refcount-based relations to customize the program towards triggering UAF bugs. Particularly, we insert multiple refcount-decreasing syscalls to the program to reduce the refcount to zero, which forces the memory manager to free the refcounted object. We also insert syscalls that ever access the same refcount in order to trigger invalid after-free uses.

We build a prototype of CountDown based on Syzkaller, embedding our designs of refcount-guided fuzzing. Our implementation consists of 477 lines of C code, 751 lines of Go code, and 190 lines of Python code. We also modify 438 lines of C code in Linux kernel for the instrumentation purpose. Currently, CountDown supports testing three mainstream Linux kernels, *i.e.*, v5.15, v6.1 and v6.6. Since our design is general, CountDown should work on any version with moderate efforts to adopt the instrumentation.

We evaluate CountDown on three different Linux kernel versions and compare it with the state-of-the-art kernel fuzzers, including Syzkaller, MoonShine, and Actor. During 72-hour fuzzing evaluations, CountDown produces at least 66.1% more UAF bugs and 32.9% more KASAN reports than other tools. It can effectively learn 110k new (syscall,refcount) pairs on average, demonstrating its capability of continuously learning and leveraging refcount operations for fuzzing. We report nine previously unknown kernel memory errors, where two of them are fixed and one is confirmed.

In summary, we make the following contributions.

- We propose the idea of refcount-guided fuzzing to test Linux kernel which focuses on exposing sophisticated temporal memory safety issues.
- We implement CountDown that constructs and utilizes refcount-based relations to guide the generation of user-space programs for triggering refcount-related UAF bugs.
- We evaluate our implementation on real-world Linux kernels. CountDown outperforms existing tools by exposing 66.1% more UAF errors and 32.9% more KASAN reports. It has detected nine new kernel memory errors.

Open Source. We will release the source code of CountDown at https://github.com/psu-security-universe/countdown.

2 BACKGROUND AND PROBLEM

We first introduce the refcount mechanism used in Linux kernel. Then, we present the refcount-related UAF issue and investigate their prevalence. At last, we examine current fuzzing techniques and discuss their limitations in detecting refcount-related UAF bugs.

2.1 Refcount in Linux Kernel

Linux refcount mechanism is part of the kernel memory management infrastructure [11, 23]. It is designed to track the number of active references of each refcounted kernel object. Figure 4 shows the common usage of refcount in Linux kernel. First, the kernel defines two sets of general structures and APIs to encapsulate refcount details, specifically, `refcount_t` and `kref`. The latter is a wrapper of the former, and both are widely used. For example, `kref_get` increases the internal refcount by one, while `kref_put` reduces the

```c
// general structures and functions
typedef struct refcount_struct { atomic_t refs; } refcount_t;
struct kref { refcount_t refcount; };
void kref_get(struct kref *kref) { refcount_inc(&kref->refcount); }
int  kref_put(struct kref *kref, void (*release)(struct kref *kref)){
  if (refcount_dec_and_test(&kref->refcount)) { release(kref);   return 1; }
  return 0;
}
// one concrete kernel structure with refcount
struct nfc_llcp_local { struct list_head list; struct kref ref; ... };
struct nfc_llcp_local * nfc_llcp_local_get(struct nfc_llcp_local *local) {
  kref_get(&local->ref);
  return local;
}
int nfc_llcp_local_put(struct nfc_llcp_local *local) {
  if (local == NULL) return 0;
  return kref_put(&local->ref, local_release);
}
```

Figure 4: Definitions and usages of refcount in Linux kernel. `kref` structure and APIs are merely wrappers of `refcount_t` type.

refcount and if the refcount reaches zero, invokes the callback function `release` to free the refcounted object. Refcounted kernel structures will include a refcount member, like the `ref` member of struct `nfc_llcp_local`, and define two related APIs, one `put` function and one `get` function. When an instance of a refcounted structure is initialized, the kernel will invoke `kref_init` to set the initial refcount value, which internally calls `refcount_set`. When a new reference is created, developers will use the `get` function (`nfc_llcp_local_get` here) to increase the refcount. When a reference is destroyed, developers will call the `put` function (`nfc_llcp_local_put` here) to decrease its refcount, and if necessary, release the object. In this way, developers invoke proper APIs around reference operations to update refcount locally, while the memory manager will release unnecessary objects and enforce temporal memory safety globally.

Linux developers are adopting the refcount mechanism to manage more kernel objects. Our measurement reveals that the number of kernel structures containing `refcount_t` or `kref` members is increased from 617 in kernel 4.19 to 1039 in kernel 6.8. Recent work CID [40] investigated the refcount usage in Linux v5.6-rc2. Among 792 refcount fields, 644 have type `refcount_t` or `kref`, accounting for 81.3%, while others are atomic variables. Given the high coverage and growing usage of `refcount_t` and `kref`, in this paper, we focus on `refcount_t` and `kref`, and ignore atomic variables.

2.2 Refcount-related Use-after-free

Refcount works well if the refcount value is consistent with the number of active references. However, developers may make mistakes when handling references and refcount, leading to imbalance and bugs. When a refcount is smaller than the reference number, the manager will release objects prematurely; if the kernel accesses the freed object via an active (but dangling) reference, it will trigger a UAF bug. When a refcount is larger than the reference number, the manager cannot release the unused object, leading to memory leaks. Refcount-related bugs are common in Linux kernel and user-space applications adopting refcount, like browsers [24, 27, 29, 40]. Developers are encouraged to skip refcount operations for better performance, which exacerbates refcount-related UAF bugs [17, 30].

Figure 1 shows the simplified `bind` syscall that contains the UAF bug CVE-2021-23134 [3]. At line 3, the code creates a new reference `llcp_sock->local` of a `nfc_llcp_local` object and invokes `nfc_llcp_local_get` to increase the refcount. Both the reference

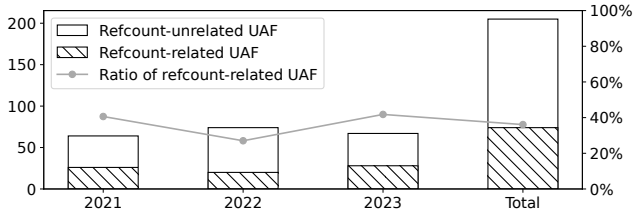


Figure 5: Statistics of use-after-free bugs in Linux kernel. We investigated all fixed memory errors reported by Syzbot in the past three years.

number and the refcount are increased by one. After completing this reference use, the kernel should destroy it and reduce the refcount. The decrement is done correctly by invoking `nfc_llcp_local_put`, but the code leaves the reference alive, making the reference number and the refcount inconsistent. Figure 2 is the proof-of-concept (PoC) for triggering this bug. During kernel booting, it creates one reference of the object and sets the refcount to 1. Syscall `socket` does not change the reference or the refcount. At line 4, syscall `bind` calls the vulnerable `llcp_sock_bind` in kernel. This syscall leaves one reference in the kernel (*i.e.*, two references in total), but keeps refcount as 1. The second invocation of `bind` leads to three references but the refcount remains 1. Syscall `close` will destroy one reference and decrease the refcount, leading to a zero refcount. The memory manager will release the object which makes the remaining two references dangling. The second `close` accesses the freed object via one dangling pointer, triggering the UAF bug.

To understand the necessity of exposing refcount-related UAF bugs, we investigate all Linux memory errors that are reported by Syzbot and get fixed in recent three years [42]. We treat a UAF bug as refcount-related if the incorrectly accessed kernel object is released automatically by the kernel refcount mechanism. Since refcount should help avoid UAF bugs among refcounted objects, such bugs will happen only if something is wrong within refcount operations. Figure 5 shows our investigation results. Each bar indicates the number of UAF bugs; the shaded portion highlights refcount-related ones; the curve shows the ratio of the latter among the former. We collect 342 memory errors, including 205 UAF bugs, 95 out-of-bound accesses, and 42 other issues. 74 UAF bugs are refcount-related, accounting for 36.1% of all UAF bugs and 21.6% of all memory errors. These results show that refcount-related UAF bugs are emerging, prevalent, and severe threats to kernel security. We should develop techniques to expose these vulnerabilities as early as possible.

2.3 Existing Techniques of Kernel Fuzzing

Recent research efforts extensively explore fuzzing techniques to test the Linux kernel, and successfully detect thousands of bugs [14, 32, 37]. The basic idea of fuzzing is to generate unexpected test cases and feed them into the tested program to trigger crashes and other abnormal behaviors [15, 28, 31, 49]. A kernel fuzzer will construct user-space programs that invoke different system calls to reach abnormal states in kernel space. For example, Syzkaller [43] is the most popular open-source coverage-guided kernel fuzzer and serves as the basis for many advanced kernel fuzzers. It generates and executes test cases in virtual machines to test Linux kernel.

Meanwhile, it adopts various kernel sanitizers, like KASAN and KMSAN [6, 22] to catch abnormal events and produce bug reports.

