Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking

Xin Tan[1, ¶], Yuan Zhang[1, ¶], Xiyu Yang[1], Kangjie Lu[2], and Min Yang[1]

[1]*School of Computer Science, Fudan University, China*
[2]*Department of Computer Science & Engineering, University of Minnesota, USA*
[¶]*co-first authors*

Abstract

In the Linux kernel, reference counting (refcount) has become a default mechanism that manages resource objects. A refcount of a tracked object is incremented when a new reference is assigned and decremented when a reference becomes invalid. Since the kernel manages a large number of shared resources, refcount is prevalent. Due to the inherent complexity of the kernel and resource sharing, developers often fail to properly update refcounts, leading to refcount bugs. Researchers have shown that refcount bugs can cause critical security impacts like privilege escalation; however, the detection of refcount bugs remains an open problem.

In this paper, we propose CID, a new mechanism that employs two-dimensional consistency checking to automatically detect refcount bugs. By checking if callers consistently use a refcount function, CID detects deviating cases as potential bugs; by checking how a caller uses a refcount function, CID infers the condition-aware rules for the function to correspondingly operate the refcount, and thus a violating case is a potential bug. More importantly, CID's consistency checking does not require complicated semantic understanding, interprocedural data-flow tracing, or refcount-operation reasoning. CID also features an automated mechanism that systematically identifies refcount fields and functions in the whole kernel. We implement CID and apply it to the Linux kernel. The tool found 44 new refcount bugs that may cause severe security issues, most of which have been confirmed by the maintainers.

1 Introduction

The Linux kernel becomes more and more important, especially with its wide use in cloud platforms, mobile devices, and IoT devices. A core functionality of the kernel is to manage the shared resources (e.g. peripherals and files). Since the Linux kernel is implemented in C language which does not support automatic garbage collection or smart pointers, it relies on reference counting (refcount for short) to keep track of the uses of a variety of resources. Naturally, refcount is quite prevalent in the Linux kernel to maintain a large number of shared resource objects. Our study also reveals that the Linux kernel alone (except third-party drivers) has about 800 structs that are managed with the refcount mechanism.

In essence, a refcount is an integer that tracks the number of references to the tracked resource object. To facilitate the uses of refcounts, the Linux kernel offers specific data types and manipulation APIs. For example, `refcount_t` is defined in the Linux kernel to represent a refcount field, while `refcount_inc()` and `refcount_dec()` are two primitive APIs for increasing (`INC`) and decreasing (`DEC`) a `refcount_t` field respectively. The refcount field of an tracked object is increased when there is a new reference to the object or decreased when a reference becomes invalid. The reference counting approach guarantees that an object is freed only when its refcount reaches zero.

Since the refcount needs to be updated manually, the developers are required to have a clear understanding of their intended uses on system resources and then perform correct refcount operations, which are actually challenging in the complex Linux kernel. On one hand, the programmers may make mistakes due to the complexity of the kernel logic. On the other hand, different functions of the same kernel module might be developed by different developers. The developer of a function may not know the details of other functions, which often leads to incorrect refcount operations. As a result, the refcount operations in the Linux kernel are error-prone.

Our study shows that there are two main kinds of refcount errors. (1) *Over decrease.* This buggy case occurs when the developer redundantly calls a refcount-decrease API or under an over-relaxed condition. A redundant decrease may cause the refcount to prematurely reach zero, which will incorrectly trigger object release. That is, the memory associated to the victim object is freed while there are still legitimate references to it. If the kernel still references to the object while it has been freed, critical use-after-free (UAF) occurs. (2) *Missing decrease*. By contrast, another kind of bug is that a necessary refcount decrease is missing. This often leads to a resource

leak (e.g., memory leak) because the recount may never reach zero, and the resource will never be released. Attackers can exploit such bugs to launch denial-of-service, such as crashing the system, by repeatedly triggering the bug. Interestingly, missing refcount release may also lead to UAF, i.e., when the refcount field overflows to be zero, the kernel will wrongly free it while some legitimate references are still used.

Refcount bugs have severe impact on the security of the system. In particular, security researchers have reported many serious refcount vulnerabilities (e.g., CVE-2016-4558, CVE-2016-0728, CVE-2019-11487), which can be exploited for privilege escalation, putting a lot of real-world devices at risk. Take CVE-2016-0728 as an example, by continuously triggering the missing-decrease path in the keyrings facility, the refcount bug finally overflows the *usage* counter, triggering a UAF vulnerability. The UAF vulnerability is further successfully exploited to perform a local privilege escalation attack. Even worse, this vulnerability is quite stealthy for hiding in the kernel for about 4 years until it was discovered. As a result, tens of millions of Linux PCs/servers, including 66% of the Android devices (phones/tablets) [2] are affected. That's to say, the refcount bugs may be too latent to discover, yet cause critical security impact against numerous devices and users.

Given the severe impact of refcount bugs, it is important to detect them in the Linux kernel. However, there are significant challenges in the detection.

Challenge-I: Lacking a refcount bug oracle. Refcount bugs happen when there is a mismatch between refcount `INC` operations and `DEC` operations. However, when to perform `DEC` operations largely depends on the purpose of the developers and the usage of the tracked object. Therefore, there lacks an oracle that models refcount bugs. Existing works mainly adopt two strategies to detect refcount bugs. Pungi [24] optimistically assumes that the change of a refcount must equal to the number of references escaped from the function. However, it may incur overwhelming false positives in the Linux kernel, since many kernel functions (e.g., refcount wrapper APIs) just increase the refcounts with no reference escaped. To accurately detect refcount bugs, RID [29] observes that the paths sharing the same argument and the same return value in the same function should have consistent refcount behaviors. Based on this observation, it proposes inconsistent path pair checking to detect refcount bugs. Though this strategy helps to reduce false positives, its scope is so narrow that it misses the majority refcount bugs. According to our study, for the 60 refcount bugs reported between 2018 and 2020 from the Linux kernel [4], RID could only detect 10 of them.

Challenge-II: Recognizing wide-spread refcount fields. The prerequisite to detect refcount bugs is the recognition of refcount fields (i.e., struct fields that are manipulated by refcount operations). However, refcount fields can be buried in various types who serve for other purposes. It is non-trivial to achieve both accuracy and coverage in identifying refcount fields. According to our study described in §2.2, only 37 out of the 100 `atomic_t` fields which we checked are used for refcount, and the remaining are for other purposes. Therefore, existing works [24, 29] involve manual efforts to label a set of refcount fields/operations. Pattern-based [36] methods could also identify refcount fields in the Linux kernel. However, such methods work for only common refcount fields, but would miss less common or custom ones.

To address these challenges, this paper proposes CID[1], which first systematically identifies potential refcount fields/-operations in the Linux kernel and then automatically detects refcount bugs with two-dimensional consistency checking. Our bug detection is based on two unique observations: (1) `INC` and `DEC` operations enforce a strict mutual relation, oftentimes with pre-conditions; (2) `INC` and `DEC` functions for the same object are often invoked multiple times, following the same usage, and the usage is bug-free in most cases. The two observations motivate the design of our two-dimensional consistency checking. In one dimension, the `INC-DEC` consistency checking infers *condition-aware refcount rules* for the `INC` or `DEC` function by examining the `DEC` or `INC` operations and their conditions in another function. Then, it uses the rules to detect violating cases in the `INC` or `DEC` function as refcount bugs. A unique strenghth of the checking is that the inferred refcount rules apply regardless of the complicated data and control flows between the `INC` and `DEC` operations, thus it avoids the complicated tracing and inter-procedural data-flow analysis. In the other dimension, the `DEC-DEC` consistency checking recognizes deviating `DEC` operations, from the majority `DEC` operations paired with the same `INC` operation, as refcount bugs.

Compared to existing works, our two-dimensional checking does not rely on an aggressive or conservative bug oracle while the checked inconsistencies still effectively uncover refcount bugs. Meanwhile, CID introduces behavior-based refcount field identification, which distinguishes refcount fields from a large number of other fields based on their purposes. By summarizing the behavior features of refcount fields, CID systematically and automatically identifies the refcount fields that are defined in different data types at a high precision.

We implement a prototype of CID with the LLVM infrastructure [23]. CID incorporates several new techniques to realize refcount bug detection with the two-dimensional consistency checking. First, CID selects candidate refcount fields through type analysis and then identifies refcount fields with a behavior-based inference method. Second, CID collects the functions that perform `INC` operations on the identified refcount fields, and performs path-sensitive data flow analysis to model the `INC` behaviors in them including the `DEC` behaviors in their callers. At last, CID checks the consistencies over the modeled behaviors between paired `INC` and `DEC` operations from two dimensions to detect refcount bugs and generates bug reports.

[1] CID is named for Checking INC/DEC operations

To evaluate the effectiveness of CID, we apply it to the Linux kernel of version 5.6-rc2. CID finishes the analysis for the whole kernel within 18 minutes and reports 149 refcount bugs. From these bugs, we manually confirmed 44 new refcount bugs and submitted their patches to the Linux maintainers. Until now, 36 bugs have been confirmed, and the patches for the 34 bugs have already been applied to the kernel. These new bugs are also confirmed to have severe security impacts, including UAF, Denial of Service (DoS) and memory leak. We analyze the confirmed false-positive cases and find most of them resulted from the imprecise static analysis used in CID. We measure the bug detection capability of CID by comparing it with RID [29] (the state-of-the-art tool) on detecting 60 known refcount bugs. The results show that CID only misses 6 bugs while RID misses 50 bugs. Besides, the refcount field identification of CID is also evaluated to be quite effective, which identifies 792 refcount fields from the kernel with an accuracy of 94.3%.

In summary, we make the following contributions.

- **A New Approach for Refcount Bug Detection.** We propose a novel approach to detect kernel refcount bugs with two-dimensional consistency checking, which examines the inconsistencies between the `INC` operations and `DEC` operations without assuming a bug oracle.
- **A New Approach for Refcount Field Identification.** We present behavior-based inference approach to systematically identify refcount fields across the whole kernel. This technique generally facilitates other works relying on refcount identification.
- **New Bugs Detected by the Prototype.** We develop a prototype of CID and apply it to the Linux kernel. The tool found 44 new refcount bugs in the latest kernel, which cause severe security impacts. Among them, 36 bugs have been confirmed by the Linux maintainers.

The rest of the paper is organized as follows: §2 introduces the refcount mechanism in the Linux kernel and studies the challenges in refcount field identification; §3 illustrates the two-dimensional consistency checking with real-world examples; §4 and §5 present the design and implementation of CID; §6 evaluates the effectiveness of CID; §7 discusses our work; §8 presents the related work; finally, §9 concludes the paper.

2 Background

2.1 Refcount in the Linux Kernel

Reference count (refcount) is a common resource management mechanism. In the Linux kernel, the refcount mechanism is widely used in various subsystems for managing all kinds of resources, such as dynamically allocated memory blocks [30], device drivers [12]. In essence, a refcount is a numeric field counting the number of references to a specific resource object. The kernel developers often maintain a refcount field in the to-be-counted resource data structure to implement the refcounting mechanism. The refcount of a resource is incremented when a new reference is taken and decremented when a reference is released. It is important to note that, by design, when a refcount reaches zero, its corresponding resource is not being used and will be recycled automatically.

