One Simple API Can Cause Hundreds of Bugs
An Analysis of Refcounting Bugs in All Modern Linux Kernels

Liang He[1] Purui Su[1,2,3]✉ Chao Zhang[5,6] Yan Cai[2,4] Jinxin Ma[7]
[1]TCA / [2]SKLCS, Institute of Software, Chinese Academy of Sciences
[3]School of Cyber Security, University of Chinese Academy of Sciences
[4]School of Computer Science and Technology, University of Chinese Academy of Sciences
[5]Tsinghua University [6]Zhongguancun Laboratory [7]CNITSEC

Abstract

Reference counting (refcounting) is widely used in Linux kernel. However, it requires manual operations on the related APIs. In practice, missing or improperly invoking these APIs has introduced too many bugs, known as refcounting bugs. To evaluate the severity of these bugs in history and in future, this paper presents a comprehensive study on them.

In detail, we study 1,033 refcounting bugs in Linux kernels and present a set of characters and find that most of the bugs can finally cause severe security impacts. Besides, we analyze the root causes at implementation and developer's sides (i.e., human factors), which shows that the careless usages of *find*-like refcounting-embedded APIs can usually introduce hundreds of bugs. Finally, we propose a set of anti-patterns to summarize and to expose them. On the latest kernel releases, we totally found 351 new bugs and 240 of them have been confirmed. We believe this study can motivate more proactive researches on refcounting problems and improve the quality of Linux kernel.

CCS Concepts: • **Security and privacy → Operating systems security**.

Keywords: reference counting, bug, Linux kernel

1 Introduction

As a simple but efficient programming technique to manage critical resources, reference counting (*refcounting* hereafter) [24, 25] is heavily used in modern programs. For example, there are 93.5% files involved refcounting in the Linux kernel [39]. Unfortunately, refcounting requires manually invoke its APIs. When across modules in large-scale programs, it becomes too complex to be free of bugs due to missing or improper invocation. From our statistics, *Linux kernels are suffering from more and more severe refcounting bugs.*

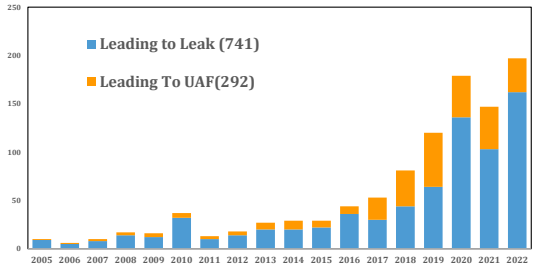

Figure 1. The growth trend of refcounting bugs in Linux kernels from 2005 to 2022.

Given that plenty of pattern-based or semantic-based detection methods or tools have been proposed to defeat refcounting bugs [20, 23, 26, 27, 35, 39, 43, 44, 48], it raises the questions about *why there are still increasingly refcounting bugs reported, what are the characteristics or root causes of these bugs* and *what can be done to proactively reduce or eliminate the refcounting bugs.*

To answer these questions, we tried to collect and analyze the refcounting bugs reported in the historical releases of the Linux kernel [13]. Totally, as shown in Figure 1, we have found 1,033 refcounting bugs in 753 versions of Linux kernels from 2005 to 2022. For each one of them, we have tried to carefully ascertain the bug details from the commit log, patch code, function-level context and even the discussions between the patch authors and developers [36].

With the bugs and dissections, our first goal is to understand the real security impacts of the bugs, which can be used to explain why more and more new refcounting bugs are detected in recent security researches [27, 35, 39, 48]. Then, we aim to explore the distribution and lifetime characteristics by conducting a throughout statistic analysis, which can help future researches to put in more effort into error-prone subsystems and long-lived kinds of bugs. Finally, we hope to correctly infer the root causes that can not only explain the reason why there are so many refcounting bugs in Linux kernel, but also help us to propose meaningful anti-patterns, which can motivate us to detect new bugs in latest releases.

Security Impacts. It is our first two findings that almost each of the refcounting bugs can lead to security bugs. Specifically, 71.7% of the bugs have caused memory leaks [9] and other ones finally introduced use-after-free (*UAF*, hereafter) bugs [10], both of which can easily make a running kernel

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613162

Listing 1. A Missing-Refcounting Bug.

```
1 //drivers/nvmem/core.c
2 struct nvmem_device *__nvmem_device_get(...)
3 {
4   dev = bus_find_device(...);
5   ...
6   if(any_error)
7     return error_code;
8 }
```

Listing 2. A Misplacing-Refcounting Bug.

```
1 //drivers/usb/serial/console.c
2 static int usb_console_setup(...)
3 {
4   ...
5   usb_serial_put(serial);
6   mutex_unlock(&serial->disc_mutex);
7   ...
8 }
```

crash. Based on the security impacts and related patch code, we gave a more reasonable classification on these bugs.

Distributions and Lifetimes. First, we found the refcounting bugs existed in every corner of the whole kernel. Besides, we found that all the bugs followed a long-tailed distribution: more than 80.0% bugs existed in only three subsystems. Finally, we found that 70.0% of bugs needed more than one year to be fixed after they were first introduced, 19 bugs existed more than 10 years and 23 bugs existed from the first Git release (v2.6) to recent ones (v5.x and v6.x).

Unexplored Root Causes and Anti-Patterns. Different with all of existing protection or detection methods [27, 35, 39, 44, 48], we tried to explore the root causes of the refcounting problems. As one of our main contribution, we found that if a refcounting API is implemented with a subtle deviation, i.e., the behaviors deviated from most implementations, it can introduce hundreds of bugs. Besides we also inferred other critical problems that have also introduced hundreds of refcounting bugs. Finally, we proposed 9 anti-patterns and detected 351 new bugs by matching them in latest releases.

Specifically, for refcounting bugs leading to memory leaks, there are two potential root causes. First, when the developers introduced any subtle deviation to the standard refcounting algorithm, it would cause many bugs. For example, we found that a special API, *pm_runtime_get_sync()*, has caused 94 bugs. The main reason is that this API will increase the reference counter (*refcounter* hereafter) even when it meets some error and returns an error-code, in which case the callers will often miss the decreasing operation. Second, when the developers, for code conciseness, hide refcounting-increasing operations into some macros or *find*-like APIs, the new or even skilled kernel developers will often miss the paired refcounting-decreasing operations, which will eventually lead to refcounting bugs. This problem has also caused a large number of bugs.

For the bugs leading to UAF bugs, we also found two critical reasons. First, when the developers accessed the objects after they invoked the decrement refcounting API, referred to use-after-decrease (*UAD*, hereafter) problem in this paper, there would be a potential UAF bug. Besides, the reference escape analysis [51] encourages developers to optimizing refcounting, which may wrongly omit updating refcounters. This is the second reason that has caused many potential UAF bugs. Each of above two problems has also caused hundreds of refcounting bugs.

Based on above root causes, we proposed 9 anti-patterns described by the semantic templates [19, 37]. Then, we designed and implemented a static checker for each of the anti-patterns. The checker totally detected 351 new bugs in several latest kernel releases and 240 ones have been confirmed when writing this paper. By deeply analyzing the new bugs, we found that their characteristics were very similar with the historical ones. Based on this fact, we believe *the refcounting bugs can be greatly diminished if the developers or researchers begin to put in more effort into the characteristics and lessons learned from this study.*

The main contributions of this work are three-fold: (1) conducting the first thorough study of refcounting bugs in modern Linux kernels; (2) exploring latent problems leading to so many refcounting bugs, which can proactively defeat the bugs in the future development; (3) the anti-patterns extracted from those bugs, which help to detect hundreds of new ones.

2 Background

2.1 Refcounting

Basically, to implement refcounting, programmers prefer to use an integer, i.e., refcounter, to record the number of references to a memory block. If there is a reference creation or destruction, the refcounter should be incremented or decremented. If there is no any reference, i.e., the refcounter becomes zero, the memory block will be deallocated. However, if there is any refcounting missed or not operated properly, i.e., refcounting bug [11], it will lead to severe security impacts, e.g., memory leaks and UAF bugs.

2.2 Refcounting Bugs

In this paper, we mainly consider following two main kinds of refcounting bugs, which can easily cause the memory leak and UAF bugs. While there are indeed other kinds of refcounting bugs, e.g., the ones caused by race problem [8], we will leave them as our future works.