Syscall-relation Learning. To trigger complicated kernel states, Syzkaller identifies and invokes related syscalls together in user-space programs. It calculates syscall relations through static analysis and dynamic analysis. We call this relation *SyzRelation*. First, Syzkaller considers the data-flow dependency between syscall arguments and return values. If one syscall writes to an argument or returns a value of type T while another syscall requires an argument of the same type, Syzkaller will set a higher value to this relation of these two syscalls. Since this relation is determined by the syscall prototype, it will not change unless Linux kernel developers update the prototype. Therefore, Syzkaller names it static relation and we use $StaticPrio$ to represent it. Second, Syzkaller counts the frequency of two syscalls used together in the seed corpus. If two syscalls are used together in one program, Syzkaller will add 1 to the relation of these two syscalls. This relation will be different for various seed corpus. Therefore, Syzkaller calls it dynamic relation and we use $DyanmicPrio$ to represent it. After normalization, Syzkaller merges these two relations to produce the $SyzRelation$ for each syscall pair, using the following formula[1].

$$SyzRelation = StaticPrio * DynamicPrio \quad (1)$$

To generate a new user-space program from one existing test case, Syzkaller randomly selects a syscall, say A, and inserts another syscall that has a high $SyzRelation$ value with A.

Several following-up works propose advanced mechanisms to learn more accurate relations. MoonShine [32] reduces sample programs to a minimal set that can trigger the same code coverage. With the reduced corpus, it can boost the fuzzing process quickly. Healer [37] removes syscalls one by one and checks which of remaining syscalls have different execution traces. It constructs relations between the removed syscall and the affected ones. Actor [14] collects heap actions triggered by each syscall during the program execution, like memory allocation, deallocation and access. It creates specific templates for particular bugs, and fills the template based on the relations between syscalls and heap actions.

Input Prioritization. Modern fuzzers commonly adopt code coverage to guide the input selection and prioritization [8]. In particular, if a randomly generated test case triggers previously unseen code or branch [44], the fuzzer will allocate more energy (*i.e.*, CPU cycles) to mutate it. The assumption is that exploring more code brings a higher probability of triggering bugs. Kernel fuzzers [32, 37, 43] also adopt similar feedback to prioritize user-space programs that trigger new code coverage. For example, Syzkaller collects branch coverage, called signal, to identify interesting new programs.

2.4 Limitations of Previous Methods

Previous syscall relations and input prioritization fall short in exposing refcount-related UAF issues. Syzkaller and MoonShine calculate syscall relations based on syscall signatures and their common usages. These high-level features can hardly capture low-level refcount operations, and thus cannot help expose refcount-related UAF bugs. Healer [37] infers syscall relations based on the change of triggered code. However, triggering refcount-related UAF bugs usually requires repeated executions of multiple refcount-related syscalls, which may not trigger new code. Actor [14] maps each syscall to their operated heap objects, which could help trigger traditional UAF bugs. But for refcount-related bugs, a heap action is merely the consequence of multiple complex refcount operations, like object release triggered by zero-value refcount. Actor only focuses on the consequence and has no awareness of the root cause, leading to several limitations. First, Actor relies on a template (allocate, free and use) to find UAF bugs, which frees the allocated object immediately without introducing complex refcount operations. Second, many syscalls access a few kernel objects and thus get prioritized by Actor, but only part of them operate on refcounts. Third, Actor records syscalls that free operations, and assumes that these syscalls will free the same objects in new programs. However, to release a refcounted object, we need to invoke many refcount-decreasing syscalls and all of them are necessary.

We use our motivating example to demonstrate these limitations. Suppose we start with the code in Figure 3, and our goal is to produce the PoC in Figure 2 which triggers the kernel UAF bug CVE-2021-23134 [3]. The vulnerable implementation is shown in Figure 1 and is discussed in §2.1. To generate the PoC, the fuzzer has to combine `bind` and `close` and call them twice separately. The relation adopted by Syzkaller identifies strong relations between `socket`, `bind`, `close`, `listen` and `accept` due to parameter types and their common appearances. It will miss the special relation between `bind` and `close` for triggering this refcount-related UAF bug. Healer's relation favors syscalls that trigger new coverage. However, adding one more `bind` or `close` to Figure 3 does not change the kernel code coverage and will not be prioritized by Healer. Actor can identify that `socket` creates an object, `bind` accesses it and `close` frees it. It will fill the template (allocate, free, use) with proper syscalls, but can hardly trigger complicated refcount operations. Moreover, Actor will identify many syscalls that access the same object, and will use all of them regardless of their operations to refcounts. Further, once Actor detects that syscall `close` releases this object, it will use syscall `close` in the following mutations to release the same object, ignoring another prerequisite that the refcount has to be 1.

The previous method of input prioritization is also insufficient to trigger refcount-related UAF bugs. It will drop newly generated programs that do not trigger unseen code even if they have unique refcount operations. For refcount-related UAF bugs, even if a refcount operation has been covered before, triggering it again via a different syscall demonstrates a unique unseen interplay and should be used for future mutation. For instance, function `nfc_llcp_local_put` may decrease the refcount of a `nfc_llcp_local` object based on runtime context. Both syscall `bind` and syscall `close` invoke this function to reduce the refcount. However, these two syscalls have distinct semantics and thus decrease the refcount for different purposes. Even if function `nfc_llcp_local_put` has been covered by previous executions of syscall `bind`, we should treat its invocation by syscall `close` as interesting since combining these two syscalls results in complex refcount operations and triggers a UAF bug.

[1] Syzkaller developers updated this formula on April 11, 2024 to replace multiplication with addition. We used the old version (*i.e.*, multiplication) when building our tool.

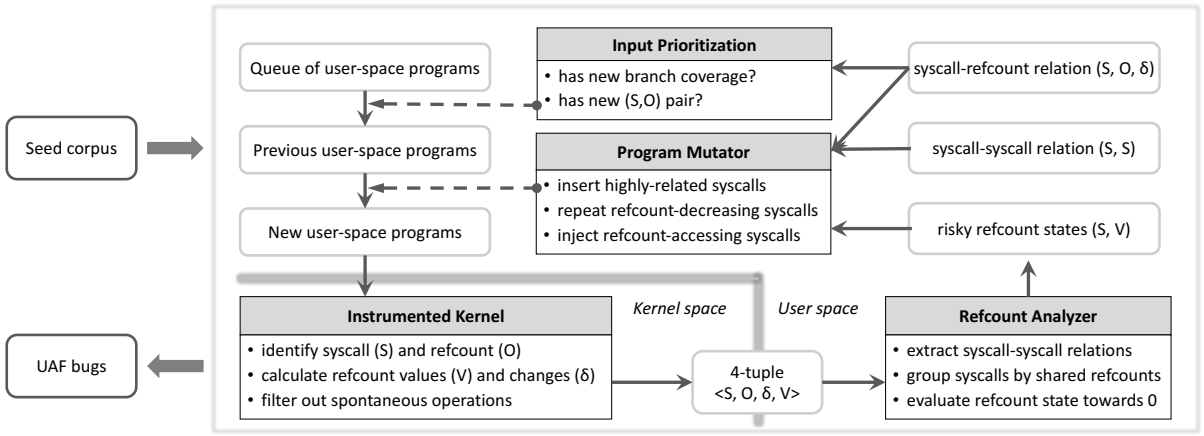


Figure 6: Overview of CountDown. Given a set of user-space programs, we will collect kernel execution traces to understand how each syscall updates refcounts. We use this information to build a new relation between syscalls and use the new relation to guide input prioritization and mutation.

3 APPROACH OVERVIEW

To promote the detection of refcount-related UAF bugs from Linux kernel, we propose *refcount-guided fuzzing*, a new mechanism that leverages refcount operations to assist the mutation and prioritization of user-space programs. Our insight is that refcount operations are the root causes of many heap actions of refcounted objects, especially for the critical object deallocation. Comparing with other feedback mechanisms, like new code coverage [37, 43] or unseen heap actions [14], refcount operations capture more accurate and fine-grained changes within the heap object management. Therefore, the guidance is likely to trigger UAF bugs related to refcounts.

To achieve the goal of refcount-guided fuzzing, we will build new syscall relations based on refcounts accessed by multiple syscalls, called *shared refcounts*. During the mutation, we will combine syscalls that access more shared refcounts, with the hope of triggering complex and error-prone refcount operations. After the execution, we will check whether the execution triggers new refcount operations, and if so, set a high priority to mutate this program.

We use the program in Figure 3 to demonstrate how the refcount-based guidance helps trigger refcount-related UAF bugs. A high-quality corpus for Linux kernel may contain this simple program. Current fuzzers like Syzkaller can generate this simple code based on Syzlang grammar. After executing this program, we will find a refcount, say O, accessed by both `bind` and `close`. In particular, `bind` first increases the refcount of O and then decreases it, while `close` merely decreases the same refcount. With the refcount-based guidance, our method will mutate this simple program in several novel ways. First, due to the shared refcount, we will assign a higher relation to the syscall pair (`bind`, `close`). Then, we randomly choose one refcount accessed by this program and one syscall that accesses this object, which could be O and `bind`. After that, we will insert another syscall that ever accesses O, like `bind` and `close`. Next, since `close` decreases the refcount of O, we will repeat it multiple times to reduce the refcount value to zero, which forces the kernel to free O. Following that, we will insert more syscalls that ever access O, like `bind` and `close`, hoping to trigger a use-after-free bug. With the new mutation strategy, we will likely generate the PoC in Figure 2, and trigger the refcount-related UAF bug CVE-2021-23134.

4 DESIGN OF COUNTDOWN

We design a system, CountDown, to test our idea of refcount-guided fuzzing for detecting refcount-related UAF bugs from Linux kernel. CountDown contains four main components, shown in Figure 6. First, it instruments Linux kernel to record refcount operations and maps them to corresponding syscalls (§4.1). Second, based on the execution trace, CountDown calculates the refcount-based syscall relations (§4.2). Specifically, if two syscalls ever access the same refcount, called *shared refcount*, we will assign them a higher relation value. Third, CountDown uses the new refcount-based relation to generate interesting user-space programs to trigger refcount bugs, release vulnerable objects and reach invalid memory accesses (§4.3). Fourth, for each newly generated program CountDown will save it for future mutation only if its execution triggers either new code coverage or new refcount operations (also in §4.3). Next, we explain each component in detail.

