Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking

Xin Tan, Yuan Zhang, and Xiyu Yang, Fudan University; Kangjie Lu, University of Minnesota; Min Yang, Fudan University

https://www.usenix.org/conference/usenixsecurity21/presentation/tan

This paper is included in the Proceedings of the 30th USENIX Security Symposium.
August 11–13, 2021
978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.
Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking

Xin Tan¹,‡, Yuan Zhang¹,‡, Xiyu Yang¹, Kangjie Lu², and Min Yang¹

¹School of Computer Science, Fudan University, China
²Department 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, and 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...
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 CID¹, 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, often times 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 strength 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.

¹CID 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.
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.

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.

### Table 3: The usage for the selected 300 fields.

<table>
<thead>
<tr>
<th>Lock/Status</th>
<th>Token/ID</th>
<th>Normal Counter</th>
<th>Refcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_t</td>
<td>16</td>
<td>2</td>
<td>45</td>
</tr>
<tr>
<td>atomic_long_t</td>
<td>2</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>atomic64_t</td>
<td>0</td>
<td>13</td>
<td>34</td>
</tr>
<tr>
<td>refcount_t</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>kref</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>18</strong></td>
<td><strong>16</strong></td>
<td><strong>121</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Refcount Type</th>
<th># of Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_t</td>
<td>2,010</td>
</tr>
<tr>
<td>atomic_long_t</td>
<td>154</td>
</tr>
<tr>
<td>atomic64_t</td>
<td>334</td>
</tr>
<tr>
<td>refcount_t</td>
<td>297</td>
</tr>
<tr>
<td>kref</td>
<td>425</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3,220</strong></td>
</tr>
</tbody>
</table>

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...
void usb_kill_anchored_urbs(struct usb_anchor *anchor)

/* /drivers/usb/core/urb.c */

while (!list_empty(&anchor->urb_list)) {
  //decrease the refcount
  while (!list_empty(&anchor->urb_list)) {
    //decrease the refcount
    urb = request_single_step_set_feature_urb(...);
    if (!hard_iface)
      return -EINVAL;
    ret = batadv_parse_throughput(...);
    if (ret)
      goto out;
  }
  hard_iface = NULL;
  return retval;
}

struct batadv_hard_iface* batadv_hardif_get_by_netdev(...)
{
  ... }

(a) Caller with Correct DEC Operations
(b) Caller with Correct DEC Operations
(c) Caller with Buggy DEC Operations

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       //increase the refcount
6    while (!list_empty(&anchor->urb_list)) {
7        //increase refcount if find the hard_iface
8        kref_get_unless_zero(&hard_iface->refcount))
9      goto out;
10    }
11  hard_iface = NULL;
12  return hard_iface;
13 }
14
does not decrease the refcount.

Figure 1 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 bytecode 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.
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 \texttt{batadv_hardif_get_by_netdev}, based on which its caller performs the corresponding refcount decrease. Therefore, CID performs 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 \langle Action, RetVal \rangle to model the INC behaviors for a path in the INC function. In this tuple, \textit{Action} can be INC or EMPTY, which depicts the refcount of the object is incremented or not. The \textit{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 \textit{Action} and \textit{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 \langle Action, Conditions \rangle to represent the DEC behaviors in this caller. The \textit{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 \textit{Actions}. 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 \textit{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 \langle Action, RetVal \rangle, 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
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 a 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.

\[
\begin{align*}
x &= \text{num of callers implies EMPTY} \\
y &= \text{num of callers implies DEC} \\
z &= \text{num of callers implies UNKNOWN} \\
\text{length} &= \sqrt{x^2 + y^2 + z^2} \\
\text{vector} &= \left(\frac{x}{\text{length}}, \frac{y}{\text{length}}, \frac{z}{\text{length}}\right) = (x_{nv}, y_{nv}, z_{nv})
\end{align*}
\]

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

\[
\begin{align*}
distance_x &= \sqrt{(x_{nv} - 1)^2 + y_{nv}^2 + z_{nv}^2} \\
distance_y &= \sqrt{x_{nv}^2 + (y_{nv} - 1)^2 + z_{nv}^2} \\
distance_z &= \sqrt{x_{nv}^2 + y_{nv}^2 + (z_{nv} - 1)^2}
\end{align*}
\]

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

\[
\text{inconsistency score} = \min(d\text{istance}_x, d\text{istance}_y, d\text{istance}_z)
\]

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 ($\theta$). 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 bytecode 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 ($\theta$) of the inconsistency score among all the callers of an INC function. The higher of the $\theta$, the more bugs would be reported by CID, but the higher false positive rate CID may have. By trying several values for $\theta$, 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 $\theta$ increases from 0.4 to 0.5, while only 10 more bugs are reported when increasing $\theta$ from 0.5 to 0.6. Therefore, to control the volume of reported bugs, we choose $\theta = 0.5$ for the following bug confirmation.

