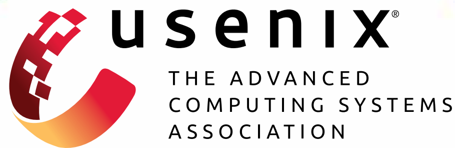

LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution

Jian Liu, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences;* Lin Yi, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences;* Weiteng Chen, Chengyu Song, and Zhiyun Qian, *UC Riverside;* Qiuping Yi, *Beijing University of Posts and Telecommunications and Beijing Key Lab of Intelligent Telecommunication Software and Multimedia*

https://www.usenix.org/conference/usenixsecurity22/presentation/liu-jian

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution

Jian Liu[1,4], Lin Yi[1], Weiteng Chen[2], Chengyu Song[2], Zhiyun Qian[2], and Qiuping Yi[3,5]

[1]*{CAS-KLONAT*, BKLONSPT†}, Institute of Information Engineering, Chinese Academy of Sciences*

[2]*UC Riverside*

[3]*Beijing University of Posts and Telecommunications*

[4]*School of Cyber Security, University of Chinese Academy of Sciences*

[5]*Beijing Key Lab of Intelligent Telecomm. Software and Multimedia*

liujian6@iie.ac.cn, teroincn@gmail.com, {wchen130,csong,zhiyunq}@cs.ucr.edu, yiqiuping@gmail.com

Abstract

Linux kernel employs reference counters, which record the number of references to a shared kernel object, to track its lifecycle and prevent memory errors like use-after-free. However, the usage of reference counters can be tricky and often error-prone, especially considering unique kernel conventions of managing reference counters (e.g., external vs. internal reference counters). In this paper, we aim to automatically discover incorrect usage of reference counters, overcoming two key challenges: (1) scalability and (2) the aforementioned unique kernel conventions. Specifically, we develop a tiered program analysis based solution to efficiently and precisely check the imbalances between the change in the actual number of references and the corresponding reference counter. We apply our tool to the 4.14.0 kernel (with allyesconfig) and find 118 bugs, out of which 87 are new. The result shows our tool is scalable and effective.

1 Introduction

Vulnerabilities in operating system (OS) kernels can lead to serious attacks (e.g., privilege escalation and information leakage) that compromise the confidentiality and integrity of the foundation that applications rely on. Memory management errors such as use-after-free (UAF) and double-free are among the most severe kernel vulnerabilities, which often allow attackers to perform arbitrary read/write and code execution [35]. Therefore, they have drawn significant attentions from the security community to harden the kernel against these vulnerabilities.

*Key Laboratory of Network Assessment Technology, CAS

†Beijing Key Laboratory of Network Security and Protection

As one of most complex system software written in unsafe languages, the Linux kernel provides many primitives to avoid memory management errors. One of these primitives that aims to prevent memory object from being prematurely freed is *reference counter* (referred to as *refcount* hereinafter). A refcount records the number of references (i.e., pointers in the C language) to a given memory object. A positive refcount means a memory object could be accessed in the future, hence it should not be freed. When a refcount is decremented to zero, the kernel knows that the associated memory object will no longer be accessed thus is safe to be freed. In the Linux kernel, refcounts are widely used for resource management in every aspect, from user-data, files, memory allocations, to hardware resources [28]. Refcount mechanisms in the Linux kernel typically employ a simple data structure called `struct kref` which encapsulates an integer as the real counter, and provide APIs (e.g., `kref_get` and `kref_put`) for manipulating the counter [13].

Since the refcount mechanisms are not automatic, Linux kernel developers are required to ensure that all operations on refcounts are performed as intended; in other words, *the refcount and corresponding reference changes should be consistent*. Unfortunately, as many other manually enforced security invariants, they are prone to errors. Based on our study, the most common bugs related to refcounts are (1) failure to increase the counter when a new reference is created, and (2) failure to decrement the counter when a reference has been removed. The first case can easily lead to use-after-free (UAF), which is usually exploitable (e.g., CVE-2016-4557 [23]); and the second case usually results in memory leaks. However, if the refcount can be incremented repeatedly and indefinitely (causing an overflow of the refcount), it can also be exploited to trigger UAF: CVE-2016-0728 [22] is an example of such vulnerabilities. For this reason, the Linux kernel has adopted

a hardening against refcount overflow [14].

In this work, we aim to discover these refcount-related errors using program analysis techniques. There are two major challenges in discovering such erroneous refcount usage patterns. First, it fundamentally requires a path-sensitive analysis to check whether a refcount increment/decrement matches the addition/removal of new references along a *feasible* path. However, it is well-known that path-sensitive static analysis is extremely difficult to scale and dynamic approaches such as fuzzing can cover only a limited number of paths. Second, refcount uses in the Linux kernel have special conventions that exhibit benign violations of the above consistency rule. Therefore, without capturing such benign conventions, a tool will likely produce a large number of false positives and become impractical to use. As an example, the Linux kernel has the notion of *internal references* that do not actually need to be refcounted [1] (see §2.2 for more details).

As of now, no research has adequately addressed both challenges. Mao et al. [20] proposed using inconsistent path pair (IPP) to find refcount bugs in the Linux kernel: if two code paths return the same error code but manipulate the refcount differently (e.g., have different number of `get` or `put`), then one of them is likely to have a bug. This specific pattern encoded in the analysis is a less precise (considering only refcount changes without reference changes) or weaker version of the consistency rule we mentioned earlier; therefore, it can only uncover a subset of refcount bugs. For instance, refcount bugs can happen along paths that return different error code (e.g., Figure 3); conversely, all paths that return the same error code might be buggy but consistent. Some works do conduct path-sensitive analyses but only within a limited scope. For example, to find refcount bugs in Python extensions written in C, Li and Tan [15] adopted an *escape rule* that requires, in any function and any of its paths, the change of a refcount must be equal to the number of references escaped from the function. Since the analysis is tailored towards the native C used in Python modules, it relies on well-defined interfaces (between Python and C) to track refcount changes and the analysis scope. In addition, it employs the notion of "shallow aliasing" which can easily miss aliases (which is especially challenging to track in the Linux kernel), leading to imprecision analysis results. Finally, none of the related work addresses the second challenge of special Linux conventions of refcount usage; therefore, they can induce many false positives.

Motivated by the above deficiencies, we develop LinKRID (Linux Kernel Refcount Imbalance Detector), a scalable and practical refcount bug discovery tool for the Linux kernel, that addresses the aforementioned challenges. To scale up the path-sensitive analysis, we apply a tiered analysis — starting with a lightweight static analysis to identify a constrained scope of a refcounted object, through which we can then analyze in detail the corresponding refcount changes and global reference changes using under-constrained symbolic execution. We also codify a set of conventions currently specified in natural language (e.g., documentations) to filter seemingly erroneous but benign usage patterns. To our knowledge, LinKRID is the first tool that attempts to fill the void of addressing both the imprecise modeling and Linux-specific refcount usage conventions, packaged in a scalable analysis tool representing an important step towards automating the discovery of Linux kernel refcount bugs. We applied LinKRID to the 4.14.0 kernel, which includes all drivers, file systems, and other peripheral modules that can be compiled under the x86 architecture. LinKRID successfully re-discovered all refcount bugs reported in [20], and further found 118 bugs, within which 87 are new.

In summary, this paper makes the following key contributions:

- We identified the limitations of prior static analyses for discovering refcount bugs, and proposed more general patterns that reflect the root causes of refcount bugs.
- We propose a combination of static analysis and under-constrained symbolic execution to track the dynamic reference and refcount changes. We show this approach is precise and efficient.
- We developed a practical refcount bug finding tool and found 87 new bugs in the v4.14 Linux kernel. We plan to open source the tool entirely to facilitate the reproduction of the results and future research.

2 Refcount Bugs in Linux

The purpose of refcount mechanisms is to manage the lifecycle of shared memory objects [28]. A refcount is, in general, a counter that records how many (live) references/pointers can be used/dereferenced to access the shared object (in the future). Note that such references/pointers do not have to point to the beginning of the object, as long as they point to the memory space occupied by the object, they should be refcounted.

Each time, a new live reference to an object is created, the corresponding refcount should be increased by 1. When the lifecycle of a reference ends (i.e., will no long be used to access the object), the refcount should be decreased by 1. Refcount can be automatically maintained through programming language (e.g., Python) or containers (e.g., `std::shared_ptr` in C++). In the Linux kernel, however, refcounts are manually managed by developers, which makes them error-prone. Developers must guarantee the refcount consistency (i.e., *refcount = #(live reference)*) in all execution paths, including all error handling paths.

2.1 Bugs due to Complex Paths

Figure 1 is a UAF example (CVE-2016-0728 [22]) in the Linux kernel caused by refcount bug in an error handling path.

```
1  long join_session_keyring(const char *name) {
2    struct cred *new;
3    struct key *keyring;
4    long ret;
5    ...
6    keyring = find_keyring_by_name(...);
7    if (PTR_ERR(keyring) == -ENOKEY) {
8      keyring = keyring_alloc(...);
9      if (IS_ERR(keyring)) {
10       ret = PTR_ERR(keyring);
11       goto error2;
12     }
13   } else if (IS_ERR(keyring)) {
14     ret = PTR_ERR(keyring);
15     goto error2;
16   } else if (keyring == new->session_keyring) {
17     ret = 0;
18     goto error2;
19   }
20   ...
21   install_session_keyring_to_cred(new, keyring);
22   ret = keyring->serial;
23   key_put(keyring);
24   return ret;
25 error2:
26   return ret;
27 }
28 struct key *find_keyring_by_name(...) {
29   struct key *keyring;
30   // if name match
31   if (atomic_inc_not_zero(&keyring->usage))
32     return keyring;
33   return ERR_PTR(-ENOKEY);
34 }
```

Figure 1: Simplified code snippet of CVE-2016-0728 [25], a missing refcount decrement bug.