4.1 Recording Refcount Operations

To build refcount-based syscall relations, CountDown first records every refcount operation and its triggering syscall. We design each record to be a 4-tuple <S, O, δ, V>, where syscall S updates the refcount O of a refcount. by δ. After the update, the refcount value reaches V. To collect refcount operations, we instrument the general refcount APIs used in Linux kernel, as shown in Table 1. As we discuss in §2.1, another set of APIs on `kref` are simply wrappers of `refcount_t` APIs. We only hook functions that directly update refcount_t values and exclude others to avoid duplicated records.

4.1.1 Identifying Refcount O. Kernel fuzzing involves multiple parallel tasks running in different virtual machines (VM). To merge records obtained from different VMs, we need a proper way to identify and match refcounts. Within refcount functions such as `refcount_inc`, we only have the address of the refcounts, which will be different in multiple fuzzing instances due to the non-deterministic heap management and address space layout randomization (ASLR) [33]. To solve this problem, we observe that all refcounts are initialized either by the `refcount_set` function or via the `REFCOUNT_INIT` macro. The latter is used to initialize a refcount

Table 1: Instrumented refcount APIs in the Linux kernel. We ignore APIs of `kref` since they are merely warppers of `refcount_t` APIs.

Name	Description
`refcount_set`	set refcount to the given value
`refcount_add`	add the given value to refcount
`refcount_inc`	increase refcount by 1
`refcount_dec`	decrease refcount by 1
`refcount_add_not_zero`	add given value to refcount if refcount is not 0
`refcount_inc_not_zero`	increment if the refcount is not 0
`refcount_dec_not_one`	decrement if the refcount is not 1
`refcount_dec_if_one`	decrement if the refounct is 1
`refcount_dec_and_test`	decrement refcount and check if result is 0
`refcount_sub_and_test`	subtract value from refcount and check it with 0

for global refcounted objects. Since such objects cannot be freed, we ignore them during fuzzing and only focus on the ones initialized by `refcount_set`. The call stack of `refcount_set` indicates a unique purpose of the refcounted object. As long as we test the same kernel image, the call stack information will be consistent among multiple VMs. Within `refcount_set`, CountDown will calculate the checksum of the call stack and use the checksum as the cross-VM identity (ID) for this refcount.

CountDown creates a refcount-address-ID map (RAI map) to connect the addresses of refcounts to their refcount IDs. When one refcount is initialized, we create an entry in the RAI map, which consists of the refcount address and the ID. When the refcount reaches zero, we remove the corresponding entry from the map. For each refcount increment and decrement, we search the address in the map to find the corresponding ID and record the refcount operation. Many refcounts are initialized during the kernel booting and used later by fuzzing-invoked tasks. We record all of them into the RAI map to ensure CountDown can find the proper entry and record the correct refcount ID for each refcount operation.

4.1.2 Passing Syscall Number S. During fuzzing, one program invokes many different syscalls through multiple threads. To connect refcount operations with syscalls, we need to record the triggering syscall for each operation, specifically, the Syzkaller syscall number (NR). We design two helper syscalls and add a new member to the per-thread `task` structure. Before each original syscall, we invoke the helper syscall `before_sys` to stores NR of the original syscall into the new member of the current `task` structure. After the original syscall, we invoke another helper syscall `after_sys` to reset the member of `task` to a magic value. In this way, we can identify the syscall triggering each refcount operation. A magic value indicates the syscall is not triggered by fuzzer-generated programs.

Thread-safe Recording. We design our recording process to be thread-safe since Linux kernel utilizes multiple threads to execute many syscalls in parallel. First, we update the program mutator to add two helper syscalls around each original syscall and use the same thread to execute them sequentially. Since each thread has a distinct `task` structure in kernel, our method guarantees that the recorded syscall NR is the one triggering the original syscall. Second, we use spinlocks to protect global variables to avoid race conditions. Therefore, our recording process is thread-safe.

4.1.3 Recording Refcount Change δ and Final Value V. We also record how each syscall modifies every refcount, including the change δ and the final value V. One syscall may modify a refcount many times, but we only care about the overall impact δ. For each refcount API in Table 1, we modify it to record every single refcount change. Some refcount APIs check the refcount value to conditionally change the refcount. Their return values indicate whether the operation is successful or not [20]. Based on this semantic, we maintain a two-dimension table, with one dimension for the syscall number, and another dimension for the refcount ID. Each table entry records the current δ and the current V. Within each API function, we update the table to accumulate refcount changes. We use `before_sys` to reset the table, and leverage `after_sys` to dump the refcount changes and final values for the following analysis.

Example. For the program in Figure 3, CountDown will insert `before_sys` and `after_sys` before and after every syscall, leading to a new sequence of (`before_sys`, `socket`, `after_sys`, `before_sys`, `bind`, `after_sys`, `before_sys`, `close`, `after_sys`). After running this program within the instrumented kernel, we will collect two 4-tuples of refcount operations, (`#bind`, `hashX`, +0, 1) and (`#close`, `hashX`, -1, 0). `hashX` is the checksum of the call stack when the kernel initializes this refcount. Given the initial refcount value 1, syscall `bind` first increments and then decrements it, so the δ is 0 and the V is still 1. Syscall `close`, decrements the refcount by 1, so the δ is -1 and the V is 0. Syscall `socket` does not modify this refcount in this execution and we will not collect any record of it.

4.2 Reshaping Syscall Relation

Based on the records of refcount operations in the 4-tuple format (S, O, δ, V), we will construct multiple relations among syscalls and refcounts to help achieve refcount-guided fuzzing.

4.2.1 Refcount-syscall Relation. Since our fuzzing strategy focuses on refcounts, we first identify syscalls that ever change the refcount of any object. Given a set of 4-tuples, we will only use S, O and δ to fill the refcount-syscall relation map, where the key is the refcount ID, and the value is a set of syscalls and their changes to the refcount (*i.e.*, δ). We only consider refcount records within the same program execution for the relation learning, since it is unreliable to build relations between syscalls from different programs. The first two tables in Figure 7 demonstrate our method to get the refcount-syscall map. Five original records in the first table are classified into two groups in the second table, where three syscalls access the first refcount and two syscalls visit the second.

4.2.2 Syscall Relation Reshaping. Based on shared refcounts, we reshape syscall relations for refcount-guided fuzzing. We call our new syscall relation $RefcntRelation$. Our assumption is that if two syscalls access the same refcount within one execution, they have a stronger relation with respect to triggering more complicated refcount operations. For each refcount in the refcount-syscall relation map, we split its associated syscalls into a set of syscall pairs. The third table in Figure 7 shows the result. After that, we will create the syscall relation table, where each unique pair of syscalls has an entry, and the value is the number of shared refcounts accessed by both syscalls. For example, in the last table of Figure 7, S_0 and S_1 operate on two refcounts, and therefore, their relation value is 2.

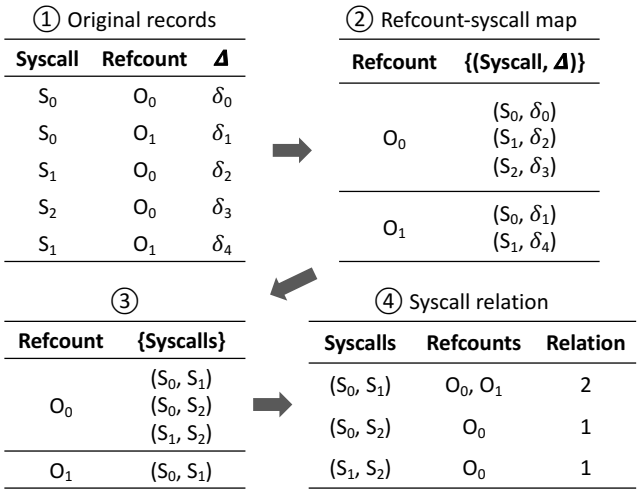


Figure 7: An example of refcount-based syscall relation calculation. All records of refcount operations come from the same program execution.

Compared with the traditional code coverage-based method, our new mechanism of calculating syscall relations focuses on refcount operation and will help detect refcount-related UAF bugs. However, refcount operations are sparsely distributed among the large kernel code base. Only using this relation as guidance may render the fuzzer miss important changes in code coverage. Especially at the beginning of kernel fuzzing, code coverage-based relation is important to help boost kernel exploration. Therefore, instead of completely dropping the previous relation, we merge it into our new method. At the beginning of fuzzing, we assign a high priority to the original coverage-based relation to quickly explore more kernel code. As time goes on, we will allocate more weight to our refcount-based relation such that the fuzzer can focus on testing complex refcount operations. We adopt the following formula to merge these two relations and calculate the overall relation.

$$OverallRelation = \log_2(SyzRelation) + k * \log_2(RefcntRelation)$$

First, *SyzRelation* is the syscall relation proposed by the popular fuzzer Syzkaller. We introduce this relation and discuss its calculation formula Equation 1 in §2.3. The high-level idea is setting stronger relations for syscall pairs with same-type arguments or used together in the corpus. With the guidance of *SyzRelation*, Syzbot has found thousands of kernel bugs, which demonstrates the effectiveness of *SyzRelation*. Therefore, we adopt it in our design. Second, *RefcntRelation* is the refcount-based relation we propose. It is the number of unique refcounts operated by a syscall pair, as we discuss in §4.2.2. The assumption behind this design is that if two syscalls operate on many unique refcounts, combining them together may bring more diverse behaviors. For each executed program, we record the triggered refcount operations, from which we extract *RefcntRelation*. Third, k is increased every time after testing a fixed number of test cases. At the beginning of fuzzing, k has a small value, where *SyzRelation* will contribute more to the *OverallRelation*. Then, as we increase k, *OverallRelation* will be influenced more by *RefcntRelation*, making the fuzzer focus more on combining syscalls that operate on the same refcounts.