Missing-Refcounting Bugs are bugs that programmers miss the decrement or increment that should have been done for a reference creation or destruction. Specifically, missing decrement will cause a memory leak as the refcounter will

never be zero and missing increment will cause a UAF bug as the refcounter will prematurely become zero. Listing 1 presents a missing-decrement bug in the NVMEM driver [30]. Specifically, the developers only focus on the returned *dev* from the *bus_find_device* function (Line 4) and they do not realize the existence of the embedded increment refcounting in the *find*-like API. When there is some error, the code will be terminated without any paired decrement operation (Line 7), which will inevitably lead to a memory leak.

Misplacing-Refcounting Bugs are bugs that programmers do not place the increment or decrement APIs into proper places. Listing 2 shows a real-world misplacing-refcounting bug in the USB serial drivers [17]. Specifically, the developers only consider to protect related operations on the target object in a *locked* code block to avoid race problem. However, they do not realize the potential UAF bug as the *usb_seriral_put*, a decreasing API, will release all the resources attached into the *serial* and then free itself if its refcounter is one when calling the API.

2.3 Our Goals

Considering the severity, there have been several efficient protection-/detection-strategies proposed to defeat the refcounting bugs (see §8). However, different with existing works, the main goals of this paper are to *answer* the reasons of increasing numbers of the refcounting bugs and to be an *instructive study* for future proactive solutions.

Specifically, our first goal is to collect all (most) of the historical refcounting bugs fixed in Linux kernels and to build a considerable bug dataset. Then, based on it, we try to do a statistical research on the security impacts, classifications, distribution and lifetime characteristics. Finally, we want to infer the latent reasons, i.e., root causes, for these bugs, which can be useful for future kernel developments. While we also try to apply our study results to design and implement static checkers to detect new refcounting bugs, we only aim to prove the reasonableness of our findings and root causes. *Pursuing a sound and complete detection solution is not the goal of this work.*

3 Methodology

3.1 Refcounting Bug Dataset

To disclose the characteristics and latent reasons, as our first primary goal, we conducted a thorough study that covers about 753 versions of Linux kernels released from 2005 to 2022. Specifically, we use a two-level filtering method to identify historical refcounting bugs. First of all, by extracting key words in the refcounting API names, e.g., "get", "take", "hold", "grab" for increasing APIs and "put", "drop", "unhold", "release" for decreasing APIs, we found out the committed patches that *add*/*delete*/*move* the APIs whose name strings contains above words. Then, we also conducted a deep analysis by checking the implementation of the related APIs, as the second filtering stage, which can help us to only select the refcounting bug related patches.

After above filtering, we totally collected 1,825 candidates refcounting bugs from more than one million of committed logs. By manually confirming each of them, e.g., trying to understand the real meaning of the commit messages, checking the contexts of the patch code and even the discussions between the patch authors and the developers, we finally extracted 1,033 bugs as our studied dataset.

Threads to Validity. Like all characteristic's studies, on the one hand, we may miss other kinds of refcounting bugs when we use above methods. For example, in Listing 2, if developers chose to move the *mutex_unlock* after *usb_serial_put*, where no change of the refcounting API will be found, then we will miss it. Besides, the patch authors can also not use any above key word in their commit message, which will be a challenge for our current collecting strategies. To reduce the potential false negatives and collect the missed historical bugs is left as one of our future works.

On the other hand, we may also collect wrong patch commits as false positives. For example, the commit-dcb4b8ad [7] has been proved wrong by the commit-0a96fa64 [1]. Specifically, the former patch, adding a "missing" decrement operation, indeed introduced an extra decrement, which will cause a potential premature free, i.e., UAF bug. To filter out above false positives, we turned to the "Fixes" tags [16], shown in following gray box, in commit logs and removed the selected bugs whose commit-IDs are the tag values of other commits.

> Fixes: dcb4b8ad6a44 ("misc/uss720: fix memory ...")

3.2 Semantic Template based Bug Description

Considering refcounting bugs as a kind of specific semantic bugs [34, 47, 50] and inspired by a recent work [37], we adopt a semantic template-driven method to describe the studied refcounting bugs. In this paper, we mainly consider following semantic elements to build a reasonable template.

Semantic Operators. First of all, we use two symbols to refer to refcounting operators: $\mathcal{G}$ to increment refcounting and $\mathcal{P}$ to decrement refcounting. Then, we will also use $\mathcal{A}$, $\mathcal{D}$, $\mathcal{L}$, $\mathcal{U}$ to refer to common *assignment*, *dereference*, *lock* and *unlock* operations. Finally, we can use the operators as functions with one or more parameters p_i, which are actually the object pointers. For example, we use $\mathcal{G}(p_0)$ to mean the increasing operation on the pointer p_0. It is noted that two or more operators can be nested, with $\circ$, to represent the complicate behaviors. For example, we can use the $\mathcal{U} \circ \mathcal{D}$ to mean the *unlock* operation nested with a pointer *dereference*.

Contexts. Firstly, We use following symbols to refer to different contexts: $\mathcal{S}$ to a statement, $\mathcal{B}$ to a basic block, $\mathcal{F}$ to a function and $\mathcal{M}$ to a macro. Secondly, we can combine each of above symbols with an operator, as a subscript, to refer to

Bug	Semantic Tamplates
Listing 1	$\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{G}} \rightarrow \mathcal{B}_{error} \rightarrow \mathcal{F}_{end}$
Listing 2	$\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{P}(p_0)} \rightarrow \mathcal{S}_{\mathcal{U} \circ \mathcal{D}(p_0)} \rightarrow \mathcal{F}_{end}$

Table 1. Semantic templates for the two listed bugs. We use $\rightarrow$ to mean a potential execution path.

a specific context containing the proper operation. For example, we will use $\mathcal{S}_{\mathcal{G}}$, $\mathcal{S}_{\mathcal{P}}$ to refer to a pair of statements that complete the increment and decrement refcounting. Finally, we can add any meaningful semantic names as subscripts into the context symbols, which can represent specific context. For example, we will use the $\mathcal{F}_{start}$ and $\mathcal{F}_{end}$ to refer to the entry and exist of a function, and use $\mathcal{B}_{error}$ to mean an error-handling code block.

Bug Description. As shown in Table 1, we use above symbols to generate two semantic templates to describe the previous bugs presented in §2.2. Specifically, the first template can tell us that there is a potential execution path in which the developers only call the increment refcounting and then jump into a error-handling code without any paired decrement operation. From the second template, we can get that there is a potential execution path in which the developers call a decrement refcounting, with the pointer p_0 and then directly dereference the same pointer within a nested *unlock* statement. Based on the above two templates, we cannot only easily infer the refcounting bugs but also be motivated to design and implement static checkers to detect new similar bugs (see the details in §6).

4 General Findings

In this section, we describe the general findings from the analysis of selected refcounting bugs, including *impacts*, *classification*, *distributions* and *lifetimes*.

4.1 Security Impacts

To understand the real-world security impacts, we begin to search in patch description the key words that can reveal the potential impacts, e.g., "leak", "use-after-free", "uaf", "crash", "out of memory". Finally, we conclude following two findings.

Finding 1. *A majority (741/1033, about 71.7%) of the studied refcounting bugs can lead to memory leaks, and more than two-thirds (694/1033, about 67.2%) of all bugs are caused by missing-decreasing problems. More than one half (590/1033, about 57.1%) of the bugs can be detected by searching unpaired operations within the same functions.*

Based on whether added decreasing APIs in the same functions with the increasing ones, as shown in the top half of the Table 2, we divided the missing-decreasing bugs into *Intra-Unpaired* and *Inter-Unpaired*. For other special bugs,

Impact	Refcounting Bug		%
Leak (**71.7%**)	1. Missing-Decreasing		**67.2**%
	1.1 Intra-Unpaired	57.1%	
	1.2 Inter-Unpaired	10.1%	
	2. Others		**4.5%**
UAF (**28.3%**)	3. Misplacing-Refcounting		**13.9**%
	3.1 Decreasing (UAD)	11.5% (9.1%)	
	3.2 Increasing	2.4%	
	4. Missing-Increasing		**7.2%**
	5.1 Intra-Unpaired	5.1%	
	5.2 Inter-Unpaired	2.1%	
	5. Others		**7.2%**

Table 2. The percentage of different kinds of refcounting bugs. The underlines of the percentages mean we will infer the root causes for them.

e.g., operating on a wrong API, we categorized them into *Others* and do not consider them in following analysis. While memory leaks usually involve a small structures, but the attackers can easily trigger the leaks many times through some *loop*-based scripts [4], which will eventually lead to serious security impacts.