In function `join_session_keyring()`, if the kernel successfully found the corresponding keyring, the object `keyring`'s refcount is increased by `atomic_inc_not_zero()` at line 31; otherwise it returns an error code. A special case/path, however, is when the look up is successful but the keyring is the same as the current one (line 16). One the return path, the refcount is not decremented (e.g., line 23). This violates the refcount consistency of `keyring` and can lead to UAF as it can be triggered multiple times to overflow the `keyring`'s refcount (at that time, before the overflow hardening is introduced). Note that this refcount bug can only be triggered on this path. On other paths, the refcount changes and reference changes are balanced, as the reference `keyring` is a local variable and will be automatically released once the function returns; and the refcount is not incremented on other error returning paths.

This example illustrates a few important points: (1) We need a more fundamental invariant like refcount consistency *within a single path* to reliably detect all refcount bugs. Prior work RID [20] detects inconsistencies *across paths*: if there are two paths where the refcount changes are different but the return values are the same, it usually indicates a refcount bug. RID is unable to detect the example bug because the erroneous path is the only path that returns 0. (2) We need a

```
1  static void amp_mgr_destroy(struct kref *kref) {
2    struct amp_mgr *mgr = container_of(kref, struct
      amp_mgr, kref);
3    mutex_lock(&amp_mgr_list_lock);
4    list_del(&mgr->list);
5    mutex_unlock(&amp_mgr_list_lock);
6    kfree(mgr);
7  }
8  int amp_mgr_put(struct amp_mgr *mgr) {
9    return kref_put(&mgr->kref,
      &amp_mgr_destroy);
10 }
11
```

Figure 2: An example of internal reference.

path-sensitive analysis to precisely identify the problematic path, which leads to two decrements of a refcount but only one actual reference removal. If we only perform a flow-sensitive analysis and consider what "may" happen (instead of what is feasible), then we will report false alarms for error other paths that also go to label `error2`. (3) It is necessary to conduct an inter-procedural analysis to reduce false positives. In the example, refcount manipulation and reference acquisition/release happen in different functions, intra-procedural analysis like [7] would result in many false alarms.

2.2 Internal References

Besides the sheer complexity, special design patterns of refcounts in the Linux kernel introduce additional challenges in detecting refcount bugs. In particular, there are two distinct types of references to a kernel object [1]: *external reference* and *internal reference*. An external reference (or strong reference) is a reference we have discussed above: its purpose is to access the object. Therefore, an external reference has to strictly abide the reference consistency invariant.

On the contrary, an internal reference (or weak reference) *is not refcounted*. Unlike external references, the interpretation of internal references is completely subject to the kernel subsystem. The most common type of internal references are pointers inside a software cache, such as a radix tree, a double-linked list, and a hash map. A kernel thread can lookup the corresponding object (e.g., by name), then derive an external reference from the internal reference stored in the cache. Such an internal reference does not imply the corresponding object is "in use," it simply means the object "exists." For this reason, they are not refcounted. To prevent memory management errors like UAF, such internal references should be automatically released when the last external reference is released (i.e., when refcount reaches zero). Figure 2 shows an example, when `mgr->kref` reaches zero, `kref_put` would call `amp_mgr_destroy` to remove the internal reference to `mgr`. Without understanding these special patterns, we would face many false positives because internal references clearly violates the refcount consistency invariant.

Unfortunately, the flexibility of internal references also makes it hard to reason about their correctness. For example,

```
1  func_1(){ // local reference scope begins
2    p = kmalloc(sizeof( *p ));
3    refcnt_init(p); // init refcnt = 1
4    list_insert(p, list); // save to heap, escaped
       reference += 1
5  } // local reference scope ends
6  // #escape = 1, #release = 0,  Δrefcnt = 1.
7  // Δrefcnt == (#escape - #release) == 1. correct!
8
9  func_2(){ // local reference scope begins
10   p = list_lookup("name", list);
11   refcnt_get(p); // refcnt += 1
12   use(p);
13   refcnt_put(p); // refcnt -= 1
14 } // local reference scope ends
15 // #escape = 0, #release = 0, Δrefcnt = 0.
16 // Δrefcnt == (#escape - #release) == 0. correct!
17
18 func_3(){ // local reference scope begins
19   p = list_lookup("name", list);
20   refcnt_get(p); // refcnt += 1
21   insert(p, list2);  // escaped reference += 1
22 } // local reference scope ends
23 // #escape = 1, #release = 0,  Δrefcnt = 1.
24 // Δrefcnt == (#escape - #release) == 1. correct!
25
26 func_4(){ // local reference scope begins
27   p = list_lookup("name", list);
28   list_remove(p, list); // released reference += 1
29   refcnt_put(p); // refcnt -= 1
30 } // local reference scope ends
31 // #escape = 0, #release = 1, Δrefcnt = -1.
32 // Δrefcnt == (#escape - #release) == -1. correct!
```

Figure 3: Four types of local reference scopes.

another typical type of internal references are back-pointers. The network namespace object `struct net` is such an example. The `net` object has a refcount and many network related objects contain a back-pointer to the network namespace they reside in. However, most of these pointers are *not refcounted*. The reason is that when a network namespace is torn down, its destructor will automatically tear down those objects reside in the namespace. In other words, those internal references will be automatically released when the object they refer to is freed. But there are exceptions, like the Point-to-Point Protocol (PPP) channels, which can reside in a different namespace. Therefore, such back-pointers will no longer be internal and should be refcounted. When developers simply follow the common pattern in other network subsystems, they introduced an UAF vulnerability (CVE-2016-4805 [24]).

3 Methodology

3.1 Problem Definition

Ideally, we would like to use the invariant *"refcount == # of live reference"* throughout the lifetime of a refcounted object to detect refcount bugs. However, due to the complexity of the kernel code, it is intractable to precisely track live references across different system calls and possibly across different threads with static analysis. We address this problem by tracking the *changes* (Δ) to the refcount and live references within a limited *local references scope* [15] instead. Our key observations are: (1) references stored in globally-visible objects (global variables or heap objects reachable from global variables), or simply *global references*, can be considered as always live because they can be accessed by other syscalls or threads at any arbitrary time; and (2) when a syscall or thread needs to access refcounted objects, the kernel code (at least at the LLVM IR level) will always load global references into local references. Therefore, by maintaining the local invariant *"Δrefcount == Δ#(live reference)"* (where Δ denotes the changes) within the local reference scope, one could also guarantee the global invariant *"refcount == # of live reference"*.

Unfortunately, even after reducing the analysis scope to local ones, if we simply define a refcount bug as a violation of the local invariant at any program location, we will face too many false positives due to *borrowed and stolen references* [15]. For example, in Figure 1, a reference is borrowed when `keyring` is passing to `install_session_keyring_to_cred()` at line 21. However, since the corresponding refcount has already been incremented previously at line 31, i.e., the developer has already indicated that the corresponding object is "in use", it is safe and unnecessary to increment the refcount again. Therefore, to eliminate the false positives caused by borrowed and stolen references, we only perform the invariant check at the end of each local analysis scope.

Unlike other software like the Python extensions written in C [15], the Linux kernel does not have well-defined interfaces that marks the beginning and the end of a local reference scope. We solve this problem by dividing a global reference's lifecycle into four common types (Figure 3):

- Creation: from allocation (e.g., `kmalloc()`) site to local reference(s) to global reference(s) (i.e., `func_1`);
- Usage: from global reference to local reference(s) to usage (i.e., `func_2`);
- Escape: from global reference to local reference(s) to another global reference (i.e., `func_3`);
- Release: from global reference to local reference(s) to the removal of the global reference (`func_4`).

Based on this modeling, we define the beginning of a local references scope when a local reference is derived from a global reference to a refcounted object. Correspondingly, the local reference scope ends when all local references to the refcounted object are released. During this analysis scope, if the reference flows into a global/heap object, we consider the reference as *escaped* to the global scope. Similarly, if a global reference has been removed (e.g., being overwritten by a NULL pointer), we consider the reference as being *released*. With these definitions, we define refcount bugs are follows.

DEFINITION 1 ***Refcount bugs.*** *There is a refcount bug if at the end of a local reference scope, the refcount changes (i.e., Δrefcount) are not the same as the changes to globally visible references (i.e., Δ#(reference) == #escaped - #released).*

Figure 3 showcases the four different types of local reference scope and how the refcount changes invariant is checked. Note that while our approach is conceptually similar to Pungi's [15], analyzing the Linux kernel requires a vastly different and more comprehensive solution.

3.2 Overview

Figure 4 shows an overview of LinKRID and its whole workflow, which involves three phases: a static analysis, a summary-based under-constrained symbolic execution, and the bug detector.