Our goal here is to gradually increase the weight of the refcount-based relation, and therefore, any formula satisfying this goal should work. We pick up this formula since it is neat and should be adequate to demonstrate the benefit of the refcount-based relation. We leave the exploration of advanced algorithms to future work.

Relation Synchronization. Due to the non-determinism nature of fuzzing, kernel instances running in multiple VMs may learn different relations. When a new relation is received from one VM, other fuzzing instances should directly make use of it. CountDown synchronizes refcount relations across all fuzzing instances. The fuzzer process in each VM generates its own reports, consisting of each syscall pair and the accessed refcount IDs. It utilizes a relation-updating process in the host machine to periodically retrieve the reports from every instance, merge them, and distribute the merged relation to all instances. To merge syscall relations, CountDown will count the unique IDs operated by each pair in all reports. As we explain in §4.1.1, the refcount ID is calculated from the initialization call stack of the refcount, and therefore, can be used across multiple fuzzing instances that adopt the same kernel image.

4.3 Refcount-guided Program Mutation

CountDown makes full use of the new refcount-based relation to guide the mutation of user-space programs. First, we leverage the relations among syscalls to mutate programs towards triggering unexpected refcount behaviors, like missing refcount increments. Second, we use the refcount-syscall relation to help insert syscalls that induce the kernel to release refcounted objects, access them via dangling pointers, and finally trigger use-after-free bugs.

4.3.1 Diversifying Refcount Behaviors. To explore diverse refcount behaviors, we utilize two strategies to combine syscalls that have refcount-based relations. First, CountDown reuses the mutators from previous fuzzing tools, like Syzkaller, to mutate existing programs. Since CountDown allocates higher weights to the refcount-based relations, with our design the mutator will prefer to combine syscalls that are strongly related regarding refcount operations.

Second, we also create a new mutator that combines refcount-related syscalls more aggressively. ① We select one program as the target of mutation and run the program to collect all accessed refcounts. We will update the refcount-syscall map if the execution triggers new records (see §4.2.1). To mutate this program, we will randomly select one refcount from all collected ones. Refcounts accessed multiple times by this program will get higher probabilities to be selected. ② We search the refcount-syscall map to find all syscalls that ever operate on this refcount and select one for insertion. We set a higher priority to select related syscalls that are not used in the current program. ③ We reuse Syzkaller's logic for inserting the selected syscalls into the program, which prefers later positions. The probability of choosing a later position is incrementally increased by a fixed value compared to the probability of choosing the previous position. It ensures that the chance of selecting the last position is five times higher than that of selecting the first position. This approach is widely used by Syzkaller and other kernel fuzzers [10, 14, 19, 36]. We adopt this approach as the syscalls at the beginning may work on the initialization of resources

and refcounts, so these positions are not the ideal insertion locations. CountDown will go through both old mutators and the new one, and choose each mutator based on a predefined probability.

Refcount-based Input Prioritization. Other than traditional code coverage, CountDown also takes new refcount behaviors into consideration to identify promising programs. In particular, CountDown will execute every newly generated program within the instrumented kernel, and check whether the execution triggers new code coverage or a new refcount operation, *i.e.*, a previously unseen (syscall,refcount) pair. If so, it will save the new program into the input queue for further mutations. Meanwhile, it will update the refcount-syscall map and the syscall relation to absorb the new finding. In this process, we can synthesize programs with strongly related syscalls and trigger more complex refcount behaviors.

4.3.2 Refcount Bug Exposure. Refcount errors are important root causes of UAF bugs, but a triggered refcount error will not immediately result in a memory error. We need to trigger more operations to make the kernel (1) release the refcounted object and (2) access the object through dangling references. For example, in Figure 3, syscall `bind` triggers the refcount bug, which renders the refcount and the number of references inconsistent. However, if we stop here, there will be no object released, and also no use-after-free bugs. We should insert more refcount-related syscalls to the executed program to reduce the gap between a refcount bug and a UAF bug, such that kernel sanitizers like KASAN can capture it.

After executing a test case, CountDown checks the 4-tuple records (S, O, δ, V) of each syscall to identify refcount-decreasing syscalls and refcount-accessing syscalls. We consider the final value V of the refcount in the original program and the capability of changing refcount δ from each syscall, and insert a proper number of syscalls. Our goal is to reduce the refcount to 0, which will force the memory manager to release the object. In particular, we insert the syscall V times if its $\delta < 0$. After that, we will insert diverse syscalls that ever access the refcount, hoping to trigger invalid accesses through dangling references.

Example. To mutate the code in Figure 3, CountDown will first check the refcount-based relation and insert multiple `bind` and `close` syscalls since they ever access the same refcount. After that, since `close` ever reduces the refcount and accesses the object, we will insert more `close` syscalls. In this way, we can generate the PoC program in Figure 2, and trigger the UAF bug.

5 IMPLEMENTATION

We implement CountDown based on the popular kernel fuzzing tool, Syzkaller. Our implementation consists of 477 lines of C code, 751 lines of Go code, and 190 lines of Python code. First, CountDown changes 438 lines of C code in Linux kernel to implement the instrumentation for recording refcount operations. Second, we implement the refcount analyzer to calculate various relations between refcounts and syscalls. Third, we update the old program mutator and add a new one to realize our algorithms for combining refcount-related syscalls to trigger refcount issues, reduce refcounts and reach UAF bugs. Currently, CountDown supports testing three mainstream Linux kernels, v5.15, v6.1 and v6.6. Our general design of CountDown should work on other kernel versions with moderate efforts to adopt instrumentation.

Next, we discuss several key implementation details to the success of refcount-guided fuzzing, including selective recording for efficient execution, feature configuration to mitigate generic refcounts, and our support to multiple-core fuzzing.

5.1 Selective Recording for Efficiency

Since Linux developers extensively use refcount to manage heap objects, the kernel will run much slower if we record all refcount operations. We design three optimizations to reduce the overhead.

First, we observe that many refcount operations are not triggered by fuzzer-generated programs. Based on our design in §4.1.2, records of these operations will have the magic syscall NR and cannot be used for relation reshaping. Therefore, we identify such operations within refcount API functions and exclude them from recording. Second, Linux kernel may execute in the process context or the interrupt context. When an interrupt occurs during a fuzzing-invoked task, the kernel will switch to the interrupt context to handle the interrupt, but will not change the `task`. CountDown will keep recording operations within the interrupt context, which could be unrelated to fuzzing. We set CountDown to ignore all operations within the interrupt context. Third, we avoid recording any refcount operations introduced by our instrumentation, such as memory allocation and call stack hashing. Before running our code, we will temporarily set the current syscall NR in the `task` structure to the magic value and restore the value after recording.

Fuzzer-invoked tasks may create asynchronous child tasks within the kernel space, such as RCU and work thread. Recording refcount operations within such asynchronous tasks will introduce performance overhead since these tasks run much longer than their parent tasks. Our tool supports recording refcount operations within these asynchronous tasks, but we turn this option off by default to improve the kernel performance. This choice is consistent with the recent work that tracks operations on heap objects [14].

Other than selective recording, we design CountDown to avoid unnecessary repetitions. In particular, Syzkaller repetitively executes the same test case multiple times during fuzzing to filter out unstable code coverage and confirm new bugs. Collecting refcount operations in these repetitions brings in unnecessary overhead. Therefore, we only enable the recording for the first execution of a new test case, and disable recording for all following repetitions.

5.2 Generic Refcounts

CountDown leverages shared refcounts to learn relations among syscalls. However, not all shared refcounts provide useful guidance. During our study, we observe that particular refcounts are widely used by all syscalls and thus, add relations to each syscall pair. For instance, AppArmor is a security module designed for managing the permissions of accessing resources [1]. It hooks many resource-accessing functions from multiple subsystems, including file management, network access, and task scheduling. AppArmor creates one refcount `aa_label`, and conducts refcount operations during all syscalls. As a consequence, many irrelevant syscalls will operate this generic refcount, leading to unreasonable relations. Our test reveals that this single refcount contributes to more than 40% of all

Table 2: The p-values of all comparison experiments, where **green** ones are statistically significant. CD-NC indicates CountDown w/o corpus.

Kernel	Our tool	Other tool	UAF	KASAN	Cov.
v6.6	CountDown	Syzkaller	**0.032**	**0.008**	**6e-4**
v6.1	CountDown	Syzkaller	**0.023**	**0.037**	**2e-4**
v5.15	CountDown	Syzkaller	**0.003**	**0.039**	**2e-4**
v6.2	CountDown	MoonShine	**0.004**	**0.011**	**2e-4**
v6.2	CountDown	Actor	**7e-5**	**1e-4**	**2e-4**
v6.2	CD-NC	MoonShine	0.413	0.146	**2e-4**
v6.2	CD-NC	Actor	**4e-4**	**1e-4**	**2e-4**

syscall relations. To maintain the rationality of the refcount-based relations, we should ignore these commonly used generic refcounts. We implement a relation checker to count how many relations each refcount contributes to. If one refcount contributes to a lot of relations, we will investigate the case and ignore the corresponding relations. AppArmor is the only special case we find currently, but we will keep detecting generic refcounts with our relation checker to avoid similar issues in other kernel configurations.