According to the kernel documentation [1, 3, 7, 30], refcount is typically manipulated through atomic operations that support concurrent allocation and release of a resource. Therefore, to avoid concurrency and performance issues, refcount is defined as an atomic integer. There are 5 data types for refcount definition in the Linux kernel—`atomic_t`, `atomic_long_t`, `atomic64_t`, `kref`, and `refcount_t`. `atomic_t`, `atomic_long_t` and `atomic64_t` are essentially of type `int`, `long`, and `s64`, respectively, whose size varies with the underlying architecture. Note that `atomic_t`, `atomic_long_t`, and `atomic64_t` generic types are not limited to refcount usage, i.e., they can be used for other purposes. The `kref` type is a refcount-specific type introduced by Greg [22], and it is subsequently replaced by `refcount_t` type in the latest kernel. Actually, the `kref` type has already been defined with `refcount_t` in the current kernel. The `refcount_t` type adds extra support to prevent accidental counter overflows and underflows, which is quite effective in reducing the severe UAF vulnerabilities. Although `refcount_t` is more secure than the other 4 types, it incurs obvious performance overhead. Besides, the conversion from old refcount types to the new `refcount_t` type requires significant efforts. Therefore, there are still a lot of legacy data structures using the old types [35], and some time-sensitive scenarios clearly refuse this new type [16].

Based on the refcount types, the Linux kernel also provides primitive APIs to manipulate refcounts. According to the developer manual [1, 3, 7], three categories of primitive APIs exist: `SET`, `INC`, and `DEC`. A `SET` primitive API initializes the refcount of a newly allocated object to 1. An `INC` primitive API increases the refcount by 1 when a new reference is assigned to the counted object, whereas a `DEC` primitive API decreases the refcount by 1 when a reference to the object becomes invalid. Note that although `INC` and `DEC` APIs allow to add or sub any value to the refcount, the Linux community recommends that the value should be changed by 1 in the context of refcounting [5].

There are a number of primitive refcount APIs; we collect 62 primitive refcount APIs from the latest Linux kernel and present some examples in Table 1. Further, with the help of the primitive APIs, Linux developers usually implement custom `INC` and `INC` wrapper functions to ease the management of various objects. The convention is that an `INC` wrapper function increments the refcount of an allocated object or allocates one if it has not been allocated, while a `DEC` wrapper function not only decrements the refcount but also releases the object if its counter drops to 0.

Table 1: Primitive refcount APIs in the Linux kernel.

Category	#	Examples
SET	5	atomic_set, refcount_set, kref_init
INC	27	atomic_inc, refcount_inc, kref_get atomic_add, refcount_add_not_zero
DEC	30	atomic_dec, refcount_dec, kref_put atomic_sub, refcount_sub_and_test

Table 2: The number of fields that are defined with the 5 refcount data types.

Refcount Type	# of Fields
atomic_t	2,010
atomic_long_t	154
atomic64_t	334
refcount_t	297
kref	425
Total	3,220

2.2 A Study on Refcount Field Identification

The wide use of the refcounting mechanism in various kernel modules, together with the general-purpose data types that are used to define refcount fields, makes the identification of refcount fields quite challenging. In order to understand the difficulties in identifying refcount fields in the Linux kernel, we perform a study on Linux 5.6-rc2.

First, to explore the possibility of manual identification for refcount fields, we write a simple LLVM-based analyzer to collect all fields that are defined in the 5 refcount data types from the whole Linux kernel. As shown in Table 2, the total number of potential refcount fields is 3,220. Since 77.6% of the fields in Table 2 (i.e., 2,498 = 2,010+154+334) belong to `atomic_t`, `atomic_long_t` or `atomic64_t` which may be used for other purposes, we can not simply flag them as refcount fields.

Second, to understand how many of these general types actually act as refcounts, we perform a further investigation. Specifically, we randomly select 300 fields from Table 2, covering all the 5 refcount types. Since `atomic_t` dominates the distribution among the 5 types in Table 2, we select 100 fields in this type. For the remaining 4 types, we select 50 fields in each.

In order to have a clear understanding of the real purposes for the selected 300 fields, we manually dig them out. During the investigation, two authors carefully examined their usage, with the help of the commit messages, code comments, and the code that manipulates these fields. Among all the fields, 71 of them can be directly labelled with commit messages; 110 of them are labelled with the help of code comments, while the usage of the remaining ones has to be inferred from code behaviors. Overall, the process cost about 100 man-hours.

Table 3: The usage for the selected 300 fields.

	Lock/ Status	Token/ ID	Normal Counter	Refcount
atomic_t	16	2	45	37
atomic_long_t	2	1	42	5
atomic64_t	0	13	34	3
refcount_t	0	0	0	50
kref	0	0	0	50
Total	18	16	121	145

During the manual analysis, we mainly observe four usages for these fields, and the detailed results are presented in Table 3. From this table, we find that *normal counter* and *refcount* contribute for the most cases of usage. Not surprisingly, all cases with `refcount_t` and `kref` types are recognized as refcounts, consistent with their specific purposes in reference counting. Nevertheless, we observe that the three general atomic types are used for more than one purposes. Take `atomic_t` type as an example, 16 cases act as lock/resource status, 2 cases are used as token/ID, 45 cases are used as normal counters, while actually only 37 cases of them are used as refcounts.

The above results clearly indicate that it is unacceptable to simply recognize refcount fields through their data types. Meanwhile, it would be impractical to manually identify refcount fields from such a large quantity.

3 Two-Dimensional Consistency Checking

This section uses several real-world refcount bugs (reported by CID) to illustrate the motivation and the approach of our two-dimensional consistency checking.

Dimension 1: INC-DEC Consistency Checking. The most intuitive approach to detecting refcount bugs is to statically trace all paths to check if a `DEC` operation is paired to an `INC` operation. This does not work well in practice because complicated conditions and data flows are involved along the paths. We observe that `INC` operations and `DEC` operations enforce a strict mutual relation with conditions. This observation motivates us to examine the consistency between conditional `INC` operations and the corresponding conditional `DEC` operations. Our insight is that the `INC` operations, which are often in a callee function, and the `DEC` operations, which are often in a caller function, should follow the same refcounting conventions. Based on how the caller conditionally performs the `DEC` operations, we can infer a set of condition-aware refcount rules for the callee, and violating cases are refcount bugs. Likewise, we can also infer the condition-aware refcount rules for the caller based on the callee. Such a design focuses on the two ends of refcount operations, and the rules always apply

```
1 /* File: net/batman-adv/hard-interface.c */
2 struct batadv_hard_iface* batadv_hardif_get_by_netdev(...)
3 {
4    struct batadv_hard_iface *hard_iface;
5    ...
6    list_for_each_entry_rcu(hard_iface, ...) {
7       if (hard_iface->net_dev == net_dev && ...
8           // increase refcount if find the hard_iface
9           kref_get_unless_zero(&hard_iface->refcount))
10          goto out;
11   }
12   hard_iface = NULL;
13   out:
14   ...
15   // return the hard_inface if found
16   return hard_iface;
17 }
```

(a) INC Function

```
1 /* File: net/batman-adv/sysfs.c */
2 static ssize_t batadv_store_throughput_override(...){
3    ...
4    //call INC function
5    hard_iface = batadv_hardif_get_by_netdev(net_dev);
6    if (!hard_iface)
7       return -EINVAL;
8    ...
9    ret = batadv_parse_throughput(...);
10   if (!ret)
11      //missing refcount decrease here
12      return count;
13   ...
14   //decrease refcount before return
15   batadv_hardif_put(hard_iface);
16   return count;
17 }
```

(b) Caller with Buggy DEC Operations

Figure 1: An Example to Illustrate INC-DEC Consistency Checking.

```
1 /* /drivers/usb/core/urb.c */
2 void usb_kill_anchored_urbs(struct usb_anchor *anchor)
3 {
4    ...
5    while (!list_empty(&anchor->urb_list)) {
6       victim = list_entry(anchor->urb_list.prev, ...);
7       //increase the refcount
8       usb_get_urb(victim);
9       ...
10      //decrease the refcount
11      usb_put_urb(victim);
12   }
13   ...
14 }
```

(a) Caller with Correct DEC Operations

```
1 /* /drivers/usb/host/ehci-hub.c */
2 static int ehset_single_step_set_feature(...)
3 {
4    ...
5    urb = request_single_step_set_feature_urb(...);
6    ...
7    //increase the refcount
8    usb_get_urb(urb);
9    ...
10   //decrease the refcount
11   usb_put_urb(urb);
12   ...
13   return retval;
14 }
```

(b) Caller with Correct DEC Operations

```
1 /* /drivers/net/wimax/i2400m/usb-fw.c */
2 ssize_t i2400mu_bus_bm_wait_for_ack(...)
3 {
4    ...
5    usb_init_urb(&notif_urb);
6    //increase the refcount
7    usb_get_urb(&notif_urb);
8    ...
9    //miss refcount decrease before return
10   return result;
11 }
```

(c) Caller with Buggy DEC Operations

Figure 2: An Example to Illustrate DEC-DEC Consistency Checking.

no matter how complicated the refcounting paths (between the two refcount operations) are.

To illustrate the rationale behind the checking, we give an example in Figure 1. Figure 1(a) presents an INC function `batadv_hardif_get_by_netdev()`. This function finds the `hard_iface` object that owns the given `net_dev` object, and if the object is found it returns the `hard_iface` object with its refcount increased (line 9); otherwise, it returns `NULL` without changing refcout. Therefore, we infer the rules. *Rule 1:* if `batadv_hardif_get_by_netdev()` succeeds, i.e., returns a non-error, its caller should decrement the refcount. *Rule 2:* if `batadv_hardif_get_by_netdev()` fails, i.e., returns an error, its caller should *not* decrement the refcount;

Figure 1(b) presents a caller function that invokes the `INC` function, and the `DEC` operation for the `hard_iface` object uses function `batadv_hardif_put()`. Now we apply the aforementioned two inferred rules to check if the caller function correctly operates the refcount. Specifically, rule 2 is honored—when the callee fails, the caller directly returns in line 7 without decreasing the refcount. However, rule 1 is not honored. All the code paths from line 8 correspond to the case in which the callee succeeds. Therefore, the refcount should be decreased in all these paths based on rule 1. A missing decrease refcount bug however occurs because the path ending in line 12 does not decrease the refcount.

Dimension 2: DEC-DEC Consistency Checking. The `DEC-DEC` consistency checking is based on an observation that `INC` and `DEC` functions for the same object are often invoked multiple times, following the same usage, and the usage is bug-free in most cases. Therefore, we can leverage statistical analysis on the multiple `DEC` operations paired with the same`INC` operation, and perform consistency checking to identify the deviating `DEC` operations from the majority `DEC` operations as potential refcount bugs.