In the first phase of static analysis, we perform two sub-analyses. (1) We need to identify refcounted heap objects and the APIs that manipulate the refcounts. In our static analysis, we take into consideration the multiple layers of abstractions (§4.1). (2) We need to identify the local reference scope to allow the symbolic execution to work on a confined code. To do so, LinKRID constructs flow chains, which correspond to the local reference scopes, to facilitate the checks on the safety invariant (§4.2).

Once we have identified the analysis scope, the next challenge is how to perform the check described in Definition 1. In the second phase, we show how to customize symbolic execution to scale the analysis to a local reference scope. As shown in §2.1, a path-sensitive and context-sensitive analysis is vital to track refcount and reference changes precisely. Two common choices are dynamic analysis, which has a coverage problem; and symbolic execution, which needs to start from fixed entry points like syscall entries or the kernel boot entry and has a path explosion problem. In this work, we opt for under-constrained symbolic execution [8], which takes an arbitrary function and runs it without initializing any of its data structures or doing environmental modeling (§5).

Finally, in the last phase, we need to filter out false positives caused by internal references (§2.2). We solve this problem by extracting Linux-specific conventions that represent the usage of internal references (§6).

4 Static Analysis

Our static analysis takes kernel code in LLVM IR (intermediate representation) as inputs and spots all local references to refcounted objects and computes their lifecycles. It first collects basic refcount information (i.e., *refcounted structures* and *refcount wrappers*), then leverages the retrieved information to construct the data-flow for local reference (referred to as *flow chains* hereinafter). Flow chains encode all local references (to refcounted objects) and their lifecycles, from the time when the reference is created to the time when it is no longer accessible.

4.1 Extract Refcount Information

One of the contributions of our methodology is a systematic recovery of refcount-related structures and functions in the Linux kernel. We collect two types of refcount information in all subsystems of the kernel: (1) *refcounted structures* which embed a reference counter, and (2) *refcount wrappers* which initialize, increase, and decrease reference counts for refcounted structures.

Extracting refcount structures. Despite the fact that the Linux kernel is mostly written in C, it makes broad use of embedded structures to implement kernel objects [2] in a manner similar to object-oriented programming. We leverage this property to identify refcounted structures. This means that refcounted objects can show up with different levels of abstractions. For example, driver-specific objects are usually refcounted through an embedded `kobject` structure, which in turn maintains the refcount through an embedded `kref` structure. However, `kref` is also a wrapper over `struct refcount_t`, which is a wrapper over `atomic_t`. Developers are free to use any abstraction they see fit by *embedding* any of these structures (and even introduce custom ones). Based on this, we collect *refcounted structures* by checking whether a data structure has embedded known refcount structures like `kref` and `refcount_t`.

Extracting refcount wrappers. Next, we collect refcount manipulation APIs (or wrappers) for refcounted structures. Typical refcount wrappers follow the `get()/put()` naming conventions to update the refcount of refcounted structures. Similar to the refcounted structures, refcount manipulation could be done via different levels of abstractions such as `kobject_get/put`, `kref_get/put`, `refcount_inc/dec`, or directly using `atomic_inc/dec`. Developers are free to use any abstraction they see fit and can even introduce their own custom wrappers.

We manually collected 16 lower level refcount manipulation APIs that are generic in Linux. To identify additional wrappers on top of these low level ones, we propose a heuristic-based approach based on following features: (1) it invokes a known *refcount wrapper* once, (2) it transfers reference by function arguments or the return value, and (3) it does not change the number of global references. Using these features, we find 685 custom refcount wrappers in the 4.14 kernel. We further divide them into two groups, one for refcount increment (i.e., `get`-like), and the other for refcount decrement (i.e., `put`-like).

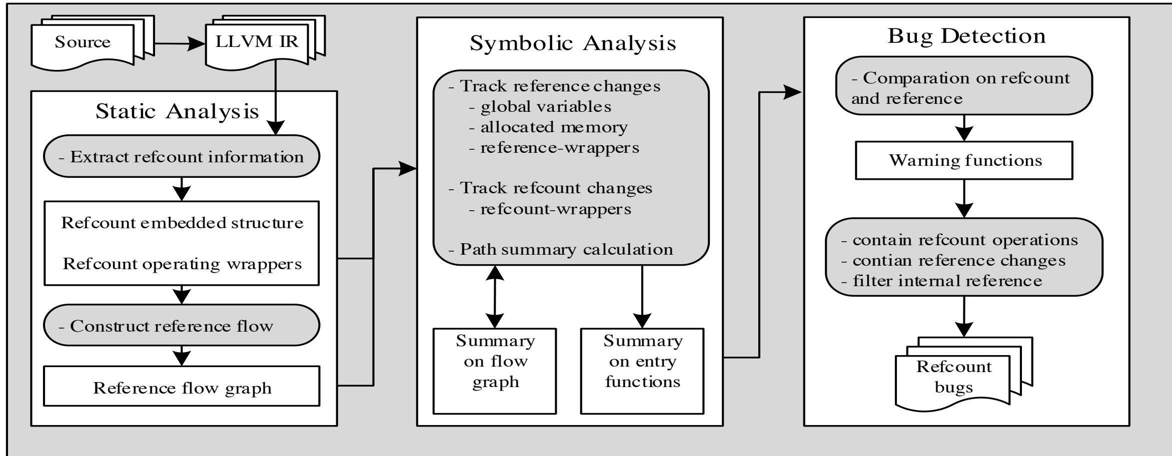

Figure 4: Overview of LinKRID's architecture and its workflow

4.2 Build Flow Chains

Once we have identified *refcounted structures* and *refcount wrappers*, we then construct *flow chains* for local references. As defined in §3, a local reference scope starts when a local reference receives the reference to a refcounted object from a heap allocator or a global reference, and ends when all the local references are released/dead. The challenge to identify the scope is that a local reference could expand its life-scope via data-flow, i.e., by passing the reference to other local variables, function arguments, return value. We use flown chains to "union" these overlapping life-scopes and generate the final analysis scope for our refcount checker.

A flow chain is a call graph containing all functions that need to be analyzed for a specific local reference. These functions are connected by inter-procedural data-flow of the local reference. It is constructed in two steps. In the first step, LinKRID performs a flow-sensitive intra-procedural data-flow analysis to identify three types of function-to-function (i.e., inter-procedural) relations regarding local references.

DEFINITION 2 ***Return-Return relation.*** *Let (f_i, f_j) be a pair of caller and callee functions, we say there is a Return-Return relation $f_i \xrightarrow{lr} f_j$ on a local reference lr, if lr is the return variable of f_i and is also returned by f_j. Figure 5 (a) depicts this relation.*

DEFINITION 3 ***Call-Call relation.*** *Let (f_i, f_j) be a pair of caller and callee functions, we say there is a Call-Call relation $f_i \xrightarrow{lr} f_j$ on a local reference lr, if lr is both an argument of function f_i and f_j. Such relation is showed in Figure 5 (b).*

DEFINITION 4 ***Call-Return relation.*** *Let (f_i, f_j) be a pair of caller and callee functions, we say there is a Call-Return relation $f_i \xrightarrow{lr} f_j$ on a local reference lr, if lr is passed to f_j as a function argument and is live (e.g., be used as return value) after f_j returns. Figure 5 (c) displays the Call-Return relation.*

After collecting the all the three types of inter-procedural relations for each specific local reference, we chain functions involved in these relationships together to derive the flow chain for a given local reference, representing its scope. As an example, Figure 6 (I) shows a set of functions call relations concerning four different local references (the complete code is in Figure 11). Two start in `tunnel_attach_1()` and two start in `tunnel_attach_2()`. Therefore, we construct four flow chains in Figure 6 (II) representing the scopes of the four local references. Once we constructed the flow chains, symbolic execution can take them and perform a much more detailed path-sensitive analysis for bug finding.

5 Symbolic Execution

After static analysis, we have the refcount structures, wrappers, and flow chains. At a high-level, we have all the necessary building blocks to start an under-constrained symbolic execution [27] to perform a path-sensitive analysis and discover refcount bugs.

5.1 Tracking Changes to References and Refcount

As mentioned in §3, we detect refcount bugs by checking the security invariant *"Δrefcount == Δ#(reference)"*. To do so, we need to precisely track the changes to the number of live references and refcount. Given a flow chain, which focuses on a specific local reference, this subsection illustrates how LinKRID traces changes of reference and refcount during analysis.