5.3 Multiprocess Support

CountDown inherits the multiprocess feature from Syzkaller to support large-scale kernel testing. In each VM, CountDown runs M fuzzing processes, each of which executes one test case. Each process runs multiple threads in parallel, where each thread executes one syscall. Following the definition in Syzkaller, one test case can invoke up to 40 syscalls. When the program has more syscalls than threads, one thread may execute several syscalls sequentially, but different syscalls may not finish in the same order as they start. To store the refcount operations in parallel, we create $40 * M$ separate memory regions for all syscalls from all fuzzing processes running in the same VM. Then, inside the logging functions, we use the fuzzing process ID and the syscall number to identify the memory regions allocated for recording. After one program completes all its syscalls, we utilize the helper syscall `after_sys` to copy the in-kernel records to the user-space fuzzer, where the latter will analyze the trace and calculate various relations.

6 EVALUATION

We evaluate CountDown on main stream Linux kernel versions to understand its effectiveness for finding bugs, especially use-after-free issues. Our evaluation aims to answer the following questions.

Q1. Can CountDown detect more use-after-free bugs?

Q2. What is the contribution of each component of CountDown?

Q3. Can CountDown effectively learn refcount relations?

Q4. What is the instrumentation overhead of CountDown?

Experiment Setup. We conduct our experiments on a 64-bit Ubuntu 20.04 server with 56-core Intel(R) Xeon(R) Gold CPU (112 threads) and 500 GB memory. We compile the instrumented kernel and the original kernel with the same configuration. Since CountDown is built on top of Syzkaller, all experiments adopt the same setting, *i.e.*, running 16 virtual machines in parallel, where each machine utilizes 2 cores and 8 GB memory and creates 4 fuzzing processes. To mitigate the impact of fuzzing randomness, we run each fuzzing instance for 72 hours, repeat each experiment for 11 times[2] and report the average result, following the suggestion in the recent work [35]. Our setting allocates more CPU time to test each tool than previous works. In particular, each fuzzer is tested for 3 day * 16 VM * 2 CPU/VM = 96 CPU*days. As a comparison, Actor, Healer and MoonShine allocate 8, 2 and 4 cores to test each fuzzer for 24 hours, resulting in 8, 2, 4 CPU*days, respectively.

Table 3: Bug-finding results, averaged from 11 runs of 72-hour fuzzing.

Kernel Version	Use-after-free bugs CountDown/Syzkaller	All KASAN reports CountDown/Syzkaller
Linux 5.11	4.4/2.3 (91.3%↑)	11.5/9.4 (22.3%↑)
Linux 6.1	4.4/2.8 (57.1%↑)	10.4/8.0 (30.0%↑)
Linux 6.6	3.9/2.6 (50.0%↑)	6.3/4.3 (46.5%↑)
Average	66.1%↑	32.9%↑

Fuzzing Kernel Versions and Corpus. We choose three popular kernel versions for testing, including long-term versions v5.15 and v6.1, and the mainline version v6.6. Kernel maintainers are actively fixing bugs in these versions due to the potential threats to real-world devices. After the long-term large-scale kernel fuzzing, the security community has released several high-quality corpus consisting of a large number of test cases. These user-space programs help fuzzers go through kernel codes that have been extensively tested and quickly move on to under-explored ones. Without special instructions, we conduct all experiments with the initial corpus indicated in the Syzkaller Github repository [41].

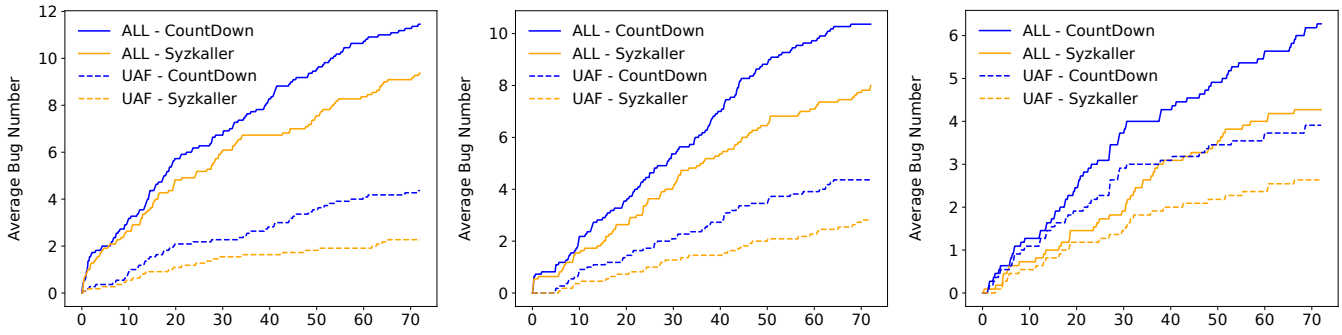
6.1 Bug Finding

6.1.1 Comparison with Syzkaller. We spend most evaluation effort on comparing CountDown with Syzkaller. First, our tool is built on top of Syzkaller, and therefore, any difference must stem from our refcount-related guidance. Second, both tools support three kernel versions and we can compare them comprehensively.

Table 3 shows the statistics of detected bugs averaged from 11 runs, while Figure 8 provides the bug-discovery progress over 72 hours. Overall, CountDown outperforms Syzkaller in finding UAF bugs and general memory errors. On different kernel versions, CountDown can detect 50.0% to 91.3% more UAF bugs than Syzkaller (66.1% on average), and reports 22.3% to 46.5% more general memory errors (32.9% on average). All the p-values are smaller than 0.05, as reported in Table 2. Such results demonstrate the advantage of our new refcount-related guidance for finding temporal and spatial memory errors.

The bug-discovery progress in Figure 8 reveals that at the beginning of fuzzing, CountDown performs similarly with Syzkaller, but gradually outperforms the latter as time goes on. The performance gap gets larger monotonically and there is no sign of convergence after the 72-hour evaluation. This result is consistent with our design in §4.2.2. In particular, at the initial stage of fuzzing we set a high priority to the original syscall relation to help explore more code paths. CountDown focuses on learning refcount-based relations and thus achieves similar performance as Syzkaller. After that, we assign a higher priority to the refcount-based relation.

[2]We notice that several p-values are slightly greater than 0.05 with 10 repetitions, so we run it one more time to obtain reliable results.



(a) experiments on v5.15 **(b)** experiments on v6.1 **(c)** experiments on v6.6

Figure 8: Bugs detected by CountDown and Syzkaller on three kernel versions. All numbers are average from 11 runs of 72-hour experiments. All the comparisons are statistically significant (p-values < 0.05).

CountDown will generate user-space programs that consist of more refcount-related syscalls, and trigger more complex temporal memory bugs. We find similar patterns in previous relation-learning fuzzers such as Healer [37] which exhibit performance comparable to Syzkaller until they accumulate sufficient information.

Although our major design goal is to effectively detect UAF bugs, during the evaluation CountDown also triggers more spatial memory bugs, like NULL pointer deference, out-of-bound access, and even direct access to user-space memory. Our understanding is that since the refcount operations are always associated with reference creation, destruction and object access, combining diverse refcount behaviors will also trigger complex memory operations, which leads to higher probabilities to trigger spatial memory bugs.

6.1.2 Comparison with Advanced Fuzzers. Next, we try to compare CountDown with state-of-the-art advanced kernel fuzzers, such as MoonShine, Healer and Actor. Since these tools have different settings and requirements on seed corpus and kernel versions, we try to identify the best configuration that works for every tool. (1) MoonShine designs a private format (*i.e.*, publicly unknown) to record kernel executions, and distills execution traces only in this format to produce a minimal set of high-quality programs. We have to reuse the corpus released by the developers to test MoonShine. (2) Healer learns syscall relations through dynamic analysis. However, the publicly released version lacks essential components [4], and can hardly find any bugs, as shown in the previous work [14]. Therefore, we exclude it from the comparison. (3) Actor is the most recent work for finding kernel memory bugs. Actor releases its instrumentation code for two particular kernel versions, v5.17 and v6.2-rc5. we choose the recent version v6.2-rc5 for testing all tools. We also follow the suggestion from Actor developers to patch the config file before compiling the kernel [13]. In summary, we compare the performance of CountDown with two advanced kernel fuzzers, MoonShine and Actor, on testing kernel v6.2-rc5. We test Actor without any corpus since VMs fuzzed by Actor will have frequent out-of-memory (OOM) issues with a large corpus (confirmed by Actor authors). We feed MoonShine its own released corpus, following the same practice in previous works [14, 37]. We test CountDown with two settings, one without any corpus (for comparison with Actor) and another with a corpus from Syzbot.