Note that most of the misplacing-refcounting bugs are actually the missing-refcounting ones. For example, we firstly identified the bug commit-bf4a9b24 [6] as a misplacing-refcounting bugs as there is an explicit movement. However, after manually confirming with the patch description, we realized that this is actually an intra-unpaired bug as the premature *exist* leads to the increasing operation of unpaired.

Finding 2. *About 28.3% (292/1033) of the studied bugs can lead to UAF bugs and most of them are caused by misplacing-refcounting. Specifically, about 9.1% (94/1033) of all bugs can be detected by checking if there is any reference access after the decreasing operations.*

First of all, it is a real surprise finding that the misplacing-refcounting is the major kind of refcounting bugs leading to UAF bugs as recent UAF-related researches [27] only think about the missing-refcounting bugs. Furthermore, among these bugs, we find a simple but frequently reported bug type, i.e., UAD, and we will describe it more detailed in §5.4.1.

Then, we conclude the following two reasons why there are fewer UAF-causing bugs reported. On the one hand, based on our patch-committing experience, the kernel developers are very strict to accept any UAF bug patch as these patches will be applied into the stable releases. Usually, they need the committer to provide a real crash report or even the Proof-of-Concept (*PoC* hereafter). On the other hand, it is difficult to successfully produce a UAF-triggered PoC even after finding a real refcounting bug, which usually need a long-time fuzzing work [46].

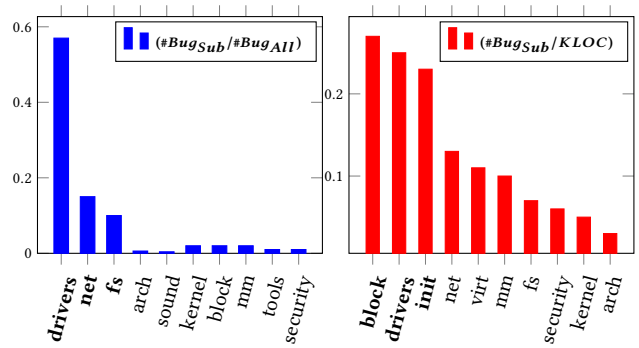

Figure 2. Distributions of refcounting bugs. The left subfigure presents the refcounting bug numbers of different subsystems and the right one presents the bug density.

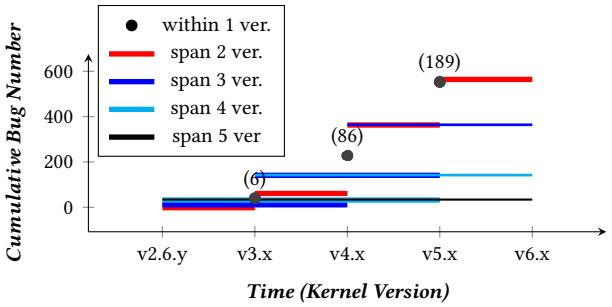

Figure 3. The lifetime of refcounting bugs. This figure is inspired by the work [22]. We use the bug-introduced time (version) and the bug-fixed time (version) to draw the lines.

4.2 Bug Distributions

To explore the location distributions of the bugs and make it clear what kind of source code can be error-prone, we begin to analyze the source files where each of the bugs are detected. Overall, our finding is depressed as the refcounting bugs can happen almost in every corner, even in the kernel *init* code [12]. By counting the numbers of bugs existed in each subsystems, we presented two kinds of results in the bar-charts of Figure 2.

Finding 3. *The refcounting bugs meet the long-tailed distributions in Linux kernel. About 82.4% (851/1033) of refcounting bugs could be detected within "drivers", "net" and "fs" subsystems, among which more than a half (588/1033, about 56.9%) of all bugs occurred in "drivers".*

In fact, the long-tailed distribution is not specific to refcounting bugs. An early work [22] has already reported this kind of distributions for many types of operating systems errors. Here we use another metric, *bug density*, i.e, number of bugs per thousand lines of code (KLOC), which is presented in the right half of the Figure 2. Different with the early work [22], it is the "block" subsystem, not the "driver", has the highest density. While there are only 18 bugs existed in "block", it only has 65 KLOC. We hope our above findings can motivate the researchers to select proper targets to check [35, 38, 39, 41, 48] and fuzz [21, 31, 32, 45, 46].

4.3 Bug Lifetimes

To expose the latent periods of our studied refcounting bugs, which can help us to realize the difficulty to detect them, we analyzed the lifetime of each bugs. Specifically, we again turned to the "Fixes" tags by which we can calculate the lifetimes for the bugs, i.e., from the time of the bug first introduced to the time of the bug fixed by the current commit. However, as not all of kernel developers require the "Fixes" tags, we found only 567 bugs that own that tags.

Finding 4. *It is surprising that about 75.7% (429/567) of refcounting bugs that need more than one year to be fixed after they are first introduced in the kernels. There are even 19 bugs that existed more than 10 years, including 7 bugs that can finally lead to UAF bugs.*

The main reason for the long lifetimes is that refcounting bugs are actually a kind of silent semantic bugs [37] which are difficult to be detected by the mainstream fuzzing-based methods. Besides, from this finding, we should realize that there may be still lots of bugs in the un-patched running kernels. To prove this point, we make a further analysis to figure out how many kernels a refcounting bugs can infect based on their lifetimes. In Figure 3, we present the details of our analysis. It is noted that the bugs are sorted by the time (version) they are introduced.

Finding 5. *There are 23 bugs that existed from the first major release (v2.6.y) to the recent ones (v5.x and v6.x), which means these bugs can infect most of the Internet machines that installed Linux kernels .*

For example, the refcounting bug commit-0711f0d7 during boot was introduced from "the pre-KASAN stone age of v2.6.19" [12], which can make most of the modern Linux machines crash. Besides, in Figure 3, there are also many bugs whose lifetimes have spanned across two or more major releases. For example, there are about 135 bugs spanning from release v4.x to v5.x and 80 bugs from release v3.x to v5.x. Finally, we can also see that there are more and more bugs that are introduced in newer versions but also more and more ones are fixed up in the same major releases, e.g., 189 bugs were introduced and fixed within the v5.x kernels.

5 Root Causes

We conducted our root cause inference only for the representative bugs. Specifically, for leak-related bugs, we chose all *missing-decreasing* bugs (67.2%). For UAF-related bugs, we chose the *UAD* bugs (9.1%) and *intra-unpaired* bugs (5.1%). These types have been introduced in §4.1. We leave other complicated and valuable bugs, e.g., race problems, as our future works.

First of all, we carefully conduct a classification for the APIs that involve refcounting operations. Specifically, based on our manual analysis of the implementations of the APIs that have caused bugs, we classify all of them into following three categories.

General Refcounting APIs. These APIs are used to directly increase or decrease the refcounter of the objects of basic structures, e.g., *refcount_t*, *kref* and *kobject*. Correspondingly, *refcount_inc* / *refcount_dec*, *kref_get* / *kref_put*, and *kobject_get* / *kobject_put* are three common pairs of general refcounting APIs. These general APIs are used widely in the whole kernel code space.

Specific Refcounting APIs. These APIs are used to operate specific objects, e.g., *device_node*, which will use the basic object *kobject* as one of the field members. The developers usually implement these specific APIs by directly invoking the general ones, and pass the specific object as their parameters. These wrapped APIs are usually used in specific subsystem. For example, the *of_node_get* / *of_node_put* for the *device_node* are only used to support the DeviceTree related code.

Refcounting-Embedded APIs. These APIs are mainly use to complete non-refcounting tasks, but embedding the refcounting operations. For example, in Listing 1, the developers provide the *bus_find_device* API mainly to find a proper device based on the specific bus parameter. During the implementation, they want to keep the aliveness of the returned objects and then chose to embed a specific increasing APIs, e.g„ *get_device* into their algorithms. Note that this kind of APIs, especially the *find*-like APIs, have caused hundreds of missing-refcounting bugs.

Now we will introduce several the reasonable root causes inferred from three dimensions: (1) the differences of refcounting APIs' implementations. (2) the locations of the unpaired and missed APIs. (3) the potential risks by calling or not calling these APIs. Here, to be beneficial for the Linux community or other researchers, we have listed the APIs error-prone to refcounting bugs in the Appendix A.

5.1 Implementation Deviation

Overall, a subtle deviation of the refcounting implementations, compared with most common ones, can cause lots of unexpected bugs as the callers may not easily notice it. In this paper, we will introduce two kinds of deviated refcounter-increasing APIs that totally have caused hundreds of bugs.