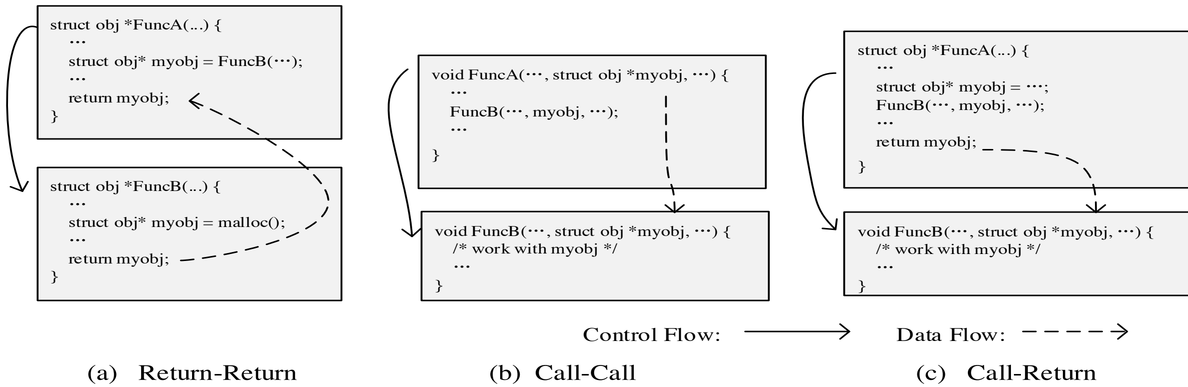

Figure 5: The example of function relations

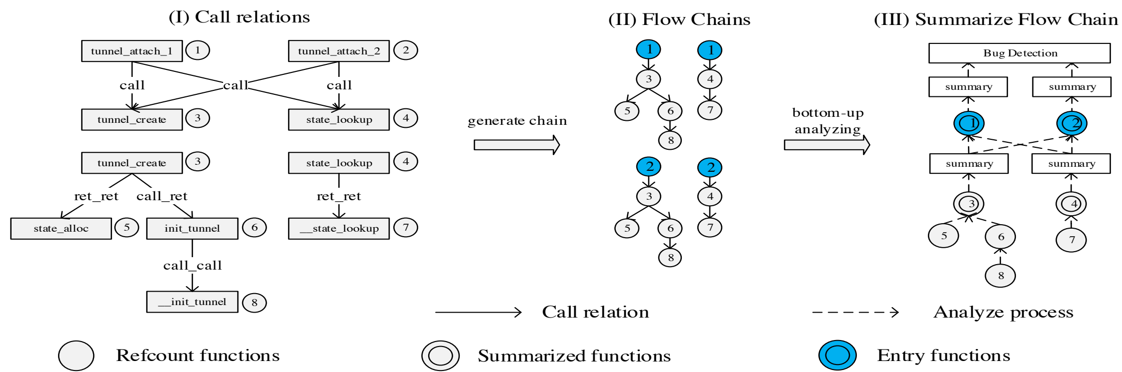

Figure 6: The flow chains and how to do symbolic execution on them.

5.1.1 Tracking Reference Changes

We can compute the changes to the number of live references as: Δ#*(reference)* = *#escaped* - *#released*. To be more clear, the right-hand side of the equation denotes two types of operations: *local reference escape* and *global reference release*. The former creates new global references and the latter removes globally references.

A reference escapes the local scope when it becomes accessible to other syscalls/threads. This happens when a local reference is copied to a globally-visible variable (e.g., a field of a heap object). Similarly, a reference is released when it is no longer accessible by other syscalls/threads. This could happen in two scenarios: (1) when a globally-visible variable that already holds a reference is overwritten, either by another reference or a constant like `NULL`; and (2) when a globally-visible variable that holds a reference is freed. Therefore, we can track the changes to the number of live references by tracking *writes* to non-local variables and *frees*.

RULE 1 ***Reference change through global variables or dynamically allocated memory****. When a local reference is assigned to a globally-visible object, this reference is escaped and a global reference is created . If a global reference is overwritten by another reference or NULL, or the object that contains the reference is freed, then the reference is considered released.*

Table 1: Summarized kernel APIs that change the number of live references. * means there are several APIs with the same name prefix.

Type	Functions
Reference Escape	`list_add*, hlist_add*, idr_alloc`
Reference Release	`list_del*, hlist_del*, idr_remove`

While most writes and frees can be tracked at the LLVM IR level, there are a few frequently used kernel APIs that are not easy to analyze (e.g., due to the use of assembly code for atomic operations and memory barriers). For example, `list_add()` can be used to add a refcounted object into a double-linked list, implying an escape of the reference. Similarly, `list_del()` can be used to remove a live reference to a refcounted object from a double-linked list, implying a release of the reference. To handle these APIs, we manually collected and summarized their effects (Table 1).

RULE 2 ***Reference change through reference wrappers****. When APIs listed in Table 1 are invoked, reference changes should be updated according to our modeling.*

5.1.2 Tracking Refcount Changes

LinKRID tracks refcount changes by tracing invocations of refcount wrappers (§4.1). While this solution is straightforward and effective, some refcount wrappers will not always process successfully. When these refcount wrappers fail, they return specific values (e.g., a `NULL` pointer) to inform their callers. For example, the wrapper `kref_get_unless_zero()` will not increase the refcount when the current refcount is `0`, and will return `0` to indicate the failure in increment; its caller should check the return value and handle the error. If we simply regard an invocation of `kref_get_unless_zero()` as an unconditional refcount increment, we will introduce false positives. Fortunately, the symbolic execution employed in LinKRID is path-sensitive by design and is therefore capable of handling successful and erroneous paths. To leverage this capability, we use symbolic path summary (see §5.2 for more details) of the wrappers to indicate what return values are associated with refcount increment (i.e., success) and what return values indicate errors.

RULE 3 ***Refcount change through successful refcount-wapper***. *When a refcount wrapper successfully operates on a reference, the reference's corresponding refcount should be changed.*

5.1.3 Tracking Changes in Asynchronous Methods

Asynchronous mechanisms like work queue and timer are widely used in the Linux kernel, which imposes additional challenges in tracking reference and refcount changes. When a local reference is used as parameters to construct an asynchronous task, the corresponding refcounted object must not be freed (i.e., its refcount cannot drop to 0) before the asynchronous callback function is invoked. One possible approach is to consider the reference as *escaped* when passed to an asynchronous task thus demands a refcount increment. Similarly, there should be a corresponding refcount decrement inside the asynchronous callback function, as when the callback function has finished using the reference, it is no long live. However, after constructing an asynchronous task, the current syscall/thread can also decide to *wait for the completion of the task while holding the reference*. In this case, the reference is effectively borrowed to the asynchronous task hence do not require additional changes to the refcount. To correctly handle these different cases, we consider callback functions as invoked *synchronously* (i.e., as normal function calls) during our analysis. Specifically, LinKRID analyzes those callback functions individually and adds their summaries (described in §5.2) into the current flow chain when encountering an asynchronous register function like `queue_work` and `mod_timer`. For the deregister functions like `cancel_work` and `del_timer`, LinKRID will create another path where the callback function is treated as not being invoked. Because these register/deregister functions may also fail and return errors, we use the same as simulation on refcount wrappers to handle successful and failed cases.

RULE 4 ***Refcount change through successful asynchronous functions***. *When an asynchronous callback function is registers/deregisters successfully, its summary will be included/removed from the current flow chain.*

5.2 Summary-based chain analysis

As highlighted in Figure 1, precise detection of refcount bugs demands path-sensitive analysis. In this subsection, we describe how LinKRID use under-constrained symbolic execution [27] to analyze flow chains symbolically to identify potential buggy paths. Despite limiting the analysis scope to a single flow chain (i.e., a local reference scope), there still could exist many functions invocations along the flow chain that could cause path explosion. We mitigate this problem from three aspects. First, we only perform symbolic execution for functions defined in the same source code file. For external functions, we assume they will return unconstrained symbolic values. Second, to avoid repeatedly analyzing the same function, we propose a lightweight symbolic execution paradigm by summarizing functions. The summary of a function encapsulates how references and refcount change under different execution paths. Finally, we set a maximum of paths to be explored for each reference.

5.2.1 Path Summary Calculation

Formally, the summary of each path in a function is a 5-tuple.

$$Sum = (lr, escape, release, \Delta refcnt, retval)$$

where, lr is the local reference that appears in the function. $escape$ is a set of instructions where lr is propagated to global references. $release$ is a set of instructions where global references that are removed. Based on these two sets, LinKRID can calculate the Δ#reference of the corresponding refcounted object (the left-hand side of the equation as in Definition 1). Note that we save the instructions corresponding to the escape and release operations instead of numbers to help filter false positives (e.g., internal references, see §6 for more details). $\Delta refcnt$ records refcount changes to the corresponding refcounted object. $retval$ is the path return value. Regarding the execution path, we only save its return value, which can influence the control-flow (i.e., path feasibility) of the caller.

To compute path summaries of a function, we run the under-constrained symbolic execution on the function by maintaining and forking "states" as necessary. Similar to the definition of summary, a state in the symbolic execution is a 7-tuple that contains the intermediate result during execution.

$$State = (ip, con, escape, release, \Delta refcnt, vmap, retval)$$

Table 2: `symbolicexec(func)`:How instructions are evaluated symbolically.

Instruction	Symbolic execution
Case 1. x =uninitial	`make_symbolic`
Case 2. $x = u$	$vmap[x] = vmap[u]$
Case 3. $x = u$, where u is local reference, and x is a global variable or a heap variable	$vmap[x] = vmap[u]$ and update $\#escape(vmap[u])$ and $\#release(vmap[x])$ according RULE1
Case 4. `if` *condition*, l_1, l_2	if l_1 is taken, $cons = cons \wedge condition$, otherwise $cons = cons \wedge (\neg condition)$
Case 5. `call` f on local reference u, where f is a reference-wrapper	update $\#escape(vmap[u])$ or $\#release(vmap[u])$ according RULE2
Case 6. `call` f on local reference u, where f is a refcount-wrapper or asynchronous functions	update $\Delta refcnt(vmap[u])$ according RULE3 or RULE4
Case 7. `return` u	$ret = vmap[u]$

where ip points to the next instruction to be executed. *cons* are path constraints. *escape*, *release* and Δrefcnt_map are the same as in the path summary. *vmap* is a map from variables to symbolic values. *retval* is the return value.