Figure 2 shows an example to explain how `DEC-DEC` consistency checking works. In this example, `usb_get_urb()`/`usb_put_urb()` are `INC`/`DEC` functions for the refcount of an urb object. There are three caller functions in Figure 2 which all invoke the `INC` function to increase the refcount of an urb object. Among the three callers, both Figure 2(a) and Figure 2(b) invokes the `DEC` function before return, while only Figure 2(c) does not perform `DEC` operation when return. Since the majority of callers has consistent `DEC` behaviors, we recognize Figure 2(a)/(b) as correct callers, while reporting Figure 2(c) as a buggy caller.

Relation of the Two Dimensions. Note that the two dimensions of consistency checking are applied under different scenarios, thus are complementary. For example, `INC-DEC` consistency checking can not apply to Figure 2, because there is no return value from its `INC` function (`usb_get_urb()`). Similarly, the bug in Figure 1 can not be detected with `DEC-DEC` consistency checking, since there are not enough callers for the same `INC` function (`batadv_hardif_get_by_netdev()`) for statistical analysis. Therefore, the two dimensions exploit the consistencies between the `INC` and the `DEC` operations, and they further complement each other to detect more bugs than either.

4 Design

This section presents the workflow of CID and describes the design of its major components.

4.1 Workflow Overview

Figure 3 presents the workflow of CID. It takes LLVM bitcode files as input and automatically reports refcount bugs. There are mainly three phases in the bug detection.

Phase 1: Behavior-based Refcount Field Identification. CID identifies refcount fields from all candidate fields which are defined in the 5 refcount data types as described in Table 2. The identification employs the novel behavior-based inference which is presented in §4.2.

Phase 2: Path-sensitive Refcount Operation Analysis. As explained in §3, the two-dimensional consistency checking relies on path condition analysis on both `INC` operations and `DEC` operations. In order to realize the two-dimensional checking, CID performs a precise path-sensitive refcount operation analysis against both `INC` functions and their callers. CID also performs reference escape analysis on the object to exclude the reference-escaped paths in callers from the analysis scope, which reduces false positives.

Phase 3: Bug Detection with Two-Dimensional Consistency Checking. Based on the results of refcount operation analysis, CID detects refcount bugs from two dimensions: `INC-DEC` consistency checking and `DEC-DEC` consistency checking.

4.2 Refcount Field Identification

There are in total 5 atomic data types (`atomic_t`, `atomic_long_t`, `atomic64_t`, `refcount_t` and `kref`) that can be used to define refcount fields. CID first uses static analysis to collect all the fields of the kernel data structures that contain any fields of these types and marks them as candidate refcount fields. As described in Table 2, CID identifies 3,220 candidate fields. However, as shown in §2.2, many of the candidate fields are not true refcount fields, thus requiring further analysis. Manually analyzing them is tedious and impractical.

Unique Behaviors of Refcount Fields. In order to identify true refcount fields from the candidate fields, we aim to analyze and profile unique behaviors of refcount fields. Therefore, we manually analyzed 300 candidate fields (as introduced in §2.2) to profile inherent behaviors of them. Fortunately, we indeed observe three unique behaviors that significantly differ refcount fields from others. First, refcount fields are usually initialized with `SET` operations and thereafter incremented/decremented with `INC`/`DEC` operations, while other-purpose fields may not be manipulated by all three kinds of operations (e.g., *lock/status* fields may not be operated by `INC`/`DEC`). Second, refcount fields are `SET` to 1 at initialization, while other-purpose fields may be set to other values (e.g., *token/ID* fields, *normal counter* fields). Third, we find refcount fields are more-likely incremented/decremented by 1 than other fields, while are less-likely incremented/decremented by other numbers (though sometimes exists). We summarize these observed behaviors as follows.

- **Rule 1 (R1)**: The operations on the field should cover all three categories of primitive APIs: `SET`, `INC`, and `DEC`.
- **Rule 2 (R2)**: For each `SET` operation, it must set the refcount to 1.
- **Rule 3 (R3)**: For the `INC` and `DEC` operations, it should include at least one increase and one decrease of the refcount by 1.

Behavior-based Inference for Refcount Fields. Based on our observation of the unique refcount behaviors, we propose *behavior-based inference* to identify the refcount fields. Our approach abstracts the behavior of a primitive refcount API as <op_type, op_value>, where op_type represents the type of the operation (including `SET`, `INC` and `DEC`), and op_value represents the value that the operation manipulated on this field. For example, the function call `refcount_set(obj->candidate_field, 1)` is summarized as <`SET`, 1>. Given the 62 manually-collected primitive refcount APIs (as introduced in §2.1) and the 5 atomic data types in Table 2, CID first identifies all the primitive API calls that manipulate the candidate fields, and then summarizes the behaviors of these callsites. At last, with the behaviors of the candidate fields, CID employs the three rules to determine the real refcount fields.

Following the above way, CID automatically and systematically identifies all possible refcount fields from the large candidate field set. Though the approach is quite intuitive, to the best of our knowledge, CID is the first to identify refcount fields in an automated and systematical way. As evaluated in §6.6, it achieves promising performance in both precision and recall. This technique is not limited to detecting refcount bugs, but can also facilitate other works on refcount (e.g., refcount type conversion [36]).

4.3 Refcount Operation Analysis

Identify INC/DEC Operations (Functions). CID detects refcount bugs by checking the consistencies between the `INC` operations and the `DEC` operations. Therefore, CID needs to collect all the paired `INC` and `DEC` operations. This process consists of the following steps. (1) CID locates the `INC` functions which perform `INC` operations on the identified refcount fields with primitive APIs; (2) For each `INC` function, CID collects its callers through call graph analysis; (3) In each caller, CID recognizes `DEC` functions that operate on the same refcount to the corresponding `INC` function with alias analysis; (4) We find all the paired `DEC` operations for each `INC` operation.

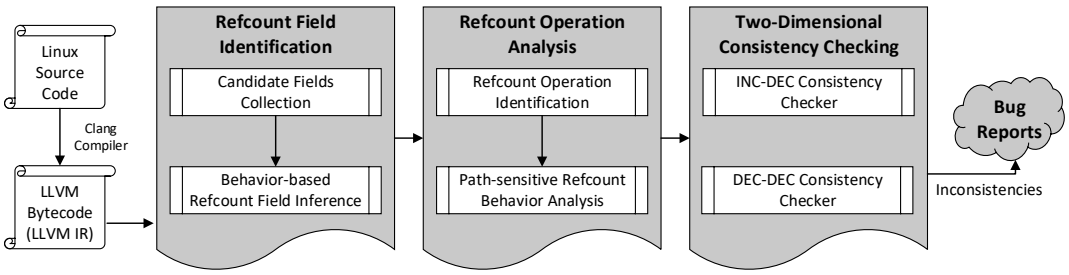

Figure 3: The overview of CID. It first identifies refcount fields (see §4.2), then performs refcount behavior analysis (see §4.3), and finally detects refcount bugs with two-dimensional consistency checking (see §4.4).

Collect INC/DEC Conditions. To be more precise, our two-dimensional consistency checking is *condition-aware*. That is, we check corresponding refcount operations based on conditions. Therefore, CID needs to collect the conditions for `INC` operations and the conditions for `DEC` operations. However, it is non-trivial to perform such analysis, since the kernel is quite complicated, and there are a lot of conditions in kernel functions while only a small of them affect the refcount operations. We observe that developers usually correlate the `INC` operations and the `DEC` operations through the return value of the `INC` functions. For example, in Figure 1, the condition is the return value of `batadv_hardif_get_by_netdev`, based on which its caller performs the corresponding refcount decrease. Therefore, CID preforms an intra-procedure path-sensitive analysis to collect the return value for each path in the `INC` function and the pre-condition for each path through the call to the `DEC` function in the caller. Since the analysis is only performed in a single function, CID could afford a path-sensitive analysis in the kernel. This way, we model the `INC` behaviors inside an `INC` function, and the `DEC` behaviors inside its callers.

Model INC Behaviors in an INC Function. After we collect the conditions, we model the INC operations and their conditions to facilitate the consistency checking. We define the tuple <*Action, RetVal*> to model the `INC` behaviors for a path in the `INC` function. In this tuple, *Action* can be `INC` or `EMPTY`, which depicts the refcount of the object is incremented or not. The *RetVal* represents the return value for this path, and it is marked as `VOID` if no return value. CID performs a path-sensitive data flow analysis in the `INC` function to collect the *Action* and *RetVal* for each path. During the analysis, infeasible paths are identified by checking contradictory path constraints (explained in §5) and eliminated from the modeling.

Model DEC Behaviors in Each Caller. CID also models the `DEC` behaviors in each caller of an `INC` function. Similarly, CID uses a tuple <*Action, Conditions*> to represent the `DEC` behaviors in this caller. The *Action* have three possible values: (1) `DEC` which means a paired `DEC` operation is performed on the same object; (2) `ESCAPE` which means there is no `DEC` operation, but the object escapes from the caller; (3) `EMPTY` which means neither the `DEC` operation nor the reference escape happens in the caller. CID again performs a path-sensitive data flow analysis in the caller to collect these *Action*s. The paths that have reference escapes are excluded from the consistency checking in §4.4, because the `DEC` behaviors of the escaped object is out of the analysis scope. During the analysis, CID also collects the constraints (i.e., `if` statements) against the return value of the `INC` function as *Conditions*.

4.4 Consistency Checking

Based on the modeling of `INC` behaviors in the `INC` functions and the `DEC` behaviors in the corresponding caller functions, CID checks consistencies to detect refcount bugs in two dimensions.

INC-DEC Consistency Checking. The `INC-DEC` consistency checker examines whether the `INC` function and the `DEC` function respect each other's refcount operations under the same conditions. The checker is mutual—from the `INC` function, it infers the context-aware refcount rules for the `DEC` function, and vice versa. For simplicity, we choose the inference based on the `INC` function to illustrate how the checker works.

Given an `INC` function and its modeled behaviors, the checker looks into each path and summarizes: (1) under what conditions (e.g., returning an error code or success code), it performs `INC`; (2) under what conditions, it also performs `DEC`; for the paths that have the same refcount behaviors, the checker unifies their conditions as $RetVal_1|RetVal_2$. After that, the checker infers the condition-aware rules based on the refcount convention, i.e., the caller should perform the opposite `DEC` operation under the consistent conditions. The condition-aware rules are expressed in the form of <*Action, RetVal*>, specifying under what conditions, the caller of the `INC` function should perform what refcount operations.

More specifically, the checker takes two inputs: the behaviors of an `INC` function and the behaviors of one caller function that invokes this `INC` function. The checker then works as follows. First, in the `INC` function, it selects its paths. For each path, the checker computes both the refcount operations as well as the post-condition (i.e., returning an error or

not). Second, it merges the results for the paths in a form of <*Action, RetVal*>. Note that if a path performs both `INC` and `DEC`, the merged action will be `EMPTY`. Third, it generates the refcount rules for the `DEC` function: <*–Action, RetVal*> where "–" denotes an opposite action. Fourth, using the rules and the modeled behaviors of a caller of the `INC` function, which are in the form of <*Action, Conditions*>, the checker detects violating cases as refcount bugs.