5.1.1 Return-Error Problem.

While there is no need to return any error code for a regular refcounting API, there is always exception. For example, the commit-87710394 [5] reports a refcounting bug involves an internal power management API, i.e., *__pm_runtime_suspend*, which is used in a helper API, *pm_runtime_get_sync* [28] (we also found another similar API *kobject_init_and_add* [33]). In the implementations of this API, the developer will increase the refcounter no matter if there is any error occurred. In other words, once the refcounting API is invoked, the caller must invoke the corresponding *kobject_put* to avoid potential memory leaks in any potential code path.

Listing 3. An Intra-Missing Bug Caused By Return-Error.

```
//drivers/base/power/runtime.c
static int __pm_runtime_suspend(...)
{
  int retval;
  if(...) atomic_inc(...); //increasing
  retval = rpm_resume();
  return retval;
}
//include/linux/pm_runtime.h
static inline int pm_runtime_get_sync()
{
  return __pm_runtime_suspend();
}
//drivers/crypto/stm32/stm32-crc32.c
static int stm32_crc_remove(...)
{
  int ret = pm_runtime_get_sync();
  if(ret<0)
    return ret; // *BUG!* decreasing is missed!
}
```

For clarity, in top part of Listing 3, we present the implementation details of the *__pm_runtime_suspend* function. Note that this API adds an extra invocation to *rpm_resume* which can be treated as a deviation with other cases. Based on this implementation, we can see that the API will always increase the refcounter and the callers should add the decreasing API in any potential code path. Unfortunately, **a common behavior of the usage of error-returned API is that when there is an error the caller will directly jump into the error path without considering to pair refcounting operations**. Specifically, we present in the bottom of Listing 3 the buggy behavior reported in commit-87710394. Totally, we found in our dataset there were totally 106 bugs caused by above two APIs.

5.1.2 Return-NULL Problem.

In a stand implementation, the increasing API will only accept an object pointer and increase its refcounter without returning anything. However, in many cases, the developers prefer to use the same object pointer as the returned value, which can be used for the caller to invoke the corresponding decreasing APIs. As a result, there will be a potential null-pointer-dereference (*NPD* hereafter) bugs as the returned pointer, which may be NULL, will be directly accessed or dereferenced without any NULL-check. We find 7 new bugs caused by the return-NULL problem in latest release and 3 of them have been confirmed by the developers.

Listing 4. A SmartLoop and A Bug Caused by Loop Break.

```
1  //include/linux/of.h
2  #define for_each_matching_node(...) \
3     for (dn = of_find_matching_node(NULL); \
4           dn; dn = of_find_matching_node(dn))
5  //drivers/of/base.c
6  struct device_node *of_find_matching_node(from)
7  {
8    for_each_of_allnode_from(from, np)
9        if(match && of_node_get(np))
10       ...
11   of_node_put(from);
12   return np;
13 }
14 //drivers/soc/bcm/brcmstb/pm/pm-arm.c
15 static int brcmstb_pm_probe(...)
16 {
17   for_each_matching_node(...) { // start - INC
18     if (condition)              // end  - DEC
19         break;   // /*BUG!*  missing the DEC
20   }
21 }
```

5.1.3 Lessons and Anti-Patterns.

While subtle deviations of refcounting APIs are accepted in kernel development considering the various requirement of the implemented functions, the callers should be very careful when they use such kinds of APIs. One way to defeat the potential refcounting bugs is to provide the detailed explanation as the API comments, which have been adopted by current releases. Another way is to proactively detect such deviations, as an important future work, and then make them public known for new or even skilled kernel developers.

For the bugs caused by implementation deviations, we propose following anti-patterns:

Anti-Pattern 1: $\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{G}_E} \rightarrow \mathcal{B}_{error} \rightarrow \mathcal{F}_{end}$.

Anti-Pattern 2: $\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{G}_N} \rightarrow \mathcal{S}_{\mathcal{D}_N} \rightarrow \mathcal{F}_{end}$.

Here, we use $\mathcal{G}_E$ to indicate the APIs that increase refcounters no matter if there is any error, use $\mathcal{G}_N$ to indicate the APIs may return NULL pointer, and use $\mathcal{D}_N$ to mean a pointer dereference without any NULL-check.

5.2 Hidden Refcounting

It is obvious that if a refcounting API is hidden, the caller will miss the paired operation with a high probability. In this paper, we use the word *hidden* to mean two things: (1) the invocation locations of the refcounting APIs in the bug-caused APIs are hard to be noticed. (2) the semantic similarity between the key words of refcounting API names and bugs-caused API names are very low. Now we will introduce three kinds of hidden refcounting problems that have frequently lead to missing-refcounting bugs.

5.2.1 Complete-Hidden Problems.

To make it clear, we use a real-world bug reported by the commit-1085f508 [2], shown in the bottom of Listing 4 and detected in a SOC driver function. During each iteration of the macro-defined *for_each_matching_node*, referred to *SmartLoop* in this paper, an refcouting-embedded API, i.e., *of_find_matching_node* will be automatically invoked in the end of each iteration. It is worth noting that this specific API will accept an object pointer *from* (Line 6) which will be used to decrease the refcounter (Line 14) and return a new object pointer whose refcounter has been increased (Line 10). However, **all of above operations are hidden to callers by the smartloop definition as the developers only care about the iteration purpose**. Consequently, when the *break* condition is satisfied (Line 21), most of the developers chose to directly break out (Line 22) and then miss the chance to pair the increasing operation at the begin of the iteration. From another side, as we can see from the Table 3, the key word *foreach* has a very low similarity with the key words of refcounting API names.

RC API Key Word	Bug-Caused API Key Word					
	foreach	**find**	**parse**	**open**	**probe**	**register**
refcount	0.19	0.33	0.16	0.30	0.28	0.19
increase	0.22	0.35	0.29	0.23	0.25	0.24
get	0.32	0.73	0.61	0.43	0.46	0.48
hold	0.29	0.43	0.28	0.32	0.23	0.30
grab	0.27	0.52	0.33	0.36	0.28	0.29
retain	0.14	0.32	0.28	0.17	0.09	0.25
decrease	0.21	0.39	0.27	0.26	0.27	0.15
put	0.38	0.58	0.48	0.46	0.39	0.36
unhold	-0.13	0.10	-0.02	0.07	-0.03	-0.14
drop	0.22	0.33	0.38	0.22	0.25	0.30
release	0.33	0.53	0.43	0.48	0.49	0.37

Table 3. The semantic similarities between the key words of refcounting API names and bug-caused API names. All the results are calculated through *word2vec* based on more than one million of commit logs from 2005 to 2022.

5.2.2 Increasing-/Decreasing-Hidden Problems.

To understand the real impacts of the low semantic similarities, we select and analyze the API names of the intra-unpaired bugs that the developers totally do not realize to pair the refcounting operation in *any potential execution path*, which can be easily inferred from the patch description. Totally, we unexpectedly find 254 such kind of bugs in our dataset. In Table 3, we present the names of the bug-caused and related refcounting APIs with the corresponding semantic similarities. Note that we use *word2vec* [18] to calculate the word semantic similarities by training the CBOW [40] model with more than one million of the historical commit logs, including the code and comment text.

From the semantic similarity results in Table 3, we can see all the key words of bug-caused APIs have very low similarities with the “refcount", “increase” and “decrease”. Besides, there are also very low similarities between the

bug-caused APIs and the key words of general refcounting APIs, e.g., "get"/"put", "hold"/"unhold', "grab"/"drop" and "retain"/"release". From this point, we can explain that, **when the developers use the *find*-like or *parse*-like APIs, they usually will not realize the existence of the refcounting behavior**. Note that there is a high similarity between the "find" and the "get"(0.73), even the "put"(0.58). The main reason is that the *find*-like API always call the *get*-named or even *put*-named refcounting APIs. For example, we should also note that, in Listing 4, there is an *of_node_put* invocation, which means the *of_node_get* should be added if the *from* parameter is not NULL. In fact, we have detected 16 new such missing-increasing bugs.

5.2.3 Lessons and Anti-Patterns. The developers should consider to add the key words that can imply the refcounting behaviors in the names of the functions, in which there are indeed the refcounting operations. Otherwise, the other developers who plan to call these functions will have a high probability of forgetting to realize the refcounting operation, which will finally lead to the missing-refcounting bugs. Consequently, we believe that finding and detecting the missing-refcounting bugs caused by the low semantic similarities can be an interesting research direction.

For the bugs caused by hidden refcounting problems, we propose following anti-patterns:

Anti-Pattern 3: $\mathcal{F}_{start} \rightarrow \mathcal{M}_{\mathcal{SL}} \rightarrow \mathcal{S}_{break} \rightarrow \mathcal{F}_{end}$.