Table 2 shows how instructions are executed in an informal manner. Briefly, cases 3, 5, and 6 are related to rules in §5.1. Others, including 1, 2, 4, and 7, are how UC-KLEE handles typical instructions. For example $vmap[x] = vmap[u]$ means the value of key x in *vmap* is set to the value of key u in *vmap*. Updating the instruction pointer ip is straightforward.

When LinKRID encounters a call instruction, it first checks whether the target function has summaries or not. If so, its summaries would be merged into current state; otherwise, this function would be symbolic analyzed directly. Besides, if a function calls itself (i.e., recursive invocation), as there are no summaries for current analyzed function, LinKRID will only execute the recursion once. In other words, the recursive depth is limited to one.

The execution will generate a set of new states, each of which will be executed independently. Each time a return instruction is executed, a summary is created to record the reference changes, the refcount changes, and the expression of the return value from the state and added into the function's summaries *sums*.

5.2.2 Summary-Based Analysis of Flow Chains

To avoid analyzing the same function repeatedly across different flow chains, we perform a bottom-up and iterative analysis. First, we merge all flow chains in the same source code file to construct the call graph over functions that need to be analyzed. In each iteration, we first put all leaf nodes in the current call graph (i.e., functions that has no callees to be analyzed) to a work list. For each function in the work list, we first check whether it has multiple callers (i.e., involved in multiple flow chains); if so, we invoke `symbolicexec(f)`, as defined in Table 2, to compute its summary. Then we checked if it is an entry function; if so, we will always perform symbolic execution with it. For other functions, we simply mark them as "analyzed" without actually performing symbolic execution. The reason is that using summaries will inevitably introduce approximations, which could cause false positives. To minimize the imprecision, we only create summary for functions that will be invoked multiple times. For functions that is invoked only once, they will be analyzed (inter-procedurally) when generating summaries or when analyzing the entry functions. Once we have finished processing the work list, we have a new set of leaf nodes as previous set of leaf nodes have already be analyzed.

Figure 6 III shows which functions need to be summarized for the flow chains illustrated in §4.2. In this example, the four flow chains are analyzed together where node1 and node2 are entry functions; and node5, node7, and node8 are the initial leaf functions. When walking the call graph from bottom up, LinKRID will compute summaries for node3 and node4 as they are invoked by multiple callers (both node1 and node2). Node5, node6, and node8 will be analyzed when summarizing node3; and node7 will be analyzed when summarizing node4. Finally, the entry functions are being processed. In this example, they (node1 and node2) will be analyzed using summaries from node3 and node4.

6 Bug Detection

Given a flow chain $\mathcal{C}$, when symbolic execution finished, we can obtain the per-path summary at the end of the local reference scope (i.e., the outmost caller). Now we check whether the invariant Δ*refcount* == *#escaped* - *#released* holds. That is, we will report a potential bug whenever there exists a local reference lr in a *Sum* satisfying the following condition: ($|escape_{lr}|$, $|release_{lr}|$ calculate the number of elements in a set):

$$\Delta refcount_{lr} \neq |escape_{lr}| - |release_{lr}|$$

However, this result does not take into account the internal reference design pattern mentioned in §2.2 thus may still result in a substantial number of false positives. In addition, there are ambiguities in the presence of embedded structures when associating the refcount change with reference change. For these reasons, LinKRID treats any detected violations as only candidates of warnings at this point.

6.1 Identifying Internal References

We first discuss how we address the internal reference problem. As described in §2.2, fundamentally, internal references are references that do not need to be refcounted because they are tracked and released through a separate mechanism. If we mistakenly require refcount changes for internal references, we will end up with false alarms. At the moment, LinKRID handles two internal reference patterns, both of which are described in §2.2.

The first pattern we handle is the internal references that are can be viewed as separate/additional references that are

released automatically when the refcount reaches zero. For example, in Figure 2, `mgr` is removed by `list_del()` when its refcount is decreased to zero. In other words, LinKRID analyzes the callback function registered in refcount wrappers (e.g., `kref_put()`) to discover such internal references.

The second pattern is a more generalized version of the first, where internal references may not be released at a specific known point (e.g., when refcount reaches zero). Instead, they are released in domain-specific manners. In the example described in §2.2, the `net` struct representing network namespace can be freed on demand (from a syscall). When such an object is freed, it will automatically and forcefully free all the network objects that have a back-pointer to it. Unfortunately, it can be challenging to identify all possible places where an internal reference is released. Therefore, we develop a generic heuristic for this. The idea is simple: when a particular reference (e.g., a pointer in a struct) is never refcounted anywhere throughout the kernel, we consider the reference an internal one. While this does help with reducing false positives, we find that it will cause some false negatives as well (which will be discussed in §7.3). Finally, we also note that there are even more complex internal reference patterns (mixed usage between internal and external) and we will discuss how they can be handled in §7.3.

6.2 Determining the Association of Refcount and Reference Changes

In addition to handling internal references, there is one more complication regarding the application of the invariant listed earlier. Specifically, $\Delta refcount_{lr}$, $escape_{lr}$, and $release_{lr}$ are implicitly assumed to operate on the same refcounted object because they share the same local reference (i.e., belong/point to the same object). However, in reality, validating this assumption is not always straightforward. As a real example in the kernel, struct `gdm` embeds a refcounted struct `tty_port`. The `tty_port->kref` field can technically represent the refcount of `tty_port`. However, due to complex pointer arithmetics (e.g., `container_of`), it could be unclear whether a pointer of type `tty_port*` actually points to a standalone `tty_port` object or a `gdm` object. Consequently, we may not be able to associate the refcount with `gdm`. In practice, if a local reference of `gdm` escapes, we may observe the refcount change is operated through a local reference of `tty_port` instead. As a result, we may end up with two false positives. One is the missing reference count change when a local reference of `gdm` escapes. The other is the missing reference change when the refcount of `tty_port` is incremented.

Fundamentally, this is an alias analysis problem of embedded structures (i.e., whether two pointers point to the same memory object), which is known to be hard to solve in the presence of pointer arithmetic and type casting (e.g., integer to pointer and pointer to integer). To infer such aliasing and filter such false positives, we apply the following heuristic:

```
join_session_keyring path record:
Error Path:
security/keys/process_keys.c L780, get keyring refcnt +
  1 by find_keyring_by_name
security/keys/process_keys.c L781, if condition is false
security/keys/process_keys.c L791, if condition is false
security/keys/process_keys.c L794, if condition is true
...
Summary: <keyring, escape=∅, release=∅, Δrefcnt=1, return=0>
Result : refcnt error in keyring
```

Figure 7: Example bug report for the erroneous path in Figure 1.

if two local references (e.g., `gdm` and `tty_port`) happen to trigger two warnings in the same function —one leads to a missing refcount change and the other leads to a missing reference change, and one local reference is "embedded" in another (e.g., `tty_port` is embedded in `gdm`), we will then consider the refcount change of one local reference is actually associated with the reference change of another. As the heuristic may also lead to false negatives, we will discuss possible improvements in handling embedded structures in a more systematic manner in §7.3.

6.3 Bug Reporting

After the filter of internal reference and the association of refcountand reference changes, the remain warnings would be regarded as bugs for further manual verification. Figure 7 shows an example bug report for the erroneous path in Figure 1. To assist manual verification, the bug report includes path information (e.g., branch direction) and the detail refcount information (e.g., refcount get/put) with corresponding source code line.

7 Evaluation

LinKRID is implemented based on the static analysis framework from KENALI [31] and symbolic execution engine KLEE [3, 33]. LinKRID 's source code contains 3.5k lines of C++ code and 2.7k line of python code. We applied LinKRID on the Linux kernel v4.14 release, compiled with LLVM 3.9 with `make defconfig` and `make allyesconfig`, respectively.

7.1 Basic Statistics

Table 3 shows the experiment statistics in different compilation options. In the static analysis phase, we extracted 85 and 445 refcount embedded structures, 149 and 685 refcount increment/decrement wrappers in the Linux kernel v4.14 compiled with `defconfig` and `allyesconfig`, respectively. Based on the above, we extracted 16,155 and 54,731 functions respectively which manipulate those refcounted structures. Furthermore, LinKRID outputs 4,063 and 12,075 flow chains respectively. During symbolic execution phase, we set the maximum

Table 3: Statistics in experiments

Phase	Description	Data	
		defconfig	allyesconfig
Static Analysis	refcounted structures	85	445
	refcount wrapper	149	685
	related functions	16155	54731
	flow chains	4063	12075
Symbolic Execution	summaries	2964	9419
Bug Detection	BUGs	31	118

explored paths mentioned in §5.2 to 1000, LinKRID generated 2,964 and 9,419 summaries respectively.

We ran our experiments on two virtual machines, both of which are configured with an Intel Xeon E3-1220 CPU and 32GB RAM. The operating system is Ubuntu 16.04 LTS x64. Our lightweight static analysis took 1 hour to finish the analysis of a whole Linux kernel (with `allyesconfig`). For symbolic execution, as there is no dependency among separate flow chains, we run the symbolic execution in 8 different processes in parallel. In this phase, LinKRID took 192 hours to process all the flow chains.