DEC-DEC Consistency Checking. For each `INC` function, the `DEC-DEC` consistency checker first summarizes the `DEC` behaviors for its each caller and then identifies deviating cases across all callers through statistical analysis. The deviating cases are identified as potential refcount bugs because in general most callers are correct.

The `DEC` behaviors of a caller function are summarized from all the paths starting from the return of the `INC` function. We call the summarized `DEC` behaviors of a caller as its tendency. There are three possible values for the tendency of an caller: (1) *EMPTY* which means all paths in the caller do not perform any `DEC` operations; (2) *DEC* which means all the paths perform `DEC` operations or conditional `DEC` operations that depends on the return value of the corresponding `INC` function; (3) *UNKNOWN* which is used in the remaining scenarios when we are unable to make the decision.

To measure the inconsistency of the tendency among all callers of an `INC` function, we define *inconsistency score*. CID uses three steps to calculate the *inconsistency score* across all callers of an `INC` function. First, the checker counts the number of callers for each tendency, and represents them as a normalized three-dimensional vector.

$$x = num\ of\ callers\ implies\ EMPTY$$
$$y = num\ of\ callers\ implies\ DEC$$
$$z = num\ of\ callers\ implies\ UNKNOWN$$
$$length = \sqrt{x^2+y^2+z^2}$$
$$vector = (\frac{x}{length}, \frac{y}{length}, \frac{z}{length}) = (x_{nv}, y_{nv}, z_{nv})$$

Second, the checker separately calculates the distances between the normalized vector and the three base vectors as follows.

$$distance_x = \sqrt{(x_{nv}-1)^2+y_{nv}^2+z_{nv}^2}$$
$$distance_y = \sqrt{x_{nv}^2+(y_{nv}-1)^2+z_{nv}^2}$$
$$distance_z = \sqrt{x_{nv}^2+y_{nv}^2+(z_{nv}-1)^2}$$

Finally, the checker gets the main tendency among the callers by comparing their distances and defines the *inconsistency score* as:

$$inconsistency\ score = \min(distance_x, distance_y, distance_z)$$

The rationale behind the *inconsistency score* is that it measures the uniformity of the `DEC` behaviors among all the callers for an `INC` function. If the score is zero, it means all callers has the same `DEC` behaviors. The high the score is, the more diverse that these callers behaves. When most of the callers tend to perform `DEC` operations, the checker marks the callers which deviate the main tendency as potential refcount bugs.

5 Implementation

We have implemented CID as multiple passes on top of LLVM (version 10.0.0), including a pass for constructing call graph, a pass for identifying reference escape, a pass for performing data flow analysis and alias analysis, and a pass for detecting and reporting potential refcount bugs. The alias analysis is based on the LLVM alias analysis infrastructure. The implementation of CID contains 10K lines of C++ code (counted by *cloc*). We present some interesting implementation details below.

Escape Analysis. As described in §4.3, CID needs to know whether an object reference escapes from the caller. Thus, CID performs reference escape analysis on an refcount-tracked object in the caller. We consider three common reference escape scenarios: (1) the referenced object may escape to an argument pointer of the caller function; (2) the referenced object may escape to a global variable; (3) the referenced object may escape to the return value of the caller function. CID tracks the use of the object within the caller through def-use analysis and alias analysis. During the analysis, CID carefully inspects each use point of the reference to test if an escape occurs.

However, since we perform intra-procedural data flow analysis, we may miss data flows through function calls, resulting in false negatives in escape analysis. For example, if a caller invokes a function to acquire a field of a global struct and then assigns the referenced object to this field, a reference escape occurs while we can not capture. To eliminate such false negatives, we perform a conservative one-layer inter-procedural analysis to generate data flow summaries for invoked functions. Specifically, we only capture the direct data flows from the arguments of an invoked function to its return value without considering other complicated situations, such as pointer alias, function calls. Note that this conservative approach may cause false positives in reference escape analysis, but it may only generate some false negatives in bug detection. More importantly, this design makes CID scale to the whole-kernel analysis.

Identify Contradictory Path Constraints. In §4.3, CID excludes infeasible paths in the `INC` function. A common practice in infeasible path elimination is to check the satisfiability of the path's constraints with the help of an SMT solver. However, this method is expensive. We observe that the unsatisfiability of a path is frequently caused by two obvious contradictory constraints on the same expression. For example, there is one constraint requires an expression being true while the other one says the expression must be false. Hence,

we implement a light-weight approach to identify such contradictory constraints: first, CID collects the path constraints for each path with data flow analysis; second, CID groups the constraints for the same expression; third, CID checks if there are contradictory constraints in each group; finally, CID reports infeasible path if there is any contradictory constraint group. In this way, CID can efficiently identify and eliminate infeasible paths.

Bug Reporting and Ranking. CID generates detailed bug reports to ease bug confirmation. Because CID checks bugs from two different dimensions, it outputs reports in two different formats.

The *INC-DEC Consistency Checker* examines the inconsistency between the conditional `INC` operations and corresponding conditional `DEC` operations. For each reported bug, it outputs the name of the `INC` function, the name of the inconsistent caller, and the inconsistent path pair. The analysts can easily confirm the bug with such information.

The *DEC-DEC Consistency Checker* identifies deviating callers from the majority. In order to reduce the burden of manual verification, CID ranks the reports based on the *inconsistency score* and prunes these reports with a threshold (θ). Therefore, the remaining bug report set may have a higher true positive rate. For each reported bug, the checker outputs the name of the `INC` function, the names of the deviated callers which may have bugs, the *inconsistency score* and suggests the appropriate refcount operation learned from the majority.

6 Evaluation

This section applies CID to the Linux kernel to evaluate its effectiveness in refcount bug detection and refcount field identification.

6.1 Setup and Configuration

The experiments are performed on a Debian 8.11 (64-bit) machine with LLVM 10.0.0 installed (git commit: *771899e94452*). The machine has 128 GB memory and two Intel Xeon E7-4830 v2 processors (2.20 GHz, 20 cores). We compiled the source code of the Linux kernel version 5.6-rc2 (git commit: *11a48a5a18c6*, released on Feb 16, 2020) with *allyesconfig* to enable all kernel modules for the x86_64 architecture. At last, 18,868 LLVM IR bitcode files are generated and used as the input of CID for evaluation.

Hyper-parameter Determination. As described in §5, bug detection from `DEC-DEC` dimension requires a hyper-parameter—threshold (θ) of the *inconsistency score* among all the callers of an `INC` function. The higher of the θ, the more bugs would be reported by CID, but the higher false positive rate CID may have. By trying several values for θ, we count the bugs reported by `DEC-DEC` consistency checking in Table 4. From this table, we find the reported bugs increased by 33 when θ increases from 0.4 to 0.5, while only 10 more bugs are reported when increasing θ from 0.5 to 0.6. Therefore, to control the volume of reported bugs, we choose θ = 0.5 for the following bug confirmation.

Table 4: Evaluating the hyper-parameter value, θ, among multiple choices.

Threshold (θ)	Reported Bugs
0.1	18
0.2	55
0.3	67
0.4	86
0.5	119
0.6	129

6.2 Bugs Reported by CID

By applying CID on Linux 5.6-rc2, CID identifies 792 refcount fields (details explained in §6.6) and reports 149 bugs. We manually analyzed all the reported bugs and confirmed 44 new refcount bugs. The details of the confirmed 44 bugs are presented in Table 8 (in Appendix A). Among all the bugs, `DEC-DEC` consistency checking reports 119 ones and 35 of them were confirmed; `INC-DEC` consistency checking reports 102 potential bugs from which we confirmed 27 real bugs. Based on the bug root cause, we wrote 42 security patches to fix these bugs and submitted them to the Linux community. Until now, 36 bugs have been confirmed by the Linux community, and the patches for 34 bugs have already been applied to the Linux mainline.

Bug Confirmation. CID relies on static analysis to detect bugs, which are known to have false positives, so manual confirmation is necessary. To ease the bug confirmation and the patch development, CID also outputs intermediate information (such as the `INC` function callsites, the detected `DEC` operation set in `DEC-DEC` consistency checking, the reference escape flag) for all the reported bugs. During the bug confirmation, we take the output information of CID as reference and manually check the inconsistency of the refcount operations in the reported buggy function. To be specific, we first observe whether the refcount behavior is operated just as the behavior tuple reported by CID. This step is to ensure the inconsistency is not caused by CID misidentifying or missing the `DEC` operation. Second, we check if the inconsistency state is caused by some special code logic, such as synchronization mechanism (e.g., completions), indirect function call (e.g., file open and close), which are known to be too difficult to handle in our current implementation. If we observe that those situations happen in the buggy function, we would like to conservatively exclude it from bugs. Otherwise, we deem it a real refcount bug and report it later. Following the above process, we manually analyzed 149 bug reports and confirmed 44 new bugs. The whole process took us 37 man-hours, which we believe is affordable.

Efficiency. CID completes the analysis of the whole kernel within 18 minutes, of which loading bitcode files and constructing call graph take 7 minutes, refcount field/operation identification costs about 1 minute, and refcount operation analysis together with bug checking cost 10 minutes. The analysis covered 19.2 million lines of code (reported by the tool *cloc*) for the Linux kernel. According to the results, we confirm that CID is quite efficient to scale to the highly complex whole-kernel analysis.

6.3 False Positives Breakdown

Among the 149 bugs reported by CID, we manually confirm 44 of them as real refcount bugs and the left of them are false positives. We analyze the 105 FPs and summarize three causes for them.

- *Imprecise escape analysis (34 FPs).* CID leverages the escape information collected through escape analysis to perform two-dimensional consistency checking. However, in addition to the situations we discussed in §5, there are other complicated reference-escape situations which CID can not recognize. When the escape analysis exhibits a false negative, CID may wrongly expects a paired `DEC` operation in the current function, causing a false positive.
- *Imprecise alias analysis (23 FPs)*: In refcount operation analysis, CID attempts to identify the `DEC` operations on the incremented object via intra-procedural data-flow analysis and alias analysis. The intra-procedural analysis used by CID prevents it from finding some paired `DEC` operations that are performed on aliased object pointers. Therefore, CID incorrectly reports bugs.
- *Others (48 FPs)*: Other reasons relate to the special features of the Linux kernel, such as the heavy use of the function pointers to support polymorphism (i.e. indirect function call), synchronization mechanism (e.g., completions) and so on. Due to those reasons, sometimes CID can not locate the paired `DEC` operations and falsely reports refcount bugs. Besides, we found that some bugs are reported on infeasible paths. Since these bugs cannot be triggered, they also belong to false positives.

It turns out that most false positives are introduced by the inaccuracy of the static analysis instead of our bug detection oracle. In §7, we discuss how to mitigate these false positives by applying more precise analysis techniques.