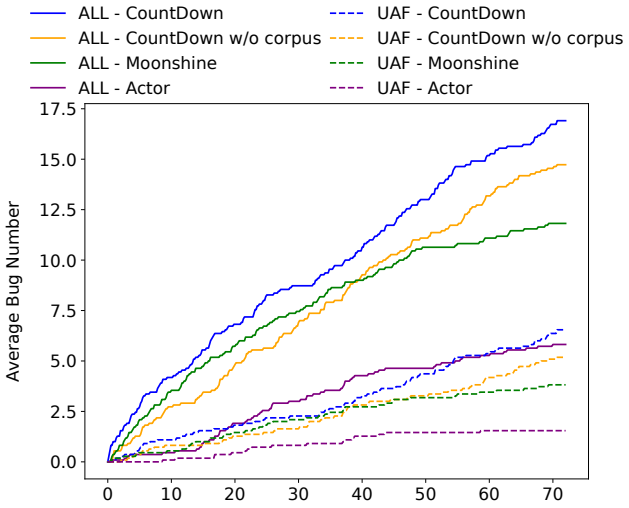
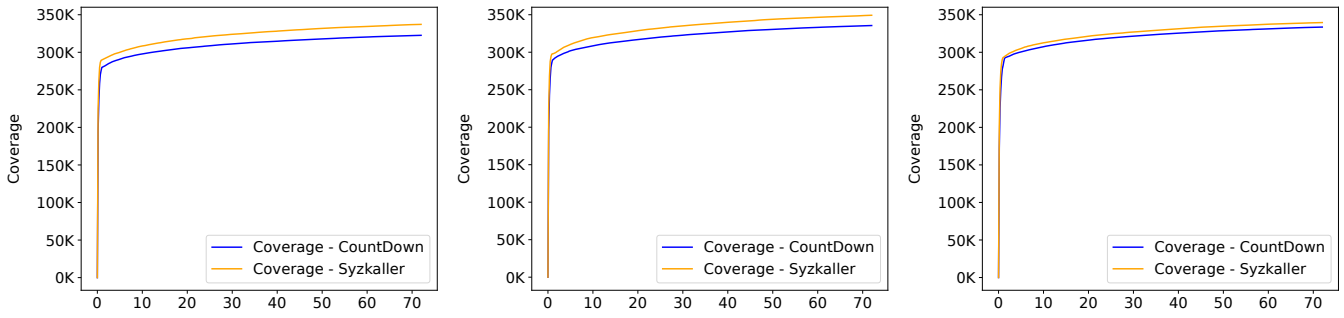


Figure 9: Bugs detected by CountDown and other advanced tools.

Figure 9 shows the results. With the seed corpus from Syzbot, CountDown detects 6.5 UAF bugs (average number across 11 72-hour runs) and 16.9 general memory errors. Without any corpus, it still reports 5.2 UAF bugs and 14.7 memory errors. To avoid getting extra benefit from the high-quality corpus. we use the result of the non-corpus CountDown to compare with others.

With its own corpus, MoonShine reports 3.8 UAF bugs and 11.8 memory errors, respectively. Our non-corpus CountDown outperforms MoonShine by 36.8% for UAF bugs and 24.6% for general errors. Under this setting, CountDown achieves similar performance as MoonShine within the first 60 hours. After CountDown learns sufficient refcount-based relations, it gradually outperforms MoonShine. We notice that the p-value keeps reducing along this process. Although it is greater than 0.05 at 72 hours (shown in Table 2), considering the stronger growth trend of our method. we believe our strength will be more significant in longer fuzzing. Considering the importance of high-quality corpus to kernel fuzzing, the extra bugs found by non-corpus CountDown suggest the strong advantage of the refcount-based feedback. Besides, we suggest to



(a) experiments on v5.15 **(b)** experiments on v6.1 **(c)** experiments on v6.6

Figure 10: Coverage triggered by CountDown and Syzkaller on three kernel versions. All numbers are average from 11 runs of 72-hour experiments. All the comparisons are statistically significant (p-values < 0.001).

Table 4: New kernel bugs detected by CountDown

Bug	Ver	Status
KASAN: slab-use-after-free in nfc_alloc_send_skb	v6.3	Fixed
UBSAN: shift-out-of-bounds in net/nfc/nci/core.c	v6.2	Fixed
KASAN: slab-use-after-free in __lock_acquire	v6.6	Confirmed
KASAN: use-after-free in gfs2_evict_inode	v4.19	Reported
KASAN: slab-out-of-bounds in sock_sendmsg	v6.1	Reported
KASAN: slab-out-of-bounds in __crypto_xor	v4.19	Reported
KASAN: slab-out-of-bounds in ext4_search_dir	v4.19	Reported
KASAN: slab-out-of-bounds in xfs_iext_get_extent	v4.19	Reported
KASAN: null-ptr-deref in mutex_lock	v4.19	Reported

equip CountDown with high-quality seed corpus, like Syzbot corpus, to maximize its performance.

Our non-corpus CountDown detects 2.47x more UAF bugs and 1.53x more memory errors than Actor with statistical significance (p-values in Table 2), which significantly outperforms this recent work. To ensure the reliability of Actor result, we have discussed with Actor developers several times. We adopt a proper setting confirmed by Actor developers and also test the latest Actor with its newest stability patch. With the help of Actor developers, we collectively discover that developing tools based on different versions of Syzkaller could be the root cause of unexpected Actor performance. Syzkaller developers have made several updates in recent years which significantly enhance the performance of Syzkaller. Actor was developed on top of an older version of Syzkaller (June 2022), while other fuzzers in this experiment get benefited from the newer Syzkaller (March 2023). Actor developers share their findings that MoonShine based on a newer version of Syzkaller performs obviously better than Actor in their latest tests, which is consistent with our findings. Furthermore, the results of the comparison to cutting-edge tools demonstrate that our tool is one of the best kernel fuzzers currently available, by combining advanced fuzzing techniques with the novel refcount feedback.

Static Refcount-bug Detectors. Researchers propose several static analysis techniques to detect refcount bugs from Linux kernel [27, 40]. Since these tools have not publicly released the source code, we cannot compare our tool with them. We will discuss the differences between static detectors and dynamic fuzzers in §7.3.

6.1.3 New Bugs Found by CountDown. Table 4 shows the new bugs detected by CountDown, including three UAF bugs, five out-of-bound access and one NULL pointer dereference. We have reported all of our findings to Linux developers, together with corresponding reproducers and kernel configurations. At the time of paper writing, Linux developers confirmed three bugs, fixed two of them, and are working on patching another one. We are still waiting for developers' responses to other reports. Among three new UAF bugs, two are refcount-related, *i.e.*, the freed object is managed by the refcount mechanism. These results show that with the refcount-related guidance, CountDown can detect previously unknown refcount-related temporal bugs from Linux kernel.

Bug Case Study. `slab-use-after-free` in `nfc_alloc_send_skb` is one new refcount-related UAF bug detected by CountDown. In function `nfc_llcp_recv_connect`, developers create a new reference to object `nfc_dev` and store the reference in a `sock` object. Although `nfc_dev` is managed by the refcount mechanism, developers forget to increase the refcount for this new reference, making the refcount inconsistent with the number of references. When kernel closes the device, it decreases the refcount (non-zero due to other refcount operations) of `nfc_dev` to zero, making the object released. Later, the dangling reference stored in `sock` is used for sending `nfc_llcp` messages, which triggers the UAF bug. Since the root cause of this UAF bug is an improper use of refcount, developers increment the refcount when creating the new reference. Within the same patch, developers also complement another refcount increment for another reference creation within function `nfc_llcp_register_device`, where a reference of `nfc_dev` is stored in an `nfc_llcp_local` object. An interesting observation is that five months after our report to the Linux community, Syzbot independently discovered and reported this vulnerability. Considering the large amount of resources used by Syzbot, we attribute our early discovery to the refcount-based guidance.

6.2 Code Coverage

Other than bug number, we also collect the code coverage triggered by different kernel fuzzers. Figure 10 shows the comparison between CountDown and Syzkaller. The coverage of CountDown is 4.3%, 3.9% and 1.8% lower than Syzkaller for kernel v5.15, v6.1 and v6.6, respectively. As shown in Table 2, all p-values are smaller

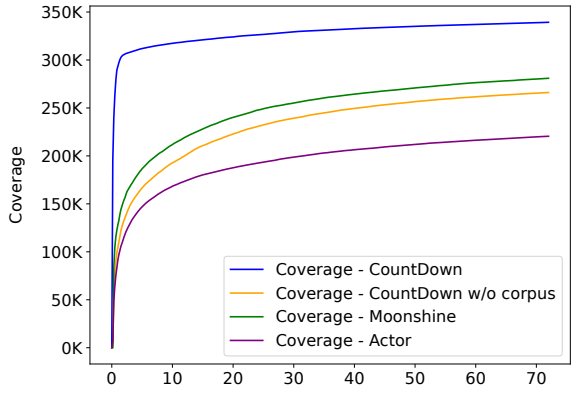


Figure 11: Coverage triggered by CountDown and advanced tools. Comparisons of any two tools are statistically significant (p-values < 0.001).

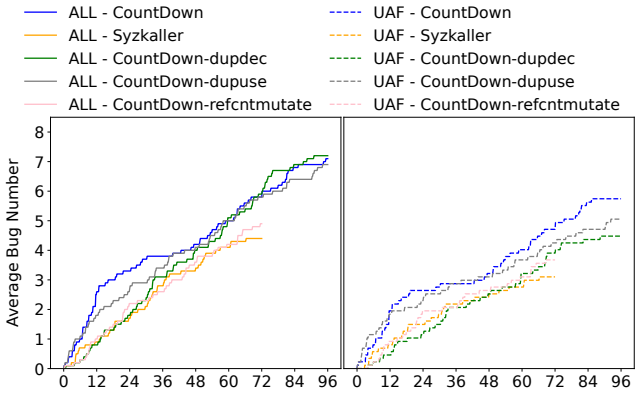


Figure 12: Ablation Study in Linux v6.6

than 0.01. Since our tool focuses on testing refcount-related logic, it may ignore new coverage of non-refcount operations and thus, achieve lower coverage. This behavior is consistent with recent kernel fuzzers [14, 19, 34] that focus on particular operations in kernel. For example, SegFuzz [19] aims to detect concurrency bugs by focusing on exploring thread interleavings, and triggers 3.2% less coverage than Syzkaller. The most recent work Actor [14], focuses on heap memory operations, and also reports less coverage. Although these tools cannot trigger the highest coverage, they find new bugs missed by Syzkaller. Similarly, CountDown reports extra UAF and KASAN reports (shown in Figure 9), indicating its unique efficacy on detecting such severe bugs.