Anti-Pattern 4: $\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{G}_H|\mathcal{P}_H} \rightarrow \mathcal{F}_{end}$.

We use $\mathcal{M}_{\mathcal{SL}}$ to mean the macro-defined smartloop, and use $\mathcal{G}_H|\mathcal{P}_H$ to indicate the hidden refcounting APIs.

5.3 Overlooked Location

5.3.1 Error-Handle Problems. Error-Handling blocks are the locations where the developers usually put more attention on undoing the other important things, e.g., resource deallocation. By searching in our bug dataset, there are totally 110 bugs that are caused by developers who add the paired decreasing APIs in all paths except the error-handling blocks.

5.3.2 Indirect-Call Problems. We find the leak involved refcounting bugs can be frequently reported in the inter-paired functions. For example, the developers often call the increasing API in the *open* function of a specific file operations but fail to call the decreasing API in the *release* function. The paired *probe* and *disconnect* of *usb_driver* operations, *connect* and *shutdown* of *proto_ops* operations, *probe* and *remove* of *platform_driver* operations are the common cases where the bugs exist.

5.3.3 Direct-Free Problems. If the developers confirm that they are removing the last reference object, they prefer to directly use the *kfree* function to free the target object, not using the decreasing API. However, there are many decreasing APIs that are not only to decrease the refcounter, but also to release other pre-allocated resources. As a result, the direct-free operation will make the allocated resource leaked as they have no chance to be freed. For example, the commit-258ad2fe [3] fixes a direct-free caused bug that leaks a name string allocated in the object initialization. Totally, there are 44 bugs that are caused by the direct-free problems.

5.3.4 Lessons and Anti-Patterns. The developers should not directly use *kfree* function in any case for the refcounted object. The main difficulty is to identify the refcounted object as it is possible that an object has no refcounter but only contains another refcounted object, e.g., the *device* structure only contains the refcounted *kobject* stucture but not its own refcounter. Fortunately, there are often the unique refcounting APIs, e.g., the *get_device*/*put_device* for the *device* structure.

We use followings anti-patterns to describe the bugs that are caused by overlooked locations:

Anti-Pattern 5: $\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{G}} \rightarrow \mathcal{S}_{\mathcal{P}}|\mathcal{B}_{error} \rightarrow \mathcal{F}_{end}$.

Anti-Pattern 6. $\mathcal{F}^{\top}_{start} \rightarrow \mathcal{S}_{\mathcal{G}} \rightarrow \mathcal{F}^{\top}_{end} \wedge \mathcal{F}^{\bot}_{start} \rightarrow \mathcal{F}^{\bot}_{end}$.

Anti-Pattern 7: $\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{G}} \rightarrow \mathcal{S}_{free} \rightarrow \mathcal{F}_{end}$.

Here, we use $\mathcal{S}_{\mathcal{P}}|\mathcal{B}_{error}$ to mean the two paths, one containing the decreasing API and another one containing the error-handling code. We use $\mathcal{F}^{\top}$ and $\mathcal{F}^{\bot}$ to refer to two inter-paired APIs, e.g., the *xx_probe* and *xx_destroy*. Finally, we use the $\mathcal{S}_{free}$ to mean *kfree* invocation.

5.4 Future Risk

5.4.1 Potential Deallocation Problems. This problem is a surprising finding, as an important root causes for the refcounting bugs that can potentially lead to the high-risk UAF bugs. Specifically, when the developers try to drop the references, they take it for granted that it is safe to access the object through the reference which has been used to decrease the refcounter. The main reason is that **some of the developers firmly believe that in all paths of current release the refcounter cannot reach zero** (see the details in Figure 4(1) and replies from the developers in Figure 4(3)). Unfortunately, in future, a new developer can call the UAD-caused API without the decreasing operation, which will free the object if the reference counter is one when the API is called. There have been 94 bugs caused by the UAD problems.

5.4.2 Reference Escape Problems. Reference escape problem has been indeed proved to introduce potential UAF bugs [35, 51]. Specifically, as shown in Figure 4(2), if there is a new assignment which can make the reference escaped out of current function, it is better to add a corresponding increasing operation around the escape point, not outside of the function, otherwise the escaped reference will potentially cause the UAF bug when new buggy path is added. Totally, there have been 74 bugs caused by escape problems.

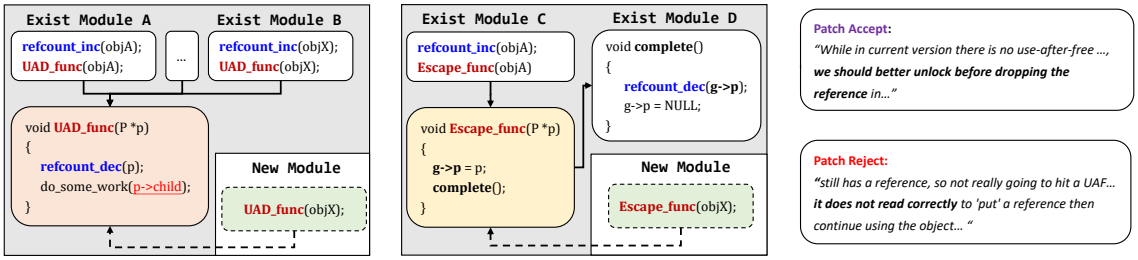

Figure 4. Future Risks.

5.4.3 Lessons and Anti-Patterns.

The developers should use the reference before the decrement and also should add the increment around the escape points, e.g., a new reference creation into a global variable or an out parameter. While there have been several methods to solve escape problem [35, 51], considering the refcounting optimization [14, 15, 27], it is still a challenge to completely defeat these bugs. Besides, automatically generation of the PoCs for the UAD bugs is also an interesting research direction.

We use followings anti-patterns to describe the bugs that are caused by future risks:

Anti-Pattern 8. $\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{P}(p_0)} \rightarrow \mathcal{S}_{\mathcal{D}(p_0)} \rightarrow \mathcal{F}_{end}$.

Anti-Pattern 9: $\mathcal{F}_{start} \rightarrow \mathcal{S}_{\mathcal{A}_{G|O}} \rightarrow \mathcal{F}_{end}$.

Here, we use $\mathcal{S}_{\mathcal{P}(p_0)}$ and $\mathcal{S}_{\mathcal{D}(p_0)}$ to refer to the deceasing and dereference with a same pointer p_0. We use $\mathcal{S}_{\mathcal{A}_{G|O}}$ to indicate an assignment to a global variable or out parameter, which creates an escaped reference.

6 Anti-Pattern Instance Detection

6.1 Static Checker Based Detection

Driven by the anti-patterns extracted from the root causes, we have designed and implemented a static checker for each of the anti-patterns. Our checkers will be released in https://github.com/windhl/checkers_sosp23. To apply the checkers on the Linux kernel, we use following steps.

Lexer Parsing ($\mathcal{G}$, $\mathcal{P}$, $\mathcal{M}_{\mathcal{SL}}$). Firstly, by extending the existing lexer-parsing tool, PLY [42], we implement three lexer parsers to explore refcounting related structures, refcounting APIs, and macro-defined smartloop. Specifically, the refcounting-related structures can be used to confirm the refcounting APIs, i.e., checking if the functions containing the structure instances and operating (increase or decrease) the refcounters. The refcounting APIs and smartloops can be used to generate the critical semantic operators, i.e., $\mathcal{G}$, $\mathcal{P}$ and $\mathcal{M}_{\mathcal{SL}}$. Note that the structure parser relies on a threshold to control the parsing levels as a refcounted object can be used in another structures, which can be nested defined.

Graph Generation($\mathcal{A}$, $\mathcal{D}$, $\mathcal{U}$, $\mathcal{S}$, $\mathcal{B}$, $\mathcal{F}$). Secondly, we transform the whole kernel source files into rich information embedded graphs, namely *Code Property Graphs* (CPGs) [49], with the tool JOERN [29] and use the embedded *Abstract Syntax Trees* (ASTs) to directly identify the other critical semantic operators and contexts, e.g., $\mathcal{A}$, $\mathcal{D}$, $\mathcal{B}$, $\mathcal{F}$. Besides, we will directly use the line numbers embedded in the graph nodes to represent the execution orders.

Bug Detection. Finally, we construct the nine static checkers based on the proposed anti-patterns with the related semantic operators and contexts. Then we detect new refcounting bugs by applying each of the static checker on all source code files to search and match the corresponding anti-pattern instances .