6.4 Security Impacts of Reported Bugs

We manually examine the security impacts of the reported bugs and find that these bugs cause severe security impacts, including UAF, DoS, and memory leak. As presented in Table 8, we confirm 37 bugs that may cause DoS, 5 bugs that may result in UAF, and all of them may cause memory leak. Here we present two case studies to explain the security impact of the reported bugs.

```
1 static int comedi_open(struct inode *inode,struct file *file)
2 {
3     ...
4     // increase refcount if success
5     struct comedi_device *dev = comedi_dev_get_from_minor(minor);
6     ...
7     cfp = kzalloc(sizeof(*cfp), GFP_KERNEL);
8     if (!cfp)
9         // missing refcount decrease here
10        return -ENOMEM;
11    ...
12    if(rc) {
13        // Other error paths
14        comedi_dev_put(dev); // decrease the refcount
15        kfree(cfp);
16    }
17    ...
18}
```

Figure 4: A missing decrease refcount bug detected by CID in `comedi_open()`, which results in memory leak and DoS.

Case Study on Bug #11. In bug #11, there is a missing refcount decrease in one exceptional path of `comedi_open()`, and we find that it causes memory leak and DoS. We present the bug-related code in Figure 4. In this bug, `comedi_open()` first invokes `comedi_dev_get_from_minor()` (line 5) which returns a reference to the comedi device and increases its refcount. When `comedi_open()` returns zero, it means the open operation is successful. Otherwise, the open operation fails and the increased refcount at line 5 should be decreased. However, if we trigger a memory allocation failure at line 7, `comedi_open()` returns an error code (line 10) without decreasing the refcount to the comedi device. Therefore, the comedi device will not be freed, causing a memory leak. Furthermore, since the refcount for the `comedi_device` struct is defined with `kref`, which has overflow/underflow protections. Therefore, we can not continuously trigger this bug to cause a UAF. However, if an overflow on `kref` is detected by Linux kernel, the kernel will panic (aka. DoS), which is severe for a long-running system.

Case Study on Bug #21. Similarly, bug #21 is also a missing decrease bug in one exceptional path of `ext4_orphan_get()`. We find that this bug may lead to an exploitable UAF vulnerability. As shown in Figure 5, `ext4_orphan_get()` invokes `ext4_read_inode_bitmap()` (line 6) to return a reference to the buffer head object. If the invocation succeeds, it increases the refcount of the object, and the reference is hold by `bitmap_bh`. Otherwise, it returns an error code and does not touch the refcount of the buffer head object. When `ext4_orphan_get()` returns, the local variable `bitmap_bh` becomes invalid. Therefore, `ext4_orphan_get()` invokes `brelse` (line 17) to decrease the refcount of the the buffer head object. However, if the invocation to `ext4_iget()` (line 10) fails, `ext4_orphan_get()` directly returns without releasing `bitmap_bh` (line 14), causing a memory leak.

```
1 struct inode *ext4_orphan_get(struct super_block *sb, ...)
2 {
3    ...
4    struct buffer_head *bitmap_bh = NULL;
5    // increase refcount if success
6    bitmap_bh = ext4_read_inode_bitmap(sb, block_group);
7    if (IS_ERR(bitmap_bh))
8        return ERR_CAST(bitmap_bh);
9    ...
10   inode = ext4_iget(sb, ino, EXT4_IGET_NORMAL);
11   if (IS_ERR(inode)) {
12       // missing refcount decrease here
13       ...
14       return inode;
15   }
16   ...
17   brelse(bitmap_bh);  // decrease refcount
18   return inode;
19 }
```

Figure 5: A missing decrease refcount bug detected by CID in `ext4_orphan_get()`, which results in UAF.

To trigger a failure for the function call at line 10, we can prepare a specially-crafted file as the argument for `ext4_orphan_get()`. Even worse, the refcount field for the buffer head object is defined with *atomic_t* type, which is a 32-bit integer without overflow/underflow protection. Therefore, we could repeatedly trigger this bug to free the buffer head object while there are still valid references to this object. This bug would finally lead to an exploitable UAF vulnerability.

6.5 Comparison with Existing Tools

We measure the bug detection capability of CID by comparing it with the state-of-the-art approaches. Since RID [29] is the state-of-the-art and the most close work to ours which also employs consistency analysis, we choose it as our baseline. According to [29], the detection of RID is based on inconsistent path pair checking. The checking starts from its pre-defined refcount wrappers and analyzes their caller functions. For a caller function, RID collects the paths which are indistinguishable outside from its arguments and return values. If these paths incur inconsistent refcount change behaviors, RID reports it as a refcount bug.

The experiments are performed on 60 known refcount bugs that are fixed in the Linux kernel between 2018 and 2020. The known bug set is collected from the Linux source code repository by using regular expressions to search the keywords in the Git commit messages. The keywords include "refcnt", "refcount" and "reference count". We manually examine the matched 821 commits and finally locate 60 known bugs (as listed in Table 9). Note that race-induced refcount bugs are excluded, because they are essentially race bugs.

Comparison Results. Since RID is not open-sourced, we check its capability on detecting known bugs by carefully following its approach with manual efforts. We assume that the implementation of RID perfectly aligns with its design and even that it has the same ability in refcount field identification as CID. We present the detection results of CID and RID in Table 5. From this table, we find that CID detects 54 (=46+8) bugs while RID only detects 10 (=2+8) bugs. In all, there are 46 bugs that can only be detected by CID, while CID only misses 2 bugs that are detected by RID.

Table 5: Comparison between CID and RID on Detecting 60 Known Refcount Bugs.

Total	Reported by CID Only	Reported by both CID and RID	Reported by RID Only
60	46	8	2

Bugs Missed by RID. The missed 50 bugs by RID are caused by two reasons. First, 41 of them do not meet the requirement of IPP (inconsistent path pair) while RID only captures the inconsistent refcount change behaviors on IPP. Second, refcount primitive APIs are used in the left 9 bugs instead of refcount wrappers, which are out of the analysis scope of RID.

Bugs Missed by CID. For the 6 bugs that are missed by CID, we conclude three causes. First, 3 bugs are missed due to the implicit control flow (e.g., queue work mechanism, indirect function call) between the `INC`/`DEC` functions and their callers. Second, 1 bug is missed because developers do not use refcount primitive APIs (e.g., they directly use *refcount* ++) to manipulate refcount fields. Third, there are 2 bugs whose `INC` functions have no return value and have less than three callers. Therefore, neither `INC-DEC` nor `DEC-DEC` consistency checking detects these bugs. Note that the 2 bugs can be detected by RID because they meet the requirements of IPP.

After comparing CID with RID, we conclude that both RID and CID incur high FP rate (>70%) due to the imprecise static analysis, and their different bug detection strategies lead to discrepant bug detection capabilities. From the design, the two-dimensional consistency checking helps CID detect bugs in a wider scope. As a result, CID is able to detect significantly more bugs.

6.6 Evaluating Refcount Field Identification

A key contribution of CID is its systematical identification of refcount fields. We also evaluate the effectiveness of this part. In all, CID identifies 792 refcount fields from the Linux kernel. The detailed results are presented in Table 6. It is interesting to find that not all `refcount_t` and `kref` fields are refcount fields. We manually analyze 11 `refcount_t` fields and 18 `kref` fields that are identified as non-refcount fields by our tool. Our results confirm that 26 of them are indeed non-refcount fields, and only 3 of them are false negatives of our tool. We find developers wrongly use these non-refcount fields for normal counters (e.g. `packet_sock->sk_wmem_alloc`), and for lock/status (e.g., `device_link->rpm_active`). This finding further demonstrates the advantages of our behavior-

Table 6: The number of the refcount fields identified by CID.

Refcount Type	# of Fields	# of Refcount Fields	Ratio
atomic_t	2,010	140	6.97%
atomic_long_t	154	5	3.25%
atomic64_t	334	3	0.90%
refcount_t	297	251	84.5%
kref	425	393	92.5%
Total	3,220	792	24.6%

Table 7: The Effectiveness of Refcount Field Identification on Ground Truth (Hint: R1 requires a refcount field has all three types of primitive operations; R2 requires a refcount field should be set to 1 at each `SET` operation; and R3 requires that a refcount field at least has one <`INC`, 1> and <`DEC`, 1>).

Rule Setting	TP	TN	FP	FN	Accuracy	Precision	Recall
R1	143	104	51	2	82.3%	73.7%	98.6%
R2	137	134	21	8	90.3%	86.7%	94.5%
R3	145	61	94	0	68.7%	60.7%	100.0%
R1&R2	137	145	10	8	94.0%	93.2%	94.5%
R1&R3	143	117	38	2	86.7%	79.0%	98.6%
R2&R3	137	143	12	8	93.3%	91.9%	94.5%
R1&R2&R3	137	146	9	8	94.3%	93.8%	94.5%

based refcount field identification, which does not rely on the specific data types.

Effectiveness on Ground Truth. The effectiveness evaluation requires a ground truth set. Since in §2.2 we have manually labelled the usage for 300 fields (the results are presented in Table 3), we use this set to evaluate CID in refcount field identification. In all, our ground truth consists of 145 positive cases and 155 negative cases.

To identify refcount fields from all possible fields (those are defined in the 5 refcount data types), CID proposes a behavior-based inference approach. There are three rules in the inference. Our experiments evaluate the effectiveness of these rules and their combinations in identifying refcount fields. The detailed results are shown in Table 7. From this table, we find that the combination of all the rules (R1&R2&R3) achieves the best performance in accuracy, precision and recall. This finding supports our design of combining the three rules in CID.

Error Case Analysis. Following the rule setting of combining all the three rules (aka. R1&R2&R3), CID reports 9 false positives and 8 false negatives in identifying refcount fields. We present the causes for them below.

- *False Positives.* In all false positive cases, CID wrongly recognizes some plain counters as refcounts. For example, the `rxrpc_net->nr_calls` field is used for counting the number of RPC calls registered in the `rxrpc_net` struct, while CID identifies it as reference counter. The reason is that the manipulation APIs operated on this field match all the three rules. Therefore, CID reports a false positive case here. Actually, CID can be improved to eliminate this kind of false positives by considering the initialization behavior of the field at the allocation site. More specifically, we observe that for refcount fields its initialization is near to the allocation site of its tracked object, while there is not such observation for normal counters. Since the current performance of CID is acceptable, we leave this optimization as our future work. Besides, false positives in refcount field identification may not lead to false positive cases in the bug detection. Our breakdown of false positives in §6.3 also confirms this point. Interestingly, when using such fields, CID can still detect inconsistencies in using the fields, which still form true bugs although they are not refcount bugs.
- *False Negatives.* All false negatives cases are due to that the `SET` operations may not initialize the refcount values to 1. For example, the `rxrpc_connection->usage` field has two `SET` operations: one sets the `usage` field to 1 in `rxrpc_alloc_client_connection()`, and the other sets the field to 2 in `rxrpc_prealloc_service_connection()`. In the latter case, developers explicitly claim in the code comments that they need to initialize the refcount field to 2 because this object will be used in two places after allocation. This behavior violates the Rule 2 of CID. As a result, CID misses this case in refcount field identification. However, it is worth noting that such a behavior is not encouraged in the kernel documentation [3]. Therefore, we do not expect this is a normal and common behavior that should be handled by CID.