Figure 11 shows the comparison among state-of-the-art tools. CountDown with Syzbot corpus triggers significantly more coverage than the other tools, indicating the necessity of using and supporting high-quality corpus. MoonShine (with its high-quality corpus) achieves higher coverage than other settings: CountDown w/o corpus (called CD-NC) and Actor. In detail, without corpus, CD-NC and Actor trigger 5.5% and 21.8% less coverage than MoonShine, separately. For the comparison between our tool with other advanced tools, all the p-values are smaller than 0.01, as shown in Table 2. Considering the coverage brought by MoonShine's high-quality corpus, it is acceptable for CD-NC to trigger less coverage, similar to the case where CountDown achieves much more coverage than other tools with Syzbot corpus. Further, as we discuss in §6.1.2, Actor was developed on top of an old version of Syzkaller, and cannot get benefited from Syzkaller's recent improvement.

6.3 Contribution of Each Component

CountDown takes three steps to mutate a user-space program, specifically, (1) inserting refcount-related syscalls to trigger refcount bugs, (2) appending refcount-decreasing syscalls to reduce refcount to zero, and (3) adding refcount-accessing syscalls to activate invalid uses. To understand the contribution of each step towards triggering bugs, we disable each component one by one to get three new settings, *i.e.*, -refcntmutate that disables refcount-guided mutator, -dupdec that does not insert refcount-decreasing syscalls, and -dupuse that does not insert refcount-accessing syscalls. We test each setting on Linux 6.6 for 72 hours and repeat each setting for 10 times. Further, we find the differences of CountDown, -dupuse and -dupuse, are not significant at the end of 72-hour fuzzing. Therefore, we extend the experiments of these three settings to 92 hours for reliable results. Figure 12 shows the average results.

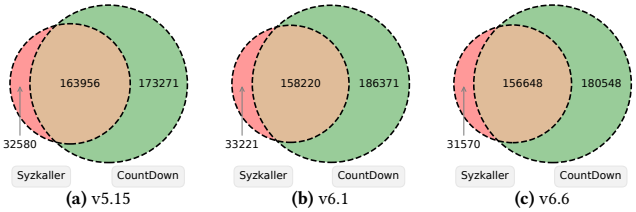
The most important component is our new refcount-guided mutator, where disabling it (*i.e.*, -refcntmutate in the figure) makes our tool only slightly better than Syzkaller. In particular, the UAF-bug detection capability of CountDown decreases by 22.0% without this refcount mutator. For an existing user-space program P that operates a refcount O, the refcount-guided mutator will find another syscall S that ever accesses O in other programs, and insert S to P. Therefore, this mutator can actively generate programs accessing refcounts by different syscall pairs, and thus provides strong contributions to the refcount-guided fuzzing.

For another two components that add refcount-decreasing and refcount-accessing syscalls, the result shows that they help trigger more UAF bugs, especially in the long-run experiment, but trigger similar number of KASAN reports. In the last 24-hour fuzzing, CountDown can still find 0.9 UAF bugs, while -dupdec and -dupuse merely detect 0.5 and 0.7 bugs, respectively. This is expected, these two components are designed for transforming the refcount bugs into UAF bugs by inserting related syscalls. Meanwhile, these syscalls are not strongly related to other bugs such as out-of-bound access, and adding such syscalls could have negative impacts to trigger other bugs. Therefore, the total number of KASAN reports are similar to that of CountDown.

6.4 Refcount-based Relation Learning

We investigate the refcount-based relations constructed by our CountDown and compare them with those built by Syzkaller.

Quantitative Comparison. Figure 13 compares the relations constructed by CountDown and Syzkaller after 72-hour fuzzing. On kernels v5.15, v6.1, and v6.6, CountDown identifies 337196, 344591 and 337227 pairs of related syscalls, including 196756, 200736, and 183158 pairs that ever access shared refcounts. With the same setting, Syzkaller constructs 188218, 191441, and 196536 relations for different kernel versions, more than 80% of which are also received by CountDown. This result shows that CountDown can capture more sensitive progress on kernel state exploration, and thus can detect more complex bugs. On the other hand, Syzlang defines



(a) v5.15 (b) v6.1 (c) v6.6

Figure 13: Relations learned by CountDown and Syzkaller on three kernel versions. CountDown learns more unique relations than Syzkaller and more than 80% relations learned by Syzkaller are learned by CountDown.

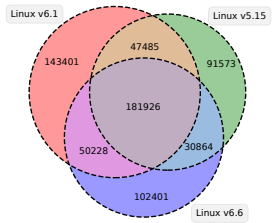


Figure 14: Syscall relations learned by CountDown from different kernel versions.

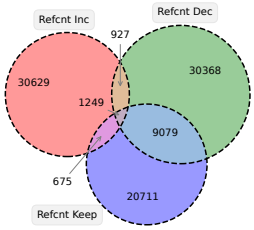


Figure 15: Refcount operations triggered in Linux v6.6 update refcount unidimensionally usually.

more than 4,300 syscalls, which form a huge syscall-relation space consisting of more than 19 million possible pairs. CountDown only identifies less than 2% of all possible combinations, which effectively highlights a small proportion of valuable syscall pairs from the enormous space. Moreover, for three kernel versions, CountDown keeps identifying more previously unseen syscall relations. Therefore, the refcount-based guidance will enhance the bug detection capability for long-term fuzzing settings.

Stability of Learned Relation. Since Linux developers make substantial changes to each kernel version, we are curious about the stability and reusability of syscall relations across different versions. Figure 14 shows the overlaps among syscall relations learned from different kernel versions. After each 72-hour fuzzing, at least 66.1% of the refcount relations learned from one version are also learned independently from another version, and at least 43.0% of the relations are captured by all three different versions. Since CountDown still discovers new syscall relations at the end of our experiment, the percentage of shared relations across different versions could be even higher. These results indicate that refcount-based relation is a stable method for combining different syscalls.

Validity of Refcount-based Relation. We inspect 52 fixed refcount-related UAF bugs to understand the value of our new refcount-based relation. For each bug, we find its reproducer from the bug report. Then, from all 495 pairs of syscalls within the reproducers, we find 109 (22.0%) syscall pairs that do not exist in seed corpus. We focus on such syscall pairs since fuzzers have to generate them to trigger UAF bugs. For each pair of syscalls, we find its relation value from our relation table and the value from Syzkaller's relation table. A stronger relation indicates a higher possibility to produce such syscall combinations. For each pair, we calculate the ratio of refcount-based relation to Syzkaller relation, which indicates the probability improvement for generating this pair. We find CountDown relation is 51.3% (geometric mean) more stronger than Syzkaller relation, resulting in higher chances of generating these crucial pairs to trigger analyzed UAF bugs.

Stability of Refcount Operations. To trigger UAF bugs, we insert refcount-decreasing syscalls to reduce refcount to 0 and add refcount-accessing syscalls to use dangling pointers. Both actions require one syscall to perform stable operations on the same refcount across different executions. To measure the stability of refcount operations, we group these operations based on the δ value. Figure 15 shows that for one particular refcount, most syscalls only engage in either an increment (refcnt-inc operation), decrement (refcnt-dec operation), or no update (refcnt-keep operation). Among 87 thousand unique refcount-syscall pairs, only 2.3% of them both increase and decrease the refcount in different runs. Such unidimensional behavior aligns with our design that repeating specific syscalls in user-space programs can make kernel execution closer to UAF bugs.

6.5 Instrumentation Overhead

CountDown instruments all refcount API functions in Linux kernel to collect refcount operations. Since these functions are invoked frequently within kernel, our instrumentation may bring in heavy performance overhead. To accelerate the kernel fuzzing, we optimize CountDown by selectively executing the instrumented code only when necessary (see §5.1). We use the speed of test case execution during fuzzing to measure the overhead. In detail, the average executing speed overhead is 5.9%, 14.6%, and 14.0% for kernel v5.15, v6.1, and v6.6, respectively. Such overhead is acceptable considering the prevalent refcount usages in kernel.

7 RELATED WORK

We have extensively discussed and compared with the most related kernel fuzzers in previous sections. Next, we will examine other works, focusing on syscall grammar inference, kernel fuzzers for concurrency bugs, and static analyzers for detecting refcount issues.

7.1 Syscall Grammar Inference

Due to extensive checks on syscall arguments, invalid syscalls trigger only shallow error-handling functions, and can hardly reach deep bugs. To address this problem, kernel fuzzers adopt predefined rules of syscall parameters and return values to generate valid syscalls. Syzkaller utilizes Syzlang [39] to define syscall input and output, like data types. For instance, for syscall `socket`, Syzlang defines concrete values `AF_INET` and `AF_INET6` for parameter `family` to indicate ipv4 and ipv6 protocols, and specifies a `sock`-type return value. This value can be used by other syscalls that accept `sock`-type arguments, like `bind` and `close`. However, Syzlang is implemented manually based on expert domain knowledge, and requires a lot of human efforts to maintain it. Several works [10, 12, 16, 38] try to generate Syzlang descriptions automatically. Difuze [12] and SyzDescribe [16] perform static analysis on kernel code to recover syscall interfaces. KSG dynamically searches the device list and analyzes the related kernel functions to infer calling conventions. Syzgen++ [10] builds dependencies by checking insertion

and lookup operations on the same kernel data container. Recent work FuzzNG [9] proposes a grammar-free solution that generates required data on demand, especially for file contents and pointers.