> **Why not LLVM.** We actually have heavily tried LLVM, but failed to compile all architecture-specific and subsystem code, which requires many combinations of compilation flags.

Subsystem	New Bugs	Impacts			Status		#FP
		Leak	UAF	NPD	#CFM	#PR	
arch	**156**	135	18	3	91	0	1
drivers	**182**	149	29	4	137	2	4
include	**2**	2	0	0	2	0	0
net	**2**	1	1	0	1	1	0
sound	**9**	9	0	0	9	0	0
Total	**351**	296	48	7	240	3	5

Table 4. The new refcounting bugs. **NPD** means null pointer dereference. **CFM** means confirm, **, PR** mean patch reject, **FP** means false positives. More details can be seen in Table 5. Note that we do not count the 5 FP bugs into total number.

6.2 New Instances (Bugs)

Overall, as shown in in Table 4, our checkers have totally detected 351 new refcounting bugs. The new bugs exist in five subsystems or directories: *arch*, *drivers*, *include*, *net* and *sound*. We present the details of the bug distributions in Table 5. While we have sent the patch for each of the bugs, when writing this paper, 240 ones have been confirmed by the developers, 3 ones have been rejected and 5 ones have been proved as false positives (not contained in the total number). For other 111 bugs, we get no response.

Subsystem	Module	Bug-Caused API (Top-2)	#Anti-Pattern Instance	#Bug	Confirmed
arch	arm	of_find_compatible_node[19], of_find_matching_node[11]	**P4**[42], **P6**[2], **P7**[2], **P9**[4]	50	18
	microblaze	of_find_matching_node[1]	**P4**[1]	1	**NR**
	mips	of_find_compatible_node[15], of_find_matching_node[1]	**P4**[17]	17	16
	powerpc	of_find_compatible_node[13], of_find_node_by_path[9]	**P3**[8], **P4**[48], **P5**[1], **P6**[2], **P8**[1], **P9**[5]	65	55
	sh	of_find_compatible_node[1]	**P4**[1]	1	**NR**
	sparc	of_find_node_by_name[8], for_each_node_by_name[4]	**P2**[3], **P3**[4], **P4**[10], **P7**[1], **P9**[1]	19	**NR**
	x86	of_find_compatible_node[1], of_find_matching_node[1]	**P4**[2]	2	**NR**
	xtensa	of_find_compatible_node[2]	**P4**[2]	2	2
drivers	block	mdesc_grab[1]	**P2**[1]	1	1
	bus	of_find_matching_node[2], of_find_node_by_path[2]	**P3**[1], **P4**[7]	8	4
	clk	of_get_node[33], of_find_matching_node[3]	**P4**[37]	37	36
	clocksource	of_find_compatible_node[1]	**P4**[1]	1	**NR**
	cpufreq	of_find_node_by_name[3], of_find_matching_node[1]	**P4**[4]	4	4
	crypto	of_find_compatible_node[4]	**P4**[4]	4	4
	dma	of_parse_phandle[1], for_each_child_of_node[1]	**P3**[1], **P5**[1]	2	1
	edac	of_find_compatible_node[1]	**P4**[1]	1	**NR**
	firmware	of_find_compatible_node[1]	**P4**[1]	1	**NR**
	gpio	of_get_parent[2], of_node_get[1]	**P4**[2], **P6**[1], **P9**[1]	4	2
	gpu	of_graph_get_port_by_id[6], of_get_node[3]	**P3**[3], **P4**[5], **P5**[3], **P6**[2], **P8**[2], **P9**[2]	17	12
	hwmon	of_find_compatible_node[2]	**P4**[2]	2	2
	i2c	device_for_each_child_node[1], for_each_child_of_node[1]	**P3**[2]	2	1
	iio	device_for_each_child_node[1], of_find_node_by_name[1]	**P3**[1], **P4**[1]	2	1
	input	of_find_node_by_path[2]	**P4**[2]	2	2
	iommu	for_each_child_of_node[1]	**P3**[1]	1	1
	irqchip	of_find_matching_node[2], of_find_node_by_phandle[1]	**P4**[3]	3	**NR**
	leds	fwnode_for_each_child_node[1]	**P3**[1]	1	1
	macintosh	of_find_compatible_node[2], of_node_get[1]	**P4**[2], **P6**[1]	3	3
	media	for_each_compatible_node[1], for_each_child_of_node[1]	**P3**[2]	2	1
	memory	of_find_node_by_name[3], for_each_child_of_node[2]	**P3**[4], **P4**[2]	6	3
	mfd	pm_runtime_get_sync[1]	**P1**[1]	1	1
	mmc	for_each_child_of_node[3], of_find_compatible_node[1]	**P3**[3], **P4**[1]	4	4
	net	for_each_child_of_node[3], of_find_compatible_node[1]	**P2**[2], **P3**[5], **P4**[12]	19	16
	nvme	nvmet_fc_tgt_q_put[1]	**P8**[1]	1	**PR**
	of	of_parse_phandle[1]	**P4**[1]	1	1
	opp	–	**P9**[2]	2	2
	pci	of_parse_phandle[2], of_find_matching_node[1]	**P4**[2], **P5**[1]	3	1
	perf	for_each_cpu_node[1]	**P3**[1]	1	1
	phy	for_each_child_of_node[1], of_parse_phandle[1]	**P3**[1], **P4**[2]	3	1
	pinctrl	of_find_node_by_phandle[1]	**P4**[1]	1	**NR**
	platform	device_for_each_child_node[2], fwnode_for_each_child_node[1]	**P3**[3]	3	2
	powerpc	of_find_compatible_node[1]	**P4**[1]	1	1
	regulator	of_find_node_by_name[1], of_get_child_by_name[1]	**P4**[2]	2	2
	sbus	of_find_node_by_path[2]	**P4**[2]	2	**NR**
	soc	of_find_compatible_node[4], of_get_parent[2]	**P3**[3], **P4**[7], **P5**[1], **P6**[1], **P9**[1]	13	11
	thermal	of_node_get[1]	**P6**[1], **P9**[1]	2	2
	tty	mdesc_grab[1], of_find_node_by_type[1]	**P2**[1], **P4**[2], **P6**[1]	4	3
	ufs	of_parse_phandle[1]	**P4**[1]	1	1
	usb	of_find_node_by_name[4], usb_serial_put[1]	**P4**[6], **P8**[1]	7	7
	video	of_graph_get_next_endpoint[1], of_graph_get_remote_port_parent[1]	**P4**[3]	3	2
	w1	of_find_matching_node[4]	**P4**[3],**P5**[1]	4	**NR**
include	linux	of_find_compatible_node[2]	**P4**[2]	2	2
net	appletalk	dev_hold[1]	**P4**[1]	1	1
	ipv4	sock_put[1]	**P8**[1]	1	**PR**
sound	soc	of_find_compatible_node[2], of_graph_get_port_parent[2]	**P4**[8], **P5**[1]	9	9
Total	**54***	**of_find_compatible_node[74], of_get_parent[48]**	**P1-P9: [1, 7, 42, 255, 9, 12, 3, 5, 17]**	**351**	**240**

Table 5. The details of new bugs. Considering the space limitation, we only list the Top-2 bug-caused APIs. [N] means the bug number. **P** means anti-patterns, **NR** means no response, **PR** means patch reject. * means the number of buggy modules.

Listing 5. A False Postive Example.

```
1 //drivers/scsi/lpfc/lpfc_bsg.c
2 list_for_each_entry(...) {
3     if (evt->reg_id == event_req->ev_reg_id)
4         lpfc_bsg_event_ref(evt); // increasing
5 }
6 if (&evt->node == &phba->ct_ev_waiters) {
7     evt = lpfc_bsg_event_new(...);
8 }
```

Listing 6. A Patch Rejct Example.

```
1 //net/ipv4/ping.c
2 void ping_unhash(struct sock *sk) {
3     sock_put(sk);
4     isk->inet_num = 0;
5     isk->inet_sport = 0;
6     sock_prot_inuse_add(..., sk->sk_prot,...);
7 }
```

From the second columns, we can see the number of new refcounting bugs for each subsystem. First, we unexpectedly detected more than one hundred of new bugs both in *arch* and *drivers*, which totally contains 338 new bugs, i.e., about 96% of all bugs. Within the buggy modules, as shown in Table 5, all of the bugs also meet the long-tailed distributions, consistent with our Finding 3. Second, we totally detected 13 bugs in other subsystems and directories. It is notable that we found 2 bugs even in two header files in the *include* directory. One is in the *include/linux/hypervisor.h*, which involves the virtualization. The other is in the *include/linux/firmware/trusted_foundation.h*, which involved the Trusted Foundation of some ARM devices. Besides, we also detected two bugs in the *net* subsystem, which involve the legacy *appletalk* and the core *ping* sub-modules. Finally, we detected 9 bugs in the *soc* module of the *sound* subsystem.