7.2 Detected Bugs

Finally, LinKRID reported in total 209 refcount errors under allyesconfig. Out of them, we manually confirmed 118 to be true positives and 31 bugs of them exist under defconfig. Among the 118, 31 have already been patched in the newer versions, 87 are previously unknown. For the 87 bugs, we submitted bug reports to kernel maintainers for verification. 47 of the submitted bugs were confirmed, among which 11 were patched with our patches. More details can be seen in Table 4. To manually analyze all the bugs, it took two researchers, a total 72 man-hours. After that, we manually checked our results against a more recent kernel version v5.10, and found that 43 of 87 bugs exist in the newer version, which are marked by '*' in Table 4.

Responsible Disclosure. We have disclosed all the unpatched vulnerabilities following the guidelines of the Linux community[1]. Among the 43 bugs that still exist in the newer version, 12 bugs have been fixed with patches from us (marked as 'A' in Table 4); 11 bugs have been confirmed and we are working with corresponding maintainers to forge patches (marked as 'C' in Table 4); 12 bug reports have been sent to maintainers and waiting for reply (marked as 'S' in Table 4). For the rest 8 bugs (marked as '/' in Table 4), we cannot find the corresponding sustainers in Linux kernel's `MAINTAINERS` list; and the modules, in which bugs were found, are likely deprecated.

We also noticed that almost all these bugs occur in error handling paths, which is a common issue in the development of the Linux kernel [14]. The result shows that LinKRID can detect these buggy paths precisely and effectively. We divide these bugs into two categories based on fix strategies: improper refcount changes and improper reference changes.

[1] https://www.kernel.org/doc/html/latest/process/submitting-patches.html

Improper refcount changes. LinKRID reported 52 new bugs that are related to improper refcount changes, with two subtypes: missing refcount decrement (i.e., over-counting) and redundant refcount decrement (i.e., under-counting). The former may cause memory leak and is usually fixed by adding a corresponding refcount decrement. Figure 8 shows a missing refcount decrement bug fixed by inserting refcount decrement before line 6. The latter may cause use-after-free and is usually fixed by avoiding the redundant decrement. Figure 9 depicts a redundant refcount decrement bug, where if line 7 is executed, the `kref_put` at line 13 would be a redundant decrement and may cause the object `orig_io_req` be freed prematurely when it is still in use. The distribution of improper refcount changes cause is heavily skewed towards missing refcount decrement. Our reports showed 50 new bugs are caused by missing refcount decrement. One explanation could be that redundant refcount decrement bugs are more likely to trigger use-after-free, thus are more likely to be captured during kernel fuzzing when kernel address sanitizer (KASAN) is enabled.

Improper reference changes. Another typical bug found by LinKRID is the improper reference changes. All of such bugs due to missing reference releases. That is, an object's refcount is decreased without releasing the reference (e.g., setting the reference to NULL), which may be used somewhere even after it is freed and cause use-after-free. Such bugs are usually fixed by releasing the reference to avoid being used accidentally and erroneously. Figure 10 displays a case where `usb_free_urb` is invoked at line 5 without release the reference at `dvb->bulk_urb`, which is accessed at line 13 in function `tm6000_stop_stream` and caused use-after-free. This is fixed by assigning a NULL pointer to `dvb->bulk_urb` after the refcount decrement.

```
1  int kobject_rename(struct kobject *kobj, const char *
       new_name) {
2    int error = 0;
3    const char *devpath = NULL;
4    kobj = kobject_get(kobj);
5    if (!kobj) return -EINVAL;
6    if (!kobj->parent) return -EINVAL;
7    devpath = kobject_get_path(kobj, GFP_KERNEL);
8    if (!devpath) {
9      error = -ENOMEM;
10     goto out;
11   }
12 out:
13   kobject_put(kobj);
14   return error;
15 }
```

Figure 8: A missing refcount decrement caused by improper error handling, which is fixed by adding a refcount decrement before return.

```
1  int bnx2fc_send_rec(struct bnx2fc_cmd *orig_io_req) {
2    struct bnx2fc_els_cb_arg *cb_arg = NULL;
3    int rc;
4    cb_arg = kzalloc(...);
5    if (!cb_arg) {
6      rc = -ENOMEM;
7      goto rec_err;
8    }
9    kref_get(&orig_io_req->refcount);
10   cb_arg->aborted_io_req = orig_io_req;
11 rec_err:
12   if (rc) {
13     kref_put(&orig_io_req->refcount,...);
14     kfree(cb_arg);
15   }
16   return rc;
17 }
```

Figure 9: A redundant refcount decrement detected by LinKRID.

```
1  static int tm6000_start_stream(struct tm6000_core *dev
       ) {
2    dvb->bulk_urb = usb_alloc_urb(0, GFP_KERNEL);
3    dvb->bulk_urb->transfer_buffer = kzalloc(size,
       GFP_KERNEL);
4    if (dvb->bulk_urb->transfer_buffer == NULL) {
5      usb_free_urb(dvb->bulk_urb);
6      return -ENOMEM;
7    }
8    return ret;
9  }
10 static void tm6000_stop_stream(struct tm6000_core *dev
       ) {
11   struct tm6000_dvb *dvb = dev->dvb;
12   if (dvb->bulk_urb) {
13     usb_kill_urb(dvb->bulk_urb);
14   }
15 }
```

Figure 10: A missing reference release, which is fixed by adding a release after refcount decrement.

7.3 False Positives and False Negatives

False Positives. As presented in §7.2, LinKRID has a false positive rate of 43% ((209 - 118) / 209). Here, we investigate the root causes and discuss potential solutions.

Non-standard refcount (mixed use of internal and external reference). Interestingly, we find that some developers may define their own refcount rules and mix the use of internal and external reference. For example, The struct `fib_lookup_arg` has a pointer member to a refcounted struct `fib_rule` and a flag member. The latter dictates whether the former reference needs to be refcounted. If the *Flag* is set to `FIB_LOOKUP_NOREF`, the reference change does not need be refcounted, effectively making it an internal reference. Otherwise, it is treated as an external reference. Given that our heuristic developed in §6.1 do not recognize this special pattern of *conditional internal references*, we will misclassify it as an external reference (as it does seem to be refcounted sometimes in the kernel), and hence the false positive. Surprisingly, these cases contribute to 66% of all false positives.

To properly handle such cases, we will need to model this special pattern by actively looking for such "flag" members whose values correlate with the presence or absence of refcount changes. For example, if refcount changes always happen under one value of the flag (but never happen under a different value) and vice versa, then we will infer that it is a conditional internal reference.

Out-of-scope reference changes. We find that some warnings of missing reference releases can be false positives. This is because the reference releases actually do happen (e.g., set to NULL) but they happen outside of our local analysis scope. In other words, the flow chains we construct are based on the local reference change which may miss a release of a global reference that happens, for example in the caller of the functions in the flow chain (after they return). Such cases account for 18% of the false positives.

To eliminate such false positives, we need to expand the analysis scope to consider not only the life-scope of local references but also the life-scope of global references. If we can track both, we can construct the flow chain by considering the "union" of the two scopes. However, this can be potentially expensive and challenging. First of all, we do not know when a global reference will be released (e.g., set to NULL). In fact, it may never happen, forcing us to exhaustively traverse the call graph all the way back to the syscall entry point. Second, since global references involve heap objects, it can be challenging to analyze statically, e.g., in terms of alias analysis. Nevertheless, this is a clear avenue for improvement.

Other causes. Besides the above, false positives can also arise due to some rarely used programming patterns that we currently do not recognize. For example, we find sometimes the refcount wrappers can be in the form of `refcount_set(refcnt, refcount_read(refcnt) + 1)` instead of `refcount_inc(refcnt)`. Another example is on reference releases — sometimes developers set a pointer to `~0` instead of `NULL`. These cases account for the remaining 16% of the false positives.

These cases are mostly trivial and can be solved by augmenting our modeling of the Linux kernel.

False Negatives. As we mentioned in §6, to avoid excessive false positives, LinKRID will drop errors it believes to be caused by internal references. However, it is also possible that a reference is supposed to be external but the programmers simply forgot to refcount it — CVE-2016-4805 [24] is such an example. In such cases, LinKRID will believe a reference is internal and ignore any warning associated with it. In addition, LinKRID currently does not verify the correctness of the internal reference use and thus lead to false negatives as well. This can happen when developers forget to remove the internal reference properly when they are supposed to.

A possible solution to the above is to distill more fine-grained internal reference patterns and check them more precisely as opposed to using generic heuristics. For example, we can check whether internal references are correctly removed

at specific points according to their typical usage patterns. If not, it is either not an internal reference, or it is a buggy use of internal reference (we should report both cases).

Another problem is the association of refcount and reference changes as described in §6.2. Our current solution only heuristically infers the association in a post-processing manner, which do lead to some false negatives. A more principled approach would be to adopt a more accurate aliasing analysis that can handle arbitrary pointer arithmetic and struct embedding in the Linux kernel.

7.4 Security Impact of Found Bugs

In this subsection, we are interested in understanding the security impact of these reported bugs. While it is time-consuming to triage these bugs, we did randomly sample 22 of the 87 bugs to analyze their exploitability and understand their security impact.