7.2 Kernel Fuzzers for Concurrency Bugs

Kernel concurrency fuzzers focus on exploring different execution orders across multiple threads and have uncovered many concurrency bugs, including a few UAF bugs. RAZZER [18] combines static analysis and two-stage fuzzing to find concurrency bugs. It first locates instructions that may concurrently access the same memory location, and then adopts single-thread fuzzing to generate programs reaching race points. At the end, it uses multi-threads fuzzing to find thread interleavings that trigger concurrency bugs. DDRace [48] adopts directed grey-box fuzzing to trigger kernel concurrency UAF bugs. It first utilizes lightweight dynamic analysis to identify instructions that use or free the same object. Then, it uses static analysis to recognize race pairs that may simultaneously release and use the object. To reach each potential race pair, DDRace leverages UAF-bug constraints and unique pairs of read/write instructions to guide the test case generation. These works mainly focus on detecting concurrency bugs, and UAF bugs are merely side effects. CountDown aims to identify UAF bugs caused by refcount issues. Therefore, we design a new mechanism that utilizes refcount operations to calculate syscall relations and prioritize test cases.

7.3 Static Analyzers to Detect Refcount Issues

Researchers propose static analysis techniques to detect refcount issues from large systems such as Linux kernel. Pungi [24] assumes that refcount changes and reference operations should match along a complete path and treats all violations as refcount bugs. RID [29] requires that different code paths within the same function have the same refcount operation. CID [40] advances the static analysis in two ways, checking refcount consistency across callers and callees, and considering functions with deviated refcount operations as suspicious. Recent work LinKRID [27] requires refcount changes and reference operations to be consistent within a function. Although each static analyzer helps identify a set of refcount issues (some leading to UAF bugs), these techniques suffer from two limitations. First, kernel code commonly splits refcount changes and reference operations into different functions. To detect refcount issues, we need to conduct heavy inter-procedural data-flow analysis, which significantly limits the scalability. Many analyzers only focus on detecting bugs from a subsystem. Second, without concrete execution context, static analysis has to over-estimate kernel states, leading to many false positives. For example, LinKRID produces around 40% false positives, which brings a heavy burden to kernel developers to distinguish true bugs from false alarms.

CountDown avoids these limitations since it uses concrete executions to detect true-positive bugs at runtime. Although these two techniques are orthogonal to each other, it is possible to combine them strategically. For example, we can use static analyzers to identify suspicious buggy locations, and adopt the refcount-based relation from CountDown to guide a directed fuzzing to confirm bugs. We plan to explore this direction in future work.

8 CONCLUSION

In this paper, we propose CountDown, a novel kernel fuzzer for finding refcount-related use-after-free bugs. CountDown reshapes syscall relations through shared refcounts among different syscalls, generates user-space programs based on refcount-based relations, and injects refcount-decreasing and refcount-accessing syscalls to free objects and use dangling pointers. We test CountDown on three Linux kernels, and it produces 66.1% more UAF bugs and 32.9% more KASAN reports than previous tools. CountDown has found nine new kernel bugs with two confirmed.

ACKNOWLEDGMENT

We thank the anonymous reviewers, especially our shepherd, for their insightful comments and valuable feedback. This research was supported by National Science Foundation (NSF) under grants CNS-2247652 and CNS-2339848. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

REFERENCES

[1] AppArmor: Linux Kernel Security Module. https://apparmor.net/.
[2] Linux Kernel DCCP Use-after-free Privilege Escalation. https://support.alertlogic.com/hc/en-us/articles/115003048363-Linux-Kernel-DCCP-Use-after-free-Privilege-Escalation, 2017.
[3] net/nfc: Fix use-after-free llcp_sock_bind/connect. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/?id=c61760e6940d, May 2021.
[4] Reproducing Evaluation Part of Paper Healer. https://github.com/SunHao-0/healer/issues/37, Oct. 2021.
[5] Linux Kernel Use-after-free in Netfilter nf_tables When Processing Batch Requests can be Abused to Perform Arbitrary Reads and Writes in Kernel Memory. https://seclists.org/oss-sec/2023/q2/133, May 2023.
[6] The Kernel Memory Sanitizer (KMSAN). https://docs.kernel.org/dev-tools/kmsan.html, 2023.
[7] Researchers Uncover New Linux Kernel 'StackRot' Privilege Escalation Vulnerability. https://thehackernews.com/2023/07/researchers-uncover-new-linux-kernel.html, July 2023.
[8] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 1032–1043, Vienna, Austria, Oct. 2016.
[9] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. In *Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2023.
[10] W. Chen, Y. Hao, Z. Zhang, X. Zou, D. Kirat, S. Mishra, D. Schales, J. Jang, and Z. Qian. Syzgen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2024.
[11] G. E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960. ISSN 0001-0782.
[12] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 2123–2138, Dallas, TX, Oct.–Nov. 2017.
[13] M. Fleischer. Actor, 2023. https://github.com/ucsb-seclab/actor.
[14] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna. ACTOR: Action-Guided Kernel Fuzzing. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*, pages 5003–5020, Anaheim, CA, USA, Aug. 2023.
[15] Google. Honggfuzz. https://google.github.io/honggfuzz/.
[16] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 3262–3278, San Francisco, CA, May 2023.
[17] L. He, H. Hu, P. Su, Y. Cai, and Z. Liang. FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, pages 2497–2512, Boston, MA, USA, Aug. 2022.
[18] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding Kernel Race Bugs Through Fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 754–768, San Francisco, CA, May 2019.

[19] D. R. Jeong, B. Lee, I. Shin, and Y. Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 2104–2121, San Francisco, CA, May 2023.

[20] Kernal Source Code. Kernel Refcount API. https://github.com/torvalds/linux/blob/master/include/linux/refcount.h. (visited in May 2023).

[21] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[22] A. Konovalov and D. Vyukov. Kernel Address Sanitizer (KASAN): A Fast Memory Error Detector for the Linux Kernel. *LinuxCon North America*, 2015.

[23] G. Kroah-Hartman. kobjects and krefs. In *Linux Symposium*, page 295, 2004.

[24] S. Li and G. Tan. Finding Reference-counting Errors in Python/C Programs with Affine Analysis. In *Proceedings of the 28th Annual European Conference on Object-Oriented Programming (ECOOP)*, pages 80–104, Uppsala, Sweden, July–Aug. 2014.

[25] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 2078–2095, San Francisco, CA, May 2022.

[26] Z. Lin, Y. Wu, and X. Xing. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, pages 1963–1976, Los Angeles, CA, USA, Nov. 2022.

[27] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, pages 125–142, Boston, MA, USA, Aug. 2022.

[28] LLVM. LibFuzzer - A Library for Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html.

[29] J. Mao, Y. Chen, Q. Xiao, and Y. Shi. RID: Finding Reference Count Bugs with Inconsistent Path Pair Checking. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[30] Microsoft. Rules for Managing Reference Counts. https://docs.microsoft.com/en/windows/desktop/com/rules-for-managing-reference-counts, Aug. 2020.

[31] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, Dec. 1990.

[32] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, Aug. 2018.

[33] PaX Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[34] H. Peng and M. Payer. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 2559–2575, Virtually, Aug. 2020.

[35] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 137–137, San Francisco, CA, May 2024.

[36] Z. Shen, R. Roongta, and B. Dolan-Gavitt. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, pages 1275–1290, Boston, MA, USA, Aug. 2022.

[37] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui. Healer: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 344–358, 2021.

[38] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *Proceedings of 2022 USENIX Annual Technical Conference (USENIX ATC)*, pages 351–366, 2022.

[39] Syzkaller developers. Syscall Description Language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[40] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, pages 2471–2488, Virtually, Aug. 2021.

[41] D. Vyukov. Syzbot Corpus. https://github.com/google/syzkaller/blob/3af7dd651dc78ce0784bef793d14dd2f72d07138/tools/demo_setup.sh#L38, 2023. corpus.

[42] D. Vyukov and A. Konovalov. Syzbot and the Tale of Thousand Kernel Bugs, 2018. Linux Security Summit.

[43] D. Vyukov and A. Konovalov. Syzkaller: An Unsupervised, Coverage-guided Kernel Fuzzer. https://github.com/google/syzkaller, 2019.

[44] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics In Greybox Fuzzing. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Sept. 2019.

[45] 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)*, pages 781–797, Baltimore, MD, Aug. 2018.

[46] Y. Wu, Z. Lin, Y. Chen, D. K. Le, D. Mu, and X. Xing. Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*, pages 4247–4264, Anaheim, CA, USA, Aug. 2023.

[47] 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 Conference on Computer and Communications Security (CCS)*, pages 414–425, Denver, Colorado, Oct. 2015.

[48] M. Yuan, B. Zhao, P. Li, J. Liang, X. Han, X. Luo, and C. Zhang. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*, pages 2849–2866, Anaheim, CA, USA, Aug. 2023.

[49] M. Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl.

[50] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, pages 71–88, Boston, MA, USA, Aug. 2022.

[51] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian. SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, pages 3201–3217, Boston, MA, USA, Aug. 2022.