7 Discussion

The Impact of Reference Escape on the Analysis Scope. As described in §4, CID performs escape analysis to exclude the reference-escaped paths in callers from the analysis scope. Such design may limit the code that can be analyzed by CID. Therefore, we measure its impact on the analysis scope. First, in `DEC-DEC` checking, we will extend our analysis to its callers if we observe that a referenced object escapes the current function with refcount increased. In particular, we will treat this functions as a new `INC` function, and then analyze its callers to locate the paired `DEC` operations. For the 792 refcount fields, CID locates 11,910 caller functions (including extended ones) for `DEC-DEC` checking, while 3,751 functions (31.5%) still can not be analyzed due to two reasons: 1) we limits the extension in 3 layers; 2) they do not have enough callers for `DEC-DEC` consistency checking after extension. Second, in `INC-DEC` checking, the current implementation of CID does not extend the scope to the caller function if reference escapes. The reason is that the extended analysis requires

accurate inter-procedural data flow to capture the conditional INC/DEC operations. For the 5,146 caller functions identified by INC-DEC checking, 639 ones cannot be analyzed (12.4%) due to reference escaping. In the future, we plan to leverage inter-procedural data flow analysis to cover these cases.

Coordinating Two-dimensional Consistency Checking. The unique advantage of DEC-DEC consistency checking is that it does not require to understand the semantics in the INC function. However, its statistical analysis requires adequate callers for the inference. One the other hand, the INC-DEC consistency checking is not limited by the number of callers, but it needs accurate analysis about the INC function.

CID decides which consistency checking strategy to use according to the specific situation of the INC function and its callers. If the situation meets the requirements of both checking strategies, the refcount operations will be checked from both dimensions. As a result, CID takes the advantage of both DEC-DEC consistency checking and INC-DEC consistency checking to effectively uncover refcount bugs and cover more codes.

Mitigating False Positives. To mitigate false positives, more advanced static analysis techniques can be adopted. First, we could use inter-procedural data-flow analysis to improve the accuracy of escape analysis and alias analysis. Second, the multi-layer type analysis [25] can be leveraged to precisely identify the targets of indirect calls in the kernel. Such information can help CID reduce the false positives due to missing paired DEC operations. Last but not least, symbolic execution [33, 34] would also help CID identify and eliminate the bug reports which can not actually be triggered.

Bug Exploitability. As a static analysis-based detection tool, CID excludes the generation of PoC or an exploit from the scope. Two general exploitation strategies are (1) to increase the refcounts, so as to maliciously consume resources, which finally leads to DoS and (2) to force the refcount to reach zero, through either over decrease or overflow, so as to trigger the release of refcount object, which often leads to use-after-free. As such, in general the exploitation of refcount bugs requires one extra primitive—repeatedly triggering the bug. Once that, the exploitation is already successful or can further reuse existing use-after-free exploitation techniques [14, 21, 40]. Actually, exploring whether a bug can be triggered is an orthogonal and extremely challenging problem. Therefore, we leave it as our future work. Specifically, we plan to combine directed fuzzing [11, 13] and concolic execution [42] to evaluate the triggerability of detected bugs.

Portability. The only prior knowledge CID requires is a list of primitive refcount types such as the ones shown in Table 3. Once the list is provided, CID can automatically identify refcount fields and perform the bug detection. We find that other OS kernels and user-space programs also widely use primitive refcount types and APIs to implement refcount mechanisms. Take FreeBSD as an example, data types like `reference_t`, `zfs_refcount_t` are used to define refcount fields; the operations on these fields are also encapsulated into primitive APIs (e.g., `refcount_init()`, `refcount_acquire()`, `refcount_release()`). Similarly, in Mozilla Firefox (written in C++), its refcounted data structures should inherit certain base classes such as `RefCounted` or `RefCountType`, and two primitive APIs (`addref()` and `release()`) are provided to perform INC and DEC operations. Therefore, CID can be applied in these platforms for refcount field identification and refcount bug detection.

Implementation Improvements. The implementation of CID can be improved from two perspectives: *parallelized analysis* and *targeted analysis*. First, though its first three passes (call graph analysis, data flow analysis and alias analysis) are hard to parallelize due to their algorithmic nature, the final pass for bug detection can be parallelized with multi-threading. Specifically, either the modeling of INC/DEC operations or the two consistency checkers can be performed in parallel. Second, CID can be enhanced to support the targeted analysis. In this scenario, the developers can provide their interested refcount fields or interested functions that have refcount behaviors. With this information, CID can be configured to only check the refcount operations on the interested fields or refcount operations in the interested functions.

8 Related Work

Reference Counting. Due to the lack of automatic garbage collection, Use-after-free (UAF) and double-free vulnerabilities are quite common in C/C++ programs. Since reference counting is quite effective in managing dynamically-allocated objects/resources, many attempts [10, 19] have been made to provide C/C++ developers with reference counting mechanism. For legacy C/C++ applications, Shin et al. proposes CRCount [37] which leverages pointer footprinting to accurately compute the reference counts with a small runtime overhead. Though CRCount releases developers from the complicated management of objects/references, its performance overhead prevents its wide application. For programs that still rely on manual manipulation of refcount operations (e.g., Linux kernel, Mozilla Firefox), CID helps to detect refcount bugs.

Refcount Bug Detection. Due to the importance of refcount, several works have attempted to detect them. Software developers and testers have implemented refcount tracing and balancing techniques [6] to track leak of refcounted objects dynamically in FireFox. The coverage of dynamical testing is limited by its inputs. The researchers thus prefer to detect refcount bugs through static analysis or symbolic execution. Referee [17] uses symbolic model checking to find the refcount errors in the presence of multiple threads. The checking needs a complete control flow about the program under analysis, and it further assumes that the resources/objects is managed in the same way. Due to this assumption, it is impossible to be applied to OS kernels. Pungi [24] performs refcount bug detection in the native implementations of

Python/C programs with a strong property that the change of a refcount must equal the number of references escaped from the function. However, it is hard to apply to the OS kernels due to two reasons: 1) it requires accurate inter-procedural escape-analysis which is hard to realize in the kernel; 2) its detection strategy doesn't fit the kernel design where many functions (e.g., wrappers) can increase/decrease the refcount without reference-escaping. RID [29] proposes inconsistent path pair (abbreviated as IPP) checking to detect refcount bugs. The IPP checking identifies the refcount inconsistency between the paths which are indistinguishable outside the target function by examining their arguments and the return value. However, its detection scope is very narrow, as shown in our evaluation—only detecting 10 out of 60 refcount bugs reported between 2018 and 2020 from the Linux kernel.

Consistency Checking. Engler et al. [18] were among the first to explore the idea of statistical analysis. Though the approach is unsound, it is widely adopted by researchers to detect different kinds of bugs. Juxta [31] applies cross-checking to detect semantic bugs between semantically equivalent implementations of file systems. Yamaguchi et al. [41] infers search patterns for taint-style vulnerabilities through clustering the sink patterns. APISan [43] aims to find deviations from majority in API usages under rich symbolic contexts. CRIX [26] cross-checks the semantics of conditional statements in the peer slices of critical variables to compare their criticalness. RoleCast [38] also applies consistency checking to detect role-specific missing checks in Web applications. CID differs from all existing works from two perspectives. First, CID is the first to apply cross checking in refcount bug detection. Refcount bug detection is much more complicated by its nature, which requires the identification of refcount fields and operations. Second, Many refcount functions are called only in a limited number of times, rendering cross checking ineffective. CID incorporates the `INC-DEC` consistency checking, which requires only one occurrence, to address this problem.

Static Analysis in Kernels. Since more and more operating systems are open-sourced (e.g., Linux, FreeBSD), static analysis technique is widely adopted in detecting many kinds of security bugs in the kernel. Firstly, source code-based static analysis tools such as Smatch [8], Sparse [9] and Coccinelle [32] are frequently used in the Linux kernel for source code analysis and manipulation. However, these tools are not suitable for implementing CID. Take Coccinelle as an example, it is not used to build CID for two reasons: 1) our detection leverages the correlation between different operations across functions instead of capturing a specific pattern in one function; 2) our approach relies on more heavy-weight data-flow analysis such as reference-escape analysis, path-constraint analysis which is hard to implement in Coccinelle scripts.

Secondly, intermediate code-based analysis is preferred by several recent works. K-Miner [20] partitions the kernel code along separate execution paths starting from system-call entry points to allow practical inter-procedural data-flow analysis. Dr.Checker [28] focuses on the Linux kernel drivers and improves the precision of data flow analysis by sacrificing soundness in a few cases. Both K-Miner [20] and Dr.Checker [28] aim to improve practicality and precision of data flow analysis in kernel and serve as general bug detection tools. Meanwhile, there are some detection tools designed for detecting a specific kind of bugs in kernel. UniSan [27] detects information leaks caused by uninitialized reads. KINT [39] detects integer errors. Other complementary approaches to static analysis use symbolic execution [15, 34]. In comparison, CID leverages precise path-sensitive intra-procedural analysis to perform refcount bug detection instead of performing complex inter-procedural analysis in the kernel. CID also employs tailored techniques to identify refcount-related fields and operations.

9 Conclusion

Refcount bugs are quite common in the Linux kernel and cause critical security impacts. This paper presented CID, a scalable and effective system for refcount bug detection using a two-dimensional consistency checking. CID models all refcount behaviors. In one dimension, it infers condition-aware rules for detecting refcount bugs, and in the other dimension, it detects deviating `DEC` behaviors across refcount callers. This design helps CID avoid complicated semantic understanding or reasoning on refcount operations, and to cover more bugs than the state-of-the-art tools. Furthermore, considering refcount operations are diversely spanned in the whole kernel, CID introduces behavior-based inference to systematically identify refcount fields and the operations. By applying CID to the Linux kernel, we found 44 new bugs, and the maintainers have confirmed 36 bugs.

Acknowledgements

We would like to thank our shepherd Thorsten Holz and anonymous reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China. The authors from the University of Minnesota were supported in part by NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Atomic type documentation. https://www.kernel.org/doc/Documentation/atomic_t.txt.

[2] CVE-2016-0728 Bug Report. https://perception-point.io/resources/research/analysis-and-exploitation-of-a-linux-kernel-vulnerability/.

[3] Kref type documentation. https://www.kernel.org/doc/Documentation/kref.txt.

[4] Linux kernel git. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[5] Refcount operation documentation. https://www.kernel.org/doc/Documentation/driver-api/basics.rst.

[6] Refcount Tracing And Balancing for Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Refcount_tracing_and_balancing.