#### 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.

<table>
<thead>
<tr>
<th>Threshold ($\theta$)</th>
<th>Reported Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>18</td>
</tr>
<tr>
<td>0.2</td>
<td>55</td>
</tr>
<tr>
<td>0.3</td>
<td>67</td>
</tr>
<tr>
<td>0.4</td>
<td>86</td>
</tr>
<tr>
<td>0.5</td>
<td>119</td>
</tr>
<tr>
<td>0.6</td>
<td>129</td>
</tr>
</tbody>
</table>
Efficiency. CIDS completes the analysis of the whole kernel within 18 minutes, of which loading bytecode 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 clocl) for the Linux kernel. According to the results, we confirm that CIDS is quite efficient to scale to the highly complex whole-kernel analysis.

6.3 False Positives Breakdown

Among the 149 bugs reported by CIDS, 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).** CIDS 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 CIDS can not recognize. When the escape analysis exhibits a false negative, CIDS may wrongly expects a paired DEC operation in the current function, causing a false positive.

- **Imprecise alias analysis (23 FPs):** In refcount operation analysis, CIDS 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 CIDS prevents it from finding some paired DEC operations that are performed on aliased object pointers. Therefore, CIDS 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 CIDS 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.

**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 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.

```c
1 static int comedi_open(struct inode *inode, struct file *file)
2 {
3    ...
4    // increase refcount if success
5   ... // 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 CIDS in `comedi_open()`, which results in memory leak and DoS.
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.

**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-
We present the causes for them below.

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

<table>
<thead>
<tr>
<th>Refcount Type</th>
<th># of Fields</th>
<th># of Refcount Fields</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_t</td>
<td>2,010</td>
<td>140</td>
<td>6.97%</td>
</tr>
<tr>
<td>atomic_long_t</td>
<td>154</td>
<td>5</td>
<td>3.25%</td>
</tr>
<tr>
<td>atomic64_t</td>
<td>334</td>
<td>3</td>
<td>0.90%</td>
</tr>
<tr>
<td>refcount_t</td>
<td>297</td>
<td>251</td>
<td>84.5%</td>
</tr>
<tr>
<td>kref_t</td>
<td>425</td>
<td>393</td>
<td>92.5%</td>
</tr>
<tr>
<td>Total</td>
<td>3,220</td>
<td>792</td>
<td>24.6%</td>
</tr>
</tbody>
</table>

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, I> and <DEC, I>.

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

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, I> and <DEC, I>.

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_preactalloc_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 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 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_preactalloc_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.
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. Through 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. CR IX [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


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.

<table>
<thead>
<tr>
<th>ID</th>
<th>File</th>
<th>Buggy Function</th>
<th>Refcount Field</th>
<th>Dimension</th>
<th>Impact</th>
<th>Years</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>net/tipc/crypto.c</td>
<td>tipc_crypto_rcv</td>
<td>tipc_aead-&gt;refcnt</td>
<td>I-D</td>
<td>remote DoS, ML</td>
<td>&lt;1</td>
<td>Applied</td>
</tr>
<tr>
<td>2</td>
<td>drivers/summer/intel-svm.c</td>
<td>prq_event_thread</td>
<td>mm_struct-&gt;mm_users</td>
<td>I-D</td>
<td>UAF, ML</td>
<td>3</td>
<td>Submitted</td>
</tr>
<tr>
<td>3</td>
<td>net/ipv6/rtcube</td>
<td>ipv6_route_info_create</td>
<td>nexthop-&gt;refcount</td>
<td>I-D</td>
<td>remote DoS, ML</td>
<td>1</td>
<td>Submitted</td>
</tr>
<tr>
<td>4</td>
<td>security/apparmor/domain.c</td>
<td>aa_change_profile</td>
<td>aa_label-&gt;account</td>
<td>I-D</td>
<td>DoS, ML</td>
<td>3</td>
<td>Applied</td>
</tr>
<tr>
<td>5</td>