6.3 Security Impacts

From the third column of Table 4, we can see the security impacts of above new refcounting bugs. Specifically, there are totally 296 (84.3%) bugs which can finally lead to leak bugs, 48(13.7%) ones to UAF and 7 (2.0%) ones to NPD. The distribution of leak bugs and UAF bugs are also consistent with our previous findings. The NPD bugs are caused by the increasing APIs who may return a NULL pointer.

6.4 Patch Committing

From the *Status* main column, we present the details of the bug patches. We have sent the patch for each of the bugs. When writing this paper, there are totally 240 bugs have been confirmed and their patches are applied in the mainline. Besides, there are 111 bugs that have not been replied by the developers and 4 bugs are refused to be confirmed as the developers do not think they are real bugs unless we can provide proper PoCs. Finally, there are 5 bugs have been proved as false positives.

False Positives. The main reason of the false positives of our checkers is that we have not tried to analyze the semantic of specific structures and operations. For example, we use a real-world case during our evaluation, shown in Listing 5, to explain the reason. First, our checkers identified the refcounting operation in Line 4, i.e., the *lpfc_bsg_event_ref*, used as the increasing API for *evt* object. Then, out tool identified the replacement of *evt* in Line 10 and reported a missing-decreasing bug. However, when we submitted a corresponding patch, the developers told us that the *if* condition before the replacement, in Line 8, ensures the correctness. Specifically, when running into the *if*-block, the *evt* will always be a NULL pointer after the iteration of *list_for_each_entry* and the increasing operation has no chance to be executed.

Patch Rejects. Totally, we got three patch rejects. As we have said before, the main reason of the rejects is that the developers do not think they are real bugs, e.g., "only not read correctly" (Figure 4 (3)). For example, as shown in Listing 6, our checkers detected a UAD bug in the *ping_unhash* function of the *net/ipv4/ping.c*. Specifically, while there has been a decreasing operation in Line 4 for the *sk* object, its pointer is dereferenced in Line 7. While there are two patch rejects for the UAD bugs, other 3 new UAD bugs have been already confirmed. Generating the PoCs for the UAD bugs is an important direction in the future work.

Potential False Negatives. While we proposed anti-patterns for the most common refcounting bugs, we will miss the ones caused by other complicated reasons, e.g., race problems.

7 Lessons From New Bugs

In this section, we will conclude the valuable lessons from the new bugs based on the four main kinds of root causes.

Implementation Deviation Caused Bugs. While there have been more than one hundred of historical bugs caused by the *pm_runtime_get_sync*, we can still find a new one in the *mfd* module of the *drivers* subsystem. The detection of this bug also means we cannot prevent new bugs until the developers realize the implementation deviations of different refcounting functions. It is notable that we also detect 7 Return-NULL caused bugs, i.e., the number of P2 in both of *arch* and *drivers* subsystems. When writing this paper, 3 out of them have been confirmed and fixed in the mainline, which proved the effectiveness of our new anti-pattern.

Hidden API Caused Bugs. We find that there are 62 bugs caused by the hidden increasing or decreasing functions or macros in the *arch* and *drivers* subsystems. Specifically, we totally detected 23 bugs caused by the hidden decreasing functions, e.g., the *of_find_node_by_name*. Besides, we detected 39 bugs caused by the break of smartloop without proper decreasing. Except *for_each_matching_node* (listed in Listing 4), we also identified other ones by our lexer parser,

e.g., *for_each_child_of_node*, *for_each_node_by_name*, *fwnode_for_each_parent_node*, *device_for_each_child_node*.

Overlooked Location Caused Bugs. We can see there are totally 24 bugs caused by the three kinds of overlooked location problems. First, there are 9 bugs are caused by the error-handling problem. In fact, during our detection, there are two kinds of error-handling locations, one is the premature exist (return) under a specific *if*-condition block, anther one is located by the *error*-labels. Second, we totally detected 13 bugs caused by inter-unpaired refcounting APIs. Without any inter-procedural analysis, we simplify the detection by using our lexer parser to identify all the initialization of global variables whose member fields contains paired function pointers, e.g., the *struct i2c_driver* and the *struct platform_driver*. Besides we also detected the paired of functions by simply matching their names with the paired words, e.g., the pairs of *register/unregister* or *create/destroy*, *init/uninit*. Then we confirmed if the developers only added an increasing refcounting API but missed the corresponding increasing one. By this way, we detected 12 new bugs. Finally, we can see that there are 3 bugs caused by directly using the *kfree* functions without the proper decreasing functions, which will inevitably cause memory leak.

Future Risk Caused Bugs. There are totally 5 new bugs caused by potential deallocation problems. While there are 3 bugs are confirmed and fixed by the developers, other 2 bugs are not be confirmed as other developers firmly believed there would be no any UAF bugs in current version, which we have shown in Figure 4(3). In fact, we restate that generating the PoC for UAD bugs is an important research direction based on the fact that there are so many bugs. For the escape problem, we use the AST to detect all the assignment statements that involved the refcounted objects and search around if there is any increasing refcounting API invocations. Considering the refcounting omission (optimization), we have manually filtered out and confirmed the escape-caused bugs and finally we found 17 new bugs and all of them have been confirmed and fixed in the mainline.

8 Related Work

Considering the security impacts of refcounting bugs, many detection methods have been proposed. We category all of these methods into four categories based on the various strategies. In comparison, this paper is the first in-depth characteristics study of historical refcounting bugs in Linux kernels, inferring the root causes for the majority of bugs and extracting anti-patterns to detect new bugs.

Inconsistent-Refcounting Methods. The key insight of these methods is very straight [39, 48]. Specifically, whenever there is an refcounter increment, there should be a paired decrement. Otherwise there must be a refcounting bug. While it is simple, it has detected lots of refcounting bugs, mainly involved the missing-refcounting bugs. The main limitation of these methods is that there are many refcounting omissions (optimizations) and they will break the consistence rules. A recent work [27], aiming to diagnose the UAF bugs caused by refcounting problems, presents an refcounting omission-aware model and detects the refcounting bugs based on dynamic analysis.

Invariant-Analysis Methods. This kind of methods are first proposed in compiler-optimizations [51], aiming to reduce redundant refcounting operations and improve the system performance. The basic idea is to guarantee the invariant that the number of escaped references should be equal to the increment number of the refcounter. Accordingly, based on this idea, the methods, like [26, 35, 44], can catch a potential refcounting bug if there is any violation of the invariant rule. However, this strategy will be not efficient in large-scale programs, such as the OS kernels, in which there are many inlined APIs or macro-defined functions which will break the invariant rule. For example, the recent work [35], adopting invariant-guarantee checking, suffers from a high false positives (about 60%) when applied into Linux kernels.

Cross-Checking Methods. Cross-Checking-based methods [32, 38, 41] are commonly used in detecting many semantic bugs. In fact, the work [48] has also use cross-checking as its second strategy to detect the refcounting bugs. Specifically, when there is a missing refcounting, they will make a cross-check within other similar places where a same object reference is created or destructed. Then, they can infer that if there is a refcounting bugs based on the common behaviors of most cases. While this strategy has also been proven to be efficient in detecting refcounting bugs, they also suffer from lots of false positives as the refcounting optimization.

Template-based Methods. While this kind of methods are very effective in detecting the shallow refcounting bugs, as the lack of deep analysis, they cannot detect inter-unpaired operations in different modular functions. It is worse that the functions are often indirectly called by function pointers. Besides, this kind of methods usually need many manual works to build the meaningful semantic templates, e.g., Coccinelle-based methods or tools [20, 23, 43]. While a recent work [37] is proposed to automatically generate the semantic templates, it is a general dynamic method, only focusing on semantic failures, not semantic bugs.

9 Conclusion

We present the first in-depth analysis of refcounting bugs in all modern Linux kernels. We find the majority of refcounting bugs can cause security impacts, leading to leak or UAF bugs. Besides the root cause inferences, we also propose meaningful anti-patterns, by which we design and implement static checkers, which help us to detect 351 new refcounting bugs and 240 of them are confirmed. We hope the bug lessons and the anti-patterns can be helpful to motivate future proactive solutions for the refcounting problems.