Among these 22 bugs, 12 can cause memory leak and 10 can cause UAF. Furthermore, all of the ten UAF cases involve objects that contain function pointers, indicating that they can likely lead to control hijacking primitives. In addition, we evaluate the following metrics of these bugs in terms of their exploitability: triggering condition (how easy it is to trigger the bug), reachability of the bug (whether it is possible to trigger the bug from userspace), and permission requirement (does it require a special uid or Linux capability). In terms of triggering conditions, 18 bugs can be triggered only when the memory allocation fails, 3 of which additionally require refcount overflows in order to trigger UAF. We note that there are indeed prior attacks that successfully exploit vulnerabilities by exhausting system memory and therefore it is not impossible to trigger memory allocation failures [34, 39]. However, the latest Linux kernel has included hardening against refcount overflows and therefore the 3 overflow cases are unlikely exploitable [14]. The remaining 3 bugs do not seem to require any extreme conditions to trigger. In terms of reachability, 6 can be reached from the userspace (i.e., through syscalls) and all of them happen to be UAF bugs. 10 are not reachable from userspace. Instead, 9 of them are called upon driver/device registration or system initialization. 1 of them is reachable from hardware interrupt. The remaining 6 are unclear. Finally, in terms of permission requirements, 3 require Linux capability, e.g., `CAP_NET_ADMIN`. The detailed breakdown of each individual bug is presented in Table 5 in appendix. Overall, we conclude that at least 3 of the UAF bugs are highly exploitable as they are reachable from userspace, possible to trigger (do not require refcount overflow), and do not require any special permission.

To give some examples, Figure 8 shows a bug that can cause UAF. If it is triggered multiple times, the refcount would overflow to zero and release the object incorrectly. Figure 9 displays a bug triggered only when memory allocation fails.

8 Related Work

Finding refcount bugs. As mentioned in §1, there are two categories of research on finding refcount bugs. An intuitional and rigorous way to detect refcount bugs is to check the refcount invariant (i.e., *refcount = #(live reference)*). Referee [7] uses compositional model checking to verify the refcount correctness. It transforms the verification on reference counting of unbounded resources on unbounded threads to arbitrary resources on an single thread by assuming that resources are arranged as an array and uniformly managed with the same piece of code. Then, Referee symbolically verifies an arbitrary resource is accessed under a positive reference count. CPyChecker [19] is a GCC plug-in that detects the errors in Python's native extension modules, including reference counting errors. Pungi [15] also aims to find refcount errors in Python's native extension modules. Both of them rely on the invariant that, the changes to refcount must matches the changes to reference number. CPyChecker applied this idea to a single function without inter-procedure analysis. Pungi extended it to inter-procedural analysis, used affine abstraction and analyzed the Static Single Assignment (SSA) form of programs. In addition, the affine abstraction on some wrappers of python refcount APIs relies on manually constructing. Naively applying affine abstraction to Linux kernel will face many challenges such as various refcount APIs and exclusive using contexts. Another method is using heuristics to find a specific type of refcount bugs. RID [20] uses inconsistent path pair (IPP) to detect developer's misunderstand of refcount APIs in Linux kernel. Unfortunately, RID could not find refcount bugs in the common case, like violating of refcount invariant.

Symbolic execution for bug detection. With the recent advances in SMT solver, symbolic execution has been widely applied on detecting bugs in software systems, including Linux applications [38], embedded firmware [4], Android applications [37], and the Android framework [17]. However, a major obstacle that prevents symbolic execution from getting even wider application is the path explosion problem. Although there are efforts on mitigating the problem, e.g., by using methods based on compositionally [9], abstraction refinement [18], interpolation [5, 10, 11, 21], parallelization [6, 12, 26, 30, 32], and machine learning [16, 29], path explosion remains a bottleneck in scaling symbolic execution to larger applications.

Given the code complexity, applying symbolic execution to analyze the entire OS kernels is challenging. To make tradeoffs between scalability and path coverage, under-constrained symbolic execution is a promising method for Linux kernel bugs detection. UC-KLEE [27] employs this method to find bugs and verifies patches in Linux kernels, specifically to memory leaks, uses of uninitialized data, and unsanitized uses of user inputs. DEADLINE [36] applies symbolic execution to the detection of double-fetch bugs in the Linux kernel.

9 Conclusion

In this paper, we proposed a scalable and effective approach to detect refcount bugs in the Linux kernel. Our approach has two key technical innovations: (1) an inter-procedural data-flow analysis to construct the local reference lifetime, which narrows down the analysis to a reasonable scope, thus scaling expensive analyses to Linux kernel; (2) a summary-based symbolic execution to provide path-sensitive analysis as well as avoiding repetitive computation on tracking reference and refcount changes. Our prototype LinKRID found 87 new refcount bugs in Linux 4.14, demonstrating the scalable and effective of our approach.

Acknowledgments

We would like to express our gratitude to our shepherd Dr.Hamed Okhravi and the anonymous reviewers for their helpful advice on the paper. We also thank the Linux kernel developers who gave useful feedback to us. This work was supported in part by National Key Research and Development Program of China under Project No. 2018YFB0805000.

References

[1] Neil Brown. Linux kernel design patterns - part 1. https://lwn.net/Articles/336224/, 2009.

[2] Neil Brown. Object-oriented design patterns in the kernel, part 2. https://lwn.net/Articles/446317/, 2011.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI 08)*, volume 8, pages 209–224, 2008.

[4] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2016.

[5] Duc-Hiep Chu and Joxan Jaffar. A complete method for symmetry reduction in safety verification. In *International Conference on Computer Aided Verification (CAV)*, pages 616–633, 2012.

[6] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009.

[7] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 352–367. Springer, 2009.

[8] Dawson Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 1–4, 2007.

[9] Patrice Godefroid. Compositional dynamic test generation. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–54, 2007.

[10] Joxan Jaffar, Andrew Santosa, and Razvan Voicu. An interpolation method for CLP traversal. In *International Conference on Principles and Practice of Constraint Programming*, pages 454–469, 2009.

[11] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, pages 297–303, 2008.

[12] Moonzoo Kim, Yunho Kim, and Gregg Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. pages 340–349, 2012.

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

[14] Greg Kroah-Hartman. Add overflow protection to kref. https://lkml.org/lkml/2012/2/24/345, 2012.

[15] Siliang Li and Gang Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *European Conference on Object-Oriented Programming*, pages 80–104. Springer, 2014.

[16] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 554–559. IEEE, 2016.

[17] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation. *IEEE Transactions on Mobile Computing*, 2019.

[18] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 416–426, 2007.

[19] D. Malcom. A static analysis tool for cpython extension code. https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html, Accessed: 2020-05-08.

[20] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 531–544, 2016.

[21] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification (CAV)*, pages 104–118, 2010.

[22] MITRE. CVE-2016-0728. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0728, 2016.

[23] MITRE. CVE-2016-4557. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4557, 2016.

[24] MITRE. CVE-2016-4805. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4805, 2016.

[25] MITRE. CVE-2017-11176. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11176, 2017.

[26] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26, 2008.

[27] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

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

[29] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[30] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 523–536, 2012.

[31] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, feb 2016.

[32] Matt Staats and Corina S. Pasareanu. Parallel symbolic execution for structural test generation. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 183–194, 2010.

[33] The KLEE Team. KLEE. http://klee.github.io/, Accessed: 2020-05-08.

[34] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.

[35] 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[36] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[37] Chao Chun Yeh, Han Lin Lu, Chun Yen Chen, Kee Kiat Khor, and Shih Kun Huang. Craxdroid: Automatic Android system testing by selective symbolic execution. In *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, pages 140–148. IEEE, 2014.

[38] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Transactions on Software Engineering*, 44(1):25–43, 2017.

[39] Hang Zhang, Dongdong She, and Zhiyun Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1663–1674, 2016.

A Appendix

```
1  struct state {
2      refcount_t refcnt;
3      struct hlist_node node;
4      struct state* tunnel;
5  }
6
7  LIST_HEAD(my_list)
8
9  int tunnel_attach_1(struct state *x) {
10         int err = 0;
11         struct state *t;
12
13         t = state_lookup();
14         if (!t) {
15                 t = tunnel_create();
16                 if (!t) {
17                         err = -EINVAL;
18                         goto out;
19                 }
20                 refcount_inc(&t->refcnt);
21         }
22         x->tunnel = t;
23 out:
24         return err;
25 }
26
27 int tunnel_attach_2(struct state *x) {
28     int err = 0;
29         struct state *t;
30
31         t = state_lookup();
32         if (!t) {
33                 t = tunnel_create();
34                 if (!t) {
35                         err = -EINVAL;
36                         goto out;
37                 }
38                 refcount_inc(&t->refcnt);
39         }
40         x->tunnel = t;
41 out:
42         return err;
43 }
44
45 struct state *state_lookup() {
46      return __state_lookup();
47  }
48 struct state *__state_lookup(){
49         struct state *x;
50
51         hlist_for_each_entry(x, my_list, node) {
52                 if (cond(x))
53                         continue;
54                 if (!refcount_inc_not_zero(&x->refcnt))
55                         continue;
56                 return x;
57         }
58         return NULL;
59 }
60
61 struct state *tunnel_create() {
62         struct state *t;
63
64         t = state_alloc();
65         if (!t)
66                 goto out;
67
68         if (init_tunnel(t))
69                 goto error;
70         hlist_add(&t->node, my_list);
71 out:
72         return t;
73
74 error:
75         kfree(t);
76         t = NULL;
77         goto out;
78 }
79
80 struct state *state_alloc() {
81         struct state *x = kzalloc(sizeof(struct state),
       GFP_ATOMIC);
82         if (x)
83                 refcount_set(&x->refcnt, 1);
84         return x;
85 }
86
87 int init_tunnel(struct state* t) {
88     return __init_tunnel(t);
89 }
90
91 int __init_tunnel(struct state* t) {
92     ...
93 }
```

Figure 11: Running example adapted from real function `ipcomp_tunnel_attach` and `ipcomp6_tunnel_attach` in Linux kernel 4.14.0

Table 4: List of bugs discovered by our tool in the Linux kernel 4.14, '*' means the bug exists(ed) in the Linux kernel version 5.10. MRD, RRD and MRR indicate bugs cause: missing refcount decrement, redundant refcount decrement and missing reference release respectively. A - patches from us are accept. C - bug reports are confirmed by developers. S - bug reports are submitted, waiting for replies. P - bugs are already patched. / - the corresponding module are no longer maintained.