[7] Refcount_t type documentation. https://www.kernel.org/doc/Documentation/core-api/refcount-vs-atomic.rst.

[8] Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/.

[9] Sparse: a semantic parser for C. https://www.kernel.org/doc/html/v4.14/dev-tools/sparse.html.

[10] A. Alexandresc. *Modern C++ design: generic programming and design patterns applied*. 2001.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, 2017.

[12] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. 2005.

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[14] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1707–1722, 2019.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 265–278, 2011.

[16] Jonathan Corbet. Faster reference-count overflow protection. https://lwn.net/Articles/728675/.

[17] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying Reference Counting Implementations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 352–367, Berlin, Heidelberg, 2009.

[18] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, 2001.

[19] David Gay, Rob Ennals, and Eric Brewer. Safe Manual Memory Management. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)*, page 2–14, 2007.

[20] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-Miner: Uncovering Memory Corruption in Linux. In *Proceedings of 25th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018.

[21] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic Heap Layout Manipulation for Exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 763–779, Baltimore, MD, August 2018.

[22] Greg Kroah-Hartman. Kobjects and Krefs. In *Proceedings of the Linux Symposium*, 2004.

[23] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[24] Siliang Li and Gang Tan. Finding Reference-counting Errors in Python/C Programs with Affine Analysis. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 80–104, 2014.

[25] Kangjie Lu and Hong Hu. Where Does It Go?: Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, pages 1867–1881, 11 2019.

[26] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[27] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 920–932, 2016.

[28] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr.Checker: A Soundy Analysis for Linux Kernel Drivers. In *Proceedings of 26th USENIX Security Symposium (USENIX Security)*, pages 1007–1024, Vancouver, BC, August 2017.

[29] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: Finding Reference Count Bugs with Inconsistent Path Pair Checking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 531–544, 2016.

[30] Paul E. McKenney. Overview of linux-kernel reference counting. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2167.pdf/.

[31] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, page 361–377, 2015.

[32] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/European Conference on Computer Systems (EuroSys)*, page 247–260, 2008.

[33] Sebastian Poeplau and Aurélien Francillon. Symbolic Execution with SymCC: Don't Interpret, Compile! In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 181–198, 2020.

[34] David A. Ramos and Dawson Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, pages 49–64, Washington, D.C., 2015.

[35] Elena Reshetova. Conversion from atomic_t to refcount_t: summary of issues. https://www.openwall.com/lists/kernel-hardening/2016/11/28/4.

[36] Elena Reshetova, Hans Liljestrand, Andrew Paverd, and N Asokan. Toward Linux kernel memory safety. *Software: Practice and Experience*, 48(12):2237–2256, 2018.

[37] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *Proceedings of 26th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.

[38] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, page 1069–1084, 2011.

[39] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 163–177, Hollywood, CA, 2012.

[40] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 781–797, Baltimore, MD, August 2018.

[41] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, page 797–812, USA, 2015. IEEE Computer Society.

[42] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security)*, 2018.

[43] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross Checking. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 363–378, Austin, TX, August 2016. USENIX Association.

A Bug Results

B Known Bugs to Compare RID with CID

Table 8: List of new refcount bugs detected by CID. We manually confirmed the security impact of each bug in column 6, where "ML" represents memory leak. We also investigate the latent period of the detected bugs (in column 7). In column 5, "I-D" and "D-D" represent the INC-DEC consistency checking and DEC-DEC consistency checking respectively.

ID	File	Buggy Function	Refcount Field	Dimension	Impact	Years	Status
1	net/tipc/crypto.c	tipc_crypto_rcv	tipc_aead->refcnt	I-D	remote DoS, ML	<1	Applied
2	drivers/iommu/intel-svm.c	prq_event_thread	mm_struct->mm_users	I-D	UAF, ML	3	Submitted
3	net/ipv6/route.c	ip6_route_info_create	nexthop->refcnt	I-D	remote DoS, ML	1	Submitted
4	security/apparmor/domain.c	aa_change_profile (#Line: 1328)	aa_label->count	I-D	DoS, ML	3	Applied
5	net/ipv4/tcp_bpf.c	tcp_bpf_recvmsg	sk_psock->refcnt	I-D	remote DoS, ML	<1	Applied
6	net/tls/tls_sw.c	tls_data_ready	sk_psock->refcnt	I-D	remote DoS, ML	2	Applied
7	net/tls/tls_sw.c	bpf_exec_tx_verdict	sk_psock->refcnt	I-D	remote DoS, ML	<1	Applied
8	drivers/gpu/drm/ttm/ttm_bo.c	ttm_bo_add_move_fence	dma_fence->refcount	I-D	DoS, ML	<1	Applied
9	drivers/gpu/drm/ttm/ttm_bo_vm.c	ttm_bo_vm_fault_reserved	dma_fence->refcount	I-D	DoS, ML	1	Applied
10	security/apparmor/domain.c	aa_change_profile (#Line: 1318)	aa_label->count	D-D	DoS, ML	3	Applied
11	drivers/staging/comedi/comedi_fops.c	comedi_open	comedi_device->refcount	D-D	DoS, ML	6	Applied
12	drivers/gpu/.../huge_pages.c	igt_ppgtt_pin_update	i915_address_space->ref	D-D	DoS, ML	<1	Applied
13	security/apparmor/apparmorfs.c	policy_update	aa_label->count	D-D	DoS, ML	3	Applied
14	fs/btrfs/relocation.c	btrfs_recover_relocation	btrfs_trans_handle->use_count	D-D	DoS, ML	8	Applied
15	fs/nfs/nfs3acl.c	nfs3_set_acl	posix_acl->a_refcount	D-D	DoS, ML	15	Confirmed
16	drivers/staging/wusbcore/devconnect.c	wusb_dev_add_ncb	usb_hcd->kref	D-D	DoS, ML	12	Submitted
17	drivers/gpu/.../amdgpu_dm.c	emulated_link_detect	dc_sink->refcount	D-D	DoS, ML	2	Submitted
18	net/x25/x25_dev.c	x25_lapb_receive_frame	x25_neigh->refcnt	D-D	remote DoS, ML	9	Applied
19	sound/usb/mixer_quirks.c	snd_microii_spdif_default_get	snd_usb_audio->usage_count	D-D	UAF, ML	4	Applied
20	drivers/scsi/mpt3sas/mpt3sas_scsih.c	_scsih_pcie_device_remove_by_handle	_pcie_device->refcount	D-D	DoS, ML	3	Submitted
21	fs/ext4/ialloc.c	ext4_orphan_get	buffer_head->b_count	D-D	UAF, ML	4	Applied
22	sound/soc/ti/davinci-mcasp.c	davinci_mcasp_get_dma_type	dma_device->ref	D-D	DoS, ML	3	Applied
23	fs/configfs/dir.c	configfs_rmdir	config_item->ci_kref	D-D	DoS, ML	<1	Applied
24	fs/nfs/nfs4proc.c	nfs4_proc_layoutget	pnfs_layout_hdr->plh_refcount	D-D	DoS, ML	10	Applied
25	sound/soc/fsl/fsl_asrc_dma.c	fsl_asrc_dma_hw_params	dma_device->ref	D-D	DoS, ML	6	Applied
26	fs/nfsd/nfs4callback.c	nfsd4_process_cb_update	svc_xprt->xpt_ref	D-D	DoS, ML	9	Applied
27	net/batman-adv/sysfs.c	batadv_show_throughput_override	batadv_hard_iface->refcount	I-D, D-D	remote DoS, ML	4	Applied
28	net/batman-adv/sysfs.c	batadv_store_throughput_override	batadv_hard_iface->refcount	I-D, D-D	remote DoS, ML	4	Applied
29	net/tipc/node.c	tipc_rcv (#Line: 2033)	tipc_node->kref	I-D, D-D	remote DoS, ML	<1	Applied
30	net/tipc/node.c	tipc_rcv (#Line: 2037)	tipc_node->kref	I-D, D-D	remote DoS, ML	<1	Applied
31	net/tipc/node.c	tipc_rcv (#Line: 2066)	tipc_node->kref	I-D, D-D	remote DoS, ML	<1	Applied
32	drivers/net/wimax/i2400m/usb-fw.c	i2400mu_bus_bm_wait_for_ack	urb->kref	I-D, D-D	remote DoS, ML	11	Applied
33	net/netrom/nr_route.c	nr_add_node	nr_neigh->refcount	I-D, D-D	remote DoS, ML	3	Applied
34	drivers/infiniband/sw/siw/siw_qp_tx.c	siw_fastreg_mr	siw_mem->ref	I-D, D-D	DoS, ML	<1	Confirmed
35	net/sunrpc/clnt.c	rpc_clnt_test_and_add_xprt	rpc_xprt_switch->xps_kref	I-D, D-D	remote DoS, ML	4	Applied
36	net/sunrpc/clnt.c	rpc_clnt_test_and_add_xprt	rpc_xprt->kref	I-D, D-D	remote DoS, ML	4	Applied
37	net/batman-adv/bat_v_ogm.c	batadv_v_ogm_process	batadv_hardif_neigh_node->refcount	I-D, D-D	remote DoS, ML	4	Applied
38	drivers/staging/gasket/gasket_sysfs.c	gasket_sysfs_register_store	gasket_sysfs_mapping->refcount	I-D, D-D	DoS, ML	2	Applied
39	drivers/staging/gasket/gasket_sysfs.c	gasket_sysfs_put_attr	gasket_sysfs_mapping->refcount	I-D, D-D	DoS, ML	2	Applied
40	net/sunrpc/rpcb_clnt.c	rpcb_getport_async	rpc_xprt->kref	I-D, D-D	remote DoS, ML	12	Applied
41	drivers/scsi/lpfc/lpfc_els.c	lpfc_els_unsol_buffer	lpfc_nodelist->kref	I-D, D-D	DoS, ML	6	Applied
42	fs/afs/rotate.c	afs_select_fileserver	afs_cb_interest->usage	I-D, D-D	DoS, ML	2	Submitted
43	drivers/tty/serial/serial_core.c	uart_port_startup	uart_state->refcount	I-D, D-D	UAF, ML	2	Submitted
44	drivers/tty/serial/serial_core.c	uart_shutdown	uart_state->refcount	I-D, D-D	UAF, ML	2	Submitted

Table 9: List of 60 known bugs reported in the Linux Kernel between 2018 and 2020. We compared CID with RID on these bugs and show whether these bugs can be detected by them in column 6 and 7 respectively.