10 Acknowledgments

We thank the anonymous reviewers, and our shepherd, Dongyoon Lee, for their helpful feedback. This research was supported, in part, by National Natural Science Foundation of China (Grand No. 62232016, 61972224, 61872386), National Key Research and Development Program of China (Grand No. 2021YFB2701000), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDC02020300) the Basic Research Program, ISCAS (Grand No. ISCAS-JCZD-202301), the Key Research Program of Frontier Sciences, CAS (Grand No. ZDBS-LY-7006), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. 2019111 and Y2021041). All options expressed in this paper are solely those of the authors.

References

[1] [n. d.]. Commit-0a96fa64: Fix improper handling of refcount in uss720_probe. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.2-rc8&id=0a96fa640dc928da9eaa46a22c46521b037b78ad

[2] [n. d.]. Commit-1085f508: Fix refcount leak and iomem leak bugs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.2-rc8&id=1085f5080647f0c9f357c270a537869191f7f2a1

[3] [n. d.]. Commit-258ad2fe: Fix possible memory leak. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.2&id=258ad2fe5ede773625adfda88b173f4123e59f45

[4] [n. d.]. Commit-3e7d18b9:fix reference count leak. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.5-rc5&id=3e7d18b9dca388940a19cae30bfc1f76dccd8c28

[5] [n. d.]. Commit-87710394: Fix PM usage reference leak. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.2&id=8771039482d965bdc8cefd972bcabac2b76944a8

[6] [n. d.]. Commit-bf4a9b24: Fix refcount leak in aries_audio_probe. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/sound/soc/samsung/aries_wm8994.c?h=v6.5-rc5&id=bf4a9b2467b775717d0e9034ad916888e19713a3

[7] [n. d.]. Commit-dcb4b8ad: Fix memory leak in uss720_probe. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.2&id=8771039482d965bdc8cefd972bcabac2b76944a8

[8] [n. d.]. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization. https://cwe.mitre.org/data/definitions/362.html

[9] [n. d.]. CWE-401: Missing Release of Memory after Effective Lifetime. https://cwe.mitre.org/data/definitions/401.html

[10] [n. d.]. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html

[11] [n. d.]. CWE-911: Improper Update of Reference Count. https://cwe.mitre.org/data/definitions/911

[12] [n. d.]. Kernel Boot Problem. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0711f0d7050b9e07c44bc159bbc64ac0a1022c7f.

[13] [n. d.]. Linux Kernel Source Tree. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/

[14] [n. d.]. NUMA Memory Policy. https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html

[15] [n. d.]. Rules For Managing Reference Counts. https://docs.microsoft.com/zh-cn/windows/desktop/com/rules-for-managing-reference-counts

[16] [n. d.]. Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/latest/process/submitting-patches.html

[17] [n. d.]. USB serial. https://www.kernel.org/doc/html/latest/usb/usb-serial.html

[18] [n. d.]. Word2vec embeddings. https://radimrehurek.com/gensim/models/word2vec.html.

[19] J. A. Goguen. 1974. Semantic Computation. In Proceedings of the First International Symposium on Category Theory Applied to Computation and Control.

[20] Julien Brunel, Damien Doligez, Rene R. Hansen, Julia Lawall, and Gilles Muller. 2009. A Foundation for Flow-Based Program Matching Using Temporal Logica and Model Chekcing. In Proceedings of 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages.

[21] Costin Carabas and Mihai Carabs. 2017. Fuzzing the Linux Kernel. In 2017 Computing Conference.

[22] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. SIGOPS Operating Systems Review 35, 5 (2001), 73–88.

[23] Coccinelle 2019. Coccinelle: A Program Matching and Transformation Tool for Systems Code.

[24] George E. Collins. 1960. A Method For Overlapping And Erasure Of Lists. Commun. ACM 3, 12 (Dec. 1960), 655–657.

[25] Paul E.McKenney. [n. d.]. Overview of Linux-Kernel Reference Counting. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2167.pdf

[26] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. 2009. Verifying Reference Counting Implementations. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems.

[27] Liang He, Hong Hu, Purui Su, and Yan Cai. 2022. FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting on Binaries. In Proceedings of the 31th USENIX Security Symposium.

[28] Rafael J. Wysocki and Alan Stern. [n. d.]. Runtime Power Management Framework. https://www.kernel.org/doc/html/latest/power/runtime_pm.html

[29] JOERN [n. d.]. JOERN: The Bug Hunter's Workbench. https://joern.io/

[30] Srinivas Kandagatla. [n. d.]. NVMEM Subsystem. https://www.kernel.org/doc/html/latest/driver-api/nvmem.html

[31] Kyungtae Kim, Dae R. Jeong, Chung H. Kim, Yeongin Jiang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In Proceedings of the Network and Distributed System Security 2020.

[32] Seulbae Kim, Meng Xu, Sanidhy Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In Proceedings of the 27th Symposium on Operating Systems Principles.

[33] Greg Kroah-Hartman. [n. d.]. Everything you never wanted to know about kobjects, ksets, and ktypes. https://www.kernel.org/doc/html/latest/core-api/kobject.html

[34] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Zhai Chengxiang. 2006. Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software. In Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability.

[35] Jian Liu, Lin Yi, Weiteng Chen, Chenyu Song, Zhiyun Qian, and Qiuping Yi. 2022. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In Proceedings of the 31th USENIX Security Symposium.

[36] LKML 2023. The Linux Kernel Mailing Lists. Retrieved Feb 8, 2023 from https://lore.kernel.org

[37] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

[38] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and

Constraints Inference. In Proceedings of the 28th USENIX Security Symposium.

[39] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. Rid: Finding Reference Count Bugs with Inconsistent Path Pair Checking. In Proceedings of the 21st International Conference on Architecture Support for Programming Lagnuages and Operating Systems.

[40] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. [n. d.]. Efficient Estimation of Word Representations in Vector Space. https://arxiv.org/pdf/1301.3781.pdf.

[41] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In Proceedings of the 25th Symposium on Operating Systems Principles.

[42] PLY [n. d.]. PLY: Python Lex-Yacc. https://ply.readthedocs.io/

[43] Luis R. Rodrigue and Julia Lawall. 2015. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In Proceedings of 11th European Dependable Computing Conference.

[44] Li Siliang and Tan Gang. 2014. Finding Reference-counting Errors in Python/C Programs with Affine Analysis. In Proceedings of European Conference on Object-Oriented Programming.

[45] Hao Sun, Yuheng Shen, Cong Wang, JianZhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In Proceedings of the 28th Symposium on Operating Systems Principles.

[46] Syzkaller 2023. Syzaller - Kernel Fuzzer. Retrieved Feb 16, 2023 from https://github.com/google/syzkaller

[47] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug Characteristics in Open Source Software. Empir Software Eng 6, 19 (2014), 1665–1705.

[48] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. 2021. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In Proceedings of the 30th USENIX Security Symposium.

[49] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graph. In Proceedings of the 35th IEEE Symposium on Security and Privacy.

[50] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs? A Comprhensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating System. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.

[51] Park Young Gil and Goldberg Benjamin. 1991. Reference Escaple Analysis: Optimizing Reference Counting based on the Lifetime of References. In Conference on Partial Evaluation and Program Manipulation.

A Error-Prone APIs

Here we list the related APIs that are error-prone to the refcounting bugs.

Bug Type		APIs
ID	Return-Error	*pm_runtime_get_sync* *kobject_init_and_add*
	Return-NULL	*mdesc_grab* *amdgpu_device_ip_init*
H	Complete-Hidden	*for_each_child_of_node* *for_each_available_child_of_node* *for_each_endpoint_of_node* *for_each_node_by_name* *for_each_compatible_node* *device_for_each_child_node* *fwnode_for_each_parent_node*
	Inc./Dec.-Hidden	*of_parse_phandle* *of_get_parent* *of_get_child_by_name* *of_find_compatible_node* *of_find_matching_node* *of_find_node_by_name* *of_find_node_by_path* *of_find_node_by_phandle* *of_find_node_by_type* *device_initialize* *ip_dev_find* *afs_alloc_read* *perf_cpu_map__new* *setup_find_cpu_node* *gfs2_glock_nq_init* *tipc_node_find* *sockfd_lookup* *fc_rport_lookup* *rxrpc_lookup_peer* *lookup_bdev* *__tcp_ulp_find_autoload* *__ipv4_neigh_lookup* *class_find_device* *mpol_shared_policy_lookup* *usb_anchor_urb* *tomoyo_mount_acl*

Table 6. The Error-Prone APIs. **ID**=Implementation Deviation, **H**=Hidden.