<td>net/ipv4/udp_bpf.c</td>
<td>tcp_bpf_rcvmsg</td>
<td>sk_psock-&gt;refcnt</td>
<td>I-D</td>
<td>remote DoS, ML</td>
<td>&lt;1</td>
<td>Applied</td>
</tr>
<tr>
<td>6</td>
<td>net/blk/blk_sw.c</td>
<td>blk_data_ready</td>
<td>sk_psock-&gt;refcnt</td>
<td>I-D</td>
<td>remote DoS, ML</td>
<td>2</td>
<td>Applied</td>
</tr>
<tr>
<td>7</td>
<td>net/blk/blk_sw.c</td>
<td>blk_exec_ts_art</td>
<td>sk_psock-&gt;refcnt</td>
<td>I-D</td>
<td>remote DoS, ML</td>
<td>&lt;1</td>
<td>Applied</td>
</tr>
<tr>
<td>8</td>
<td>drivers/gpu/dm/ttm/ttm_bo.c</td>
<td>ttm_bo_add_move_fence</td>
<td>dma_fence-&gt;refcount</td>
<td>I-D</td>
<td>DoS, ML</td>
<td>&lt;1</td>
<td>Applied</td>
</tr>
<tr>
<td>9</td>
<td>drivers/gpu/dm/ttm/ttm_bo_vm.c</td>
<td>ttm_bo_vm_fault_reserved</td>
<td>dma_fence-&gt;refcount</td>
<td>I-D</td>
<td>DoS, ML</td>
<td>1</td>
<td>Applied</td>
</tr>
<tr>
<td>10</td>
<td>security/apparmor/domain.c</td>
<td>aa_change_profile (#Line: 1318)</td>
<td>aa_label-&gt;account</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>3</td>
<td>Applied</td>
</tr>
<tr>
<td>11</td>
<td>drivers/staging/comedi/comedi_fops.c</td>
<td>comedi_open</td>
<td>comedi_device-&gt;refcount</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>6</td>
<td>Applied</td>
</tr>
<tr>
<td>12</td>
<td>drivers/gpu/i915/i915_fuse.c</td>
<td>i915_fuse-&gt;fuse</td>
<td>fuse-&gt;refcount</td>
<td>I-D</td>
<td>DoS, ML</td>
<td>15</td>
<td>Confirmed</td>
</tr>
<tr>
<td>13</td>
<td>security/apparmor/apparmorfs.c</td>
<td>policy_update</td>
<td>aa_label-&gt;account</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>3</td>
<td>Applied</td>
</tr>
<tr>
<td>14</td>
<td>fs/btrfs/btrfs_crypt</td>
<td>btrfs_after_property</td>
<td>btrfs-&gt;refcount</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>8</td>
<td>Applied</td>
</tr>
<tr>
<td>15</td>
<td>fs/btrfs/btrfs_crypt</td>
<td>btrfs_after_property</td>
<td>btrfs-&gt;refcount</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>8</td>
<td>Applied</td>
</tr>
<tr>
<td>16</td>
<td>drivers/staging/wuscore/devconnect.c</td>
<td>i915_address_space-&gt;ref</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>15</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>drivers/staging/wuscore/devconnect.c</td>
<td>i915_address_space-&gt;ref</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>15</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>net/ipv6/rtip/ipv6_rdc.c</td>
<td>ipv6_rdc-&gt;refcount</td>
<td>ipv6_rdc-&gt;refcount</td>
<td>I-D</td>
<td>DoS, ML</td>
<td>2</td>
<td>Applied</td>
</tr>
<tr>
<td>19</td>
<td>security/apparmor/apparmorfs.c</td>
<td>policy_update</td>
<td>aa_label-&gt;account</td>
<td>D-D</td>
<td>DoS, ML</td>
<td>3</td>
<td>Applied</td>
</tr>
</tbody>
</table>

Note: The table continues with more entries. The columns include ID, File, Buggy Function, Refcount Field, Dimension, Impact, Years, and Status.
<table>
<thead>
<tr>
<th>ID</th>
<th>File</th>
<th>Buggy Function</th>
<th>Refcount Field</th>
<th>Fix Commit ID</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>drivers/scuss/qdrep/qdrep.so</td>
<td>qdrep_initiate sbts</td>
<td>fc_priv_fd-&gt;kref</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>drivers/scuss/qdrep/qdrep.so</td>
<td>qdrep_20000000 sbt</td>
<td>sbt-&gt;ref_count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>drivers/scuss/qdrep/qdrep.so</td>
<td>macsec_newlink</td>
<td>net_device-&gt;opps_refcnt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>drivers/scuss/qdrep/qdrep.so</td>
<td>macsec_newlink</td>
<td>net_device-&gt;opps_refcnt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>net/conf/ulstk/ulstk</td>
<td>sock_zero_copy_realloc</td>
<td>ulstk-&gt;inf-&gt;kref</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>kernel/bpf/bpf.so</td>
<td>alloc_bpf_elem</td>
<td>bpf_bpf-&gt;kobj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>drivers/scuss/qdrep/qdrep.so</td>
<td>macsec_newlink</td>
<td>net_device-&gt;opps_refcnt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>drivers/scuss/qdrep/qdrep.so</td>
<td>macsec_newlink</td>
<td>net_device-&gt;opps_refcnt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 7 and respectively.