#	Function	Cause	Status	#	Function	Cause	Status
1	kobject_rename*	MRD	A	60	watchdog_cdev_register	MRR	S
2	amdgpu_cs_process_fence_dep*	MRD	A	61	bfusb_send_bulk*	MRR	S
3	ttm_bo_add_move_fence*	MRD	A	62	nvmet_fc_ls_disconnect*	MRR	S
4	blk_register_queue*	MRD	A	63	usbnet_start_xmit*	MRR	S
5	__blk_mq_register_dev*	MRD	A	64	fb_open	MRR	S
6	rpc_clnt_add_xprt*	MRD	A	65	mcs_net_open	MRR	S
7	write_parport_reg_nonblock*	MRD	A	66	vnt_start*	MRR	S
8	tty_lookup_driver*	MRD	A	67	pn533_usb_probe	MRR	S
9	bnx2fc_send_srr*	RRD	A	68	send_mpa_req	MRR	S
10	bnx2fc_send_rec*	RRD	A	69	port100_probe	MRR	S
11	imon_probe*	MRR	A	70	bcm203x_probe	MRR	S
12	airspy_alloc_urbs*	MRR	A	71	igorplugusb_probe	MRR	S
13	audit_list_rules_send*	MRD	C	72	ap_probe*	MRR	S
14	audit_send_reply*	MRD	C	73	usb_urb_alloc_bulk_urbs*	MRR	S
15	radeon_user_framebuffer_create*	MRD	C	74	usb_urb_alloc_isoc_urbs*	MRR	S
16	edac_device_register_sysfs_main_kobj	MRD	C	75	mos7840_open	MRR	S
17	new_lockspace	MRD	C	76	lan78xx_tx_bh	MRR	S
18	acpi_cppc_processor_probe	MRD	C	77	send_mpa_reject	MRR	S
19	bond_sysfs_slave_add	MRD	C	78	gsmld_open	MRR	S
20	cpuidle_add_state_sysfs	MRD	C	79	lvs_rh_probe	MRR	S
21	cpuidle_add_sysfs	MRD	C	80	edd_device_register*	MRD	/
22	cpuidle_add_driver_sysfs	MRD	C	81	dn_nsp_rx_packet*	MRD	/
23	dmi_system_event_log*	MRD	C	82	cx231xx_init_audio_bulk*	MRR	/
24	dmi_sysfs_register_handle	MRD	C	83	i2400mu_notification_setup*	MRR	/
25	esre_create_sysfs_entry	MRD	C	84	submit_urbs*	MRR	/
26	fw_cfg_register_file	MRD	C	85	usb_isoc_urb_init*	MRR	/
27	cm_create_port_fs*	MRD	C	86	cx231xx_init_audio_isoc*	MRR	/
28	add_port*	MRD	C	87	usb_bulk_urb_init*	MRR	/
29	add_port(different from above)	MRD	C	88	brcmf_usbdev_qinit	MRR	P
30	iommu_group_alloc	MRD	C	89	x25_connect	MRR	P
31	cxl_sysfs_afu_new_cr	MRD	C	90	tm6000_start_stream	MRR	P
32	pci_create_slot	MRD	C	91	z3fold_reclaim_page	MRD	P
33	iscsi_boot_create_kobj	MRD	C	92	cacheinfo_create_index_dir	MRD	P
34	add_mdev_supported_type	MRD	C	93	ext4_init_sysfs	MRD	P
35	create_space_info	MRD	C	94	ext4_register_sysfs	MRD	P
36	qib_create_port_files	MRD	C	95	gfs2_sys_fs_add	MRD	P
37	pdcs_register_pathentries	MRD	C	96	cifs_writev_requeue	MRD	P
38	mei_me_cl_rm_by_uuid	MRD	C	97	nfs_file_direct_write	MRD	P
39	mei_me_cl_rm_by_uuid_id	MRD	C	98	nfs_file_direct_read	MRD	P
40	edac_pci_main_kobj_setup	MRD	C	99	cuse_channel_open	MRD	P
41	acpi_sysfs_add_hotplug_profile	MRD	C	100	br_add_if	MRD	P
42	efivar_create_sysfs_entry	MRD	C	101	netdev_queue_add_kobject	MRD	P
43	edac_pci_create_instance_kobj*	MRD	C	102	rx_queue_add_kobject	MRD	P
44	edac_device_create_block*	MRD	C	103	__rfcomm_create_dev	RRD	P
45	edac_device_create_instance*	MRD	C	104	rds_ib_get_mr	MRD	P
46	usX2Y_rate_set*	MRD	C	105	meson_ao_cec_probe	RRD	P
47	usnic_ib_sysfs_register_usdev*	MRD	C	106	amdgpu_cs_parser_init	RRD	P
48	display_init_sysfs	MRD	S	107	drm_dp_mst_allocate_vcpi	MRD	P
49	l2cap_sock_alloc*	MRD	S	108	amdgpu_cs_user_fence_chunk	MRD	P
50	bnx2fc_eh_abort*	MRD	S	109	mdev_register_device	MRD	P
51	core_scsi3_emulate_pro_register_and_move	MRD	S	110	lpuart_start_rx_dma	MRD	P
52	bnx2fc_initiate_seq_cleanup*	MRD	S	111	ohci_platform_probe	MRD	P
53	__tm6000_ir_int_start	MRR	S	112	cpufreq_policy_alloc	MRD	P
54	rtl2832_sdr_alloc_urbs*	MRR	S	113	btrfs_sysfs_add_fsid	MRD	P
55	stir_net_open	MRR	S	114	hfi1_create_port_files	MRD	P
56	dw_hdmi_cec_probe	MRR	S	115	exofs_sysfs_odev_add	MRD	P
57	_dsa_register_switch	MRR	S	116	exofs_sysfs_sb_add	MRD	P
58	gigaset_if_initdriver	MRR	S	117	ldebugfs_register_mountpoint	MRD	P
59	nvmet_fc_ls_create_connection*	MRR	S	118	cpufreq_dbs_governor_init	MRD	P

Table 5: Security impact of found bugs. Impact*: we consider a UAF can be exploited to hijack the control flow if the freed object contains a function pointer.

Bug	Impact*	Triggering Condition	Reachability	Permission Requirement
__rfcomm_create_dev	UAF likely exploitable	No	Userspace	CAP_NET_ADMIN
audit_list_rules_send	UAF likely exploitable	Exhaust memory Refcount overflow	Userspace	CAP_AUDIT_READ
bnx2fc_send_srr	UAF likely exploitable	Exhaust memory	Unclear	No
bnx2fc_send_rec	UAF likely exploitable	Exhaust memory	Unclear	No
kobject_rename	UAF likely exploitable	Refcount overflow	Unclear	No
write_parport_reg_nonblock	UAF likely exploitable	Exhaust memory Refcount overflow	Unclear	No
audit_send_reply	UAF likely exploitable	Exhaust memory Refcount overflow	Userspace	CAP_AUDIT_READ
__tm6000_ir_int_start	UAF likely exploitable	Exhaust memory	Userspace	No
cx231xx_init_audio_isoc*	UAF likely exploitable	Exhaust memory	Userpsace	No
usb_urb_alloc_bulk_urbs*	UAF likely exploitable	Exhaust memory	Userspace	No
cpufreq_policy_alloc	Memory leak	Exhaust memory	Called upon registering drivers	No
iommu_group_alloc	Memory leak	Exhaust memory	Unclear	No
esre_create_sysfs_entry	Memory leak	Exhaust memory	Called upon system initialization	No
dmi_sysfs_register_handle	Memory leak	Exhaust memory	Called upon system initialization	No
cpuidle_add_sysfs	Memory leak	Exhaust memory	Called upon registering cpu drivers	No
acpi_cppc_processor_probe	Memory leak	Exhaust memory	Called upon registering devices	No
edac_pci_main_kobj_setup	Memory leak	Exhaust memory	Called upon registering devices	No
mei_me_cl_rm_by_uuid	Memory leak	No	Called by IRQ	No
mei_me_cl_rm_by_uuid_id	Memory leak	No	Unclear	No
add_mdev_supported_type	Memory leak	Exhaust memory	Called upon registering devices	No
add_port	Memory leak	Exhaust memory	Called upon registering devices	No
ttm_bo_add_move_fence	Memory leak	Exhaust memory	Called upon module initialization	No