ID	File	Buggy Function	Refcount Field	Fix Commit ID	RID	CID
1	drivers/scsi/qedf/qedf_io.c	qedf_initiate_abts	fc_rport_priv->kref	56efc304b18cbfa4a2b355c0ae817f61acea38c4	✗	✓
2	drivers/scsi/qla2xxx/qla_os.c	qla2x00_abort_srb	srb->ref_count	d2d2b5a5741d317bed1fa38211f1f3b142d8cf7a	✗	✓
3	drivers/net/macsec.c	macsec_newlink	net_device->pcpu_refcnt	2bce1ebed17da54c65042ec2b962e3234bad5b47	✗	✗
4	drivers/net/wireless/virt_wifi.c	virt_wifi_newlink	module->refcnt	1962f86b42ed06ea6af9ff09390243b99d9eb83a	✗	✗
5	net/core/skbuff.c	sock_zerocopy_realloc	ubuf_info->refcnt	100f6d8e09905c59be45b6316f8f369c0be1b2d8	✓	✗
6	kernel/bpf/hashtab.c	alloc_htab_elem	bpf_htab->count	7f93d1295131c9a8b6ff5eec13eef094f0d42921	✗	✓
7	drivers/nvme/target/fabrics-cmd.c	nvmet_install_queue	nvmet_ctrl->ref	1a3f540d63152b8db0a12de508bfa03776217d83	✗	✗
8	net/sched/cls_u32.c	u32_change	tc_u_hnode->refcnt	275c44aa194b7159d1191817b20e076f55f0e620	✗	✗
9	fs/cifs/smb2ops.c	open_shroot	cached_fid->refcount	2f94a3125b8742b05a011d62b16f52eb8f9ebe1c	✗	✓
10	drivers/scsi/qedf/qedf_main.c	qedf_xmit	fc_rport_priv->kref	4262d35c32c652344b6784cad51ec5a0e2e5258b	✗	✓
11	drivers/usb/serial/mos7720.c	write_parport_reg_nonblock	mos7715_parport->ref_count	2908b076f5198d231de62713cb2b633a3a4b95ac	✗	✓
12	drivers/media/usb/uvc/uvc_driver.c	uvc_probe	uvc_device->ref	f9ffcb0a21e1fa8e64d09ed613d884e054ae8191	✗	✓
13	fs/nfs/nfs4proc.c	nfs4_alloc_unlockdata	nfs4_lock_state->ls_count	3028efe03be9c8c4cd7923f0f3c39b2871cc8a8f	✗	✓
14	fs/nfs/nfs4proc.c	nfs4_alloc_lockdata	nfs4_lock_state->ls_count	3028efe03be9c8c4cd7923f0f3c39b2871cc8a8f	✗	✓
15	sound/pci/hda/hda_intel.c	atpx_present	kobject->kref	6e8aeda224c83c7c7841e143d410b6d0e7bda05e	✓	✗
16	drivers/md/dm-zoned-target.c	dmz_submit_bio	dmz_bioctx->ref	0c8e9c2d668278652af028c3cc068c65f66342f4	✗	✓
17	drivers/infiniband/core/device.c	iw_query_port	in_device->refcnt	390d3fdcae2da52755b31aa44fcf19ecb5a2488b	✓	✓
18	drivers/video/fbdev/clps711x-fb.c	clps711x_fb_probe	kobject->kref	fdac751355cd76e049f628afe6acb8ff4b1399f7	✓	✓
19	net/l2tp/l2tp_core.c	l2tp_tunnel_register	sock->sk_refcnt	f8504f4ca0a0e9f84546ef86e00b24d2ea9a0bd2	✓	✓
20	drivers/net/ethernet/mellanox/mlxsw/core_acl_flex_actions.c	pppol2tp_tunnel_ioctl	l2tp_session->ref_count	212dab0541eb916f29d55f914c8e84e13c6b214d	✓	✓
21	drivers/mtd/spi-nor/nxp-spifi.c	nxp_spifi_probe	kobject->kref	38ebbe2b7282e985a7acc862892564e8fbbde866	✓	✓
22	net/netfilter/ipvs/ip_vs_app.c	do_ip_vs_set_ctl	module->refcnt	62931f59ce9cbabb934a431f48f2f1f441c605ac	✓	✓
23	fs/afs/cell.c	afs_lookup_cell_rcu	afs_cell->usage	a5fb8e6c02d6a518fb2b1a2b8c2471fa77b69436	✓	✓
24	fs/nfs/nfs4proc.c	nfs41_check_delegation_stateid	cred->usage	8c39a39e28b86a4021d9be314ce01019bafa5fdc	✓	✓
25	drivers/media/platform/mtk-mdp/mtk_mdp_core.c	mtk_mdp_probe	kobject->kref	864919ea0380e62adb2503b89825fe358acb8216	✗	✓
26	drivers/media/platform/exynos4-is/media-dev.c	__of_get_csis_id	kobject->kref	da79bf41a4d170ca93cc8f3881a70d734a071c37	✗	✓
27	drivers/media/platform/exynos4-is/fimc-is.c	fimc_is_probe	kobject->kref	da79bf41a4d170ca93cc8f3881a70d734a071c37	✗	✓
28	drivers/media/platform/exynos4-is/media-dev.c	fimc_md_register_sensor_entities	kobject->kref	da79bf41a4d170ca93cc8f3881a70d734a071c37	✗	✓
29	sound/soc/samsung/odroid.c	odroid_audio_probe (#Line: 238)	kobject->kref	d832d2b246c516eacb2d0ba53ec17ed59c3cd62b	✗	✓
30	sound/soc/samsung/odroid.c	odroid_audio_probe (#Line: 239)	kobject->kref	d832d2b246c516eacb2d0ba53ec17ed59c3cd62b	✗	✓
31	sound/soc/samsung/odroid.c	odroid_audio_probe (#Line: 274)	kobject->kref	d832d2b246c516eacb2d0ba53ec17ed59c3cd62b	✗	✓
32	drivers/pci/hotplug/rpadlpar_core.c	find_vio_slot_node	kobject->kref	fb26228bfc4ce3951544848555c0278e2832e618	✗	✓
33	drivers/pci/hotplug/rpadlpar_core.c	dlpar_remove_slot	kobject->kref	fb26228bfc4ce3951544848555c0278e2832e618	✗	✓
34	drivers/pci/hotplug/rpadlpar_core.c	dlpar_add_slot	kobject->kref	fb26228bfc4ce3951544848555c0278e2832e618	✗	✓
35	drivers/gpu/drm/drm_syncobj.c	drm_syncobj_find_fence	drm_syncobj->refcount	bc9c80fe01a2570a2fd78abbc492b377b5fda068	✗	✓
36	drivers/acpi/utils.c	acpi_dev_get_first_match_name	kobject->kref	817b4d64da036f5559297a2fdb82b8b14f4ffdcd	✗	✓
37	drivers/gpu/drm/i915/gvt/dmabuf.c	intel_vgpu_get_dmabuf	drm_gem_object->refcount	41d931459b53e32c67a1f8838d1e6826a69ee745	✗	✓
38	drivers/net/wireless/intersil/p54/p54pci.c	p54p_probe	kobject->kref	8149069db81853570a665f5e5648c0e526dc0e43	✗	✓
39	drivers/md/dm-ioctl.c	dm_early_create	mapped_device->holders	311f71281ff4b24f86a39c60c959f485c68a6d36	✗	✓
40	drivers/scsi/qla2xxx/qla_os.c	qla2xxx_eh_abort	srb->ref_count	8dd9593cc07ad7d999bef81b06789ef873a94881	✗	✓
41	drivers/pinctrl/samsung/pinctrl-samsung.c	samsung_pinctrl_create_functions (#Line: 782)	kobject->kref	a322b3377f4bac32aa25fb1acb9e7afbbbbd0137	✗	✓
42	drivers/pinctrl/samsung/pinctrl-samsung.c	samsung_pinctrl_create_functions (#Line: 797)	kobject->kref	a322b3377f4bac32aa25fb1acb9e7afbbbbd0137	✗	✓
43	drivers/pinctrl/samsung/pinctrl-samsung.c	samsung_dt_node_to_map	kobject->kref	a322b3377f4bac32aa25fb1acb9e7afbbbbd0137	✗	✓
44	drivers/pinctrl/samsung/pinctrl-s3c64xx.c	s3c64xx_eint_eint0_init	kobject->kref	7f028caadf6c37580d0f59c6c094ed09afc04062	✗	✓
45	drivers/pinctrl/samsung/pinctrl-s3c24xx.c	s3c24xx_eint_init	kobject->kref	6fbbcb050802d6ea109f387e961b1dbcc3a80c96	✗	✓
46	drivers/pinctrl/samsung/pinctrl-exynos.c	exynos_eint_wkup_init	kobject->kref	5c7f48dd14e892e3e920dd6bbbd52df79e1b3b41	✗	✓
47	drivers/gpu/drm/drm_gem.c	drm_gem_ttm_mmap	drm_gem_object->refcount	9786b65bc61acec63f923978c75e707afbb74bc7	✗	✓
48	kernel/bpf/syscall.c	bpf_map_get_fd_by_id	bpf_map->usercnt	781e62823cb81b972dc8652c1827205cda2ac9ac	✗	✓
49	drivers/power/reset/zx-reboot.c	zx_reboot_probe	kobject->kref	f052df96c46dbe52fbacd02189e7906f41686f27	✗	✓
50	drivers/media/i2c/tc358743.c	tc358743_probe_of	kobject->kref	64bac6916ef7d9cc57367893aea1544fcad91b9b	✗	✓
51	net/batman-adv/bat_v.c	batadv_v_gw_dump_entry	batadv_gw_node->refcount	9713cb0cf19f1cec6c007e3b37be0697042b6720	✗	✓
52	net/batman-adv/bat_iv_ogm.c	batadv_iv_gw_dump_entry	batadv_gw_node->refcount	b5685d2687d6612adf5eac519eb7008f74dfd1ec	✗	✓
53	drivers/net/ethernet/netronome/nfp/flower/tunnel_conf.c	nfp_tun_neigh_event_handler	dst_entry->__refcnt	e62e51af3430745630f0cf76bb41a28d20c4ebdc	✗	✓
54	drivers/cpufreq/brcmstb-avs-cpufreq.c	brcm_avs_cpufreq_get	kobject->kref	a48ac1c9f294e1a9b692d9458de6e6b58da8b07d	✗	✓
55	drivers/cpufreq/s3c2416-cpufreq.c	s3c2416_cpufreq_reboot_notifier_evt	kobject->kref	8ead819f1befae08182c772b6fdf8ac201b34566	✗	✓
56	drivers/net/dsa/rtl8366rb.c	rtl8366rb_setup_cascaded_irq	kobject->kref	f32eb9d80470dab05df26b6efd02d653c72e6a11	✗	✓
57	fs/fuse/cuse.c	cuse_channel_open	fuse_conn->count	9ad09b1976c562061636ff1e01bfc3a57aebe56b	✗	✓
58	drivers/of/resolver.c	adjust_local_phandle_references	kobject->kref	60d437bbff358748fcfc3bce5f08da9a6b3761da	✗	✓
59	drivers/soc/ux500/ux500-soc-id.c	ux500_soc_device_init	kobject->kref	dbc3c6295195267ea7bc48d46030c7b244f8b11e	✗	✓
60	drivers/media/platform/ti-vpe/cal.c	of_get_next_port	kobject->kref	094efbe748c204fb2e10ebf6f100da926e10fc2f	✗	✓