# RID: Finding Reference Count Bugs with Inconsistent Path Pair Checking

Junjie Mao Yu Chen Qixue Xiao Yuanchun Shi

Department of Computer Science and Technology, Tsinghua University

maojj12@mails.tsinghua.edu.cn, yuchen@mail.tsinghua.edu.cn, xqx12@mails.tsinghua.edu.cn, shiyc@tsinghua.edu.cn

## Abstract

Reference counts are widely used in OS kernels for resource management. However, reference counts are not trivial to be used correctly in large scale programs because it is left to developers to make sure that an increment to a reference count is always paired with a decrement. This paper proposes inconsistent path pair checking, a novel technique that can statically discover bugs related to reference counts without knowing how reference counts should be changed in a function. A prototype called RID is implemented and evaluations show that RID can discover more than 80 bugs which were confirmed by the developers in the latest Linux kernel. The results also show that RID tends to reveal bugs caused by developers' misunderstanding on API specifications or error conditions that are not handled properly.

***Categories and Subject Descriptors*** F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

***Keywords*** reference counting; inconsistency; static analysis

## 1. Introduction

Reference counts (referred to as refcounts hereinafter) are widely used in operating system kernel subsystems, such as synchronization mechanisms [19], management of dynamically allocated memory blocks [18] and dynamic power management. Misuses of refcounts can thus lead to different phenomena including memory exhaustion, mutual exclusion failures, abnormally high power consumption and security vulnerabilities. For instance, memory leaks due to improper reference count handling in Linux kernel can lead to Denial of Service attacks [29]. Bugs related to wake locks, a refcount-based power management mechanism in Android, have been identified as a significant root cause of abnormal power consumption on smartphones [22]. Several vulnerabilities in Windows kernel have direct relations with refcount usages and can lead to revelation of all system files or execution of arbitrary code with administrative privileges [10].

In OS kernels, refcount mechanisms typically encapsulate an integer as a counter and provide only APIs for incrementing and decrementing the counter. Developers are then required to ensure by themselves that all refcounts are used as intended, which is not trivial in complex programs. A recent survey [27] shows that violations to developers' intention, also known as semantic bugs, have emerged as one of the dominant threats to systems even in relatively mature software including the Linux kernel.

In this paper, we propose *inconsistent path pair checking* to statically detect refcount bugs in OS kernels . An *inconsistent path pair* (abbreviated as IPP) is a pair of two paths that (1) are in the same function, (2) both start from the entry of the function and end at the exit, (3) have different changes to a refcount, and (4) are, at runtime, indistinguishable outside the function by checking arguments and the return value. The definition of IPP does not depend on the context in which the function is called. As a result, IPP checking is also applicable to part of an OS kernel like libraries and drivers. In addition, IPP checking needs no extra assumption on how refcounts should be used in a function. The only information needed is the refcount API specifications which is a common requirement in the existing work [4, 12, 13, 17].

We implement a static checker, called RID, based on inconsistent path pair checking, program abstraction and summary-based inter-procedural analysis. We evaluate RID against the Linux kernel, and the results show that RID is helpful in revealing developers' misunderstanding of kernel API specifications and detecting refcount bugs effectively.

Although we focus on the Linux kernel, we believe the idea of IPP is general and applicable to other programs. We have also applied RID to the Python/C programs in which 100+ bugs are found.

This paper makes the following key contributions:



*ASPLOS '16* April 2–6, 2016, Atlanta, Georgia, USA.

DOI: http://dx.doi.org/10.1145/2872362.2872389

1. We propose inconsistent path pair checking, a novel technique for detecting refcount bugs with only the specifications of refcount APIs.
2. We present a summary-based inter-procedural analysis algorithm to detect inconsistent path pairs and implement a prototype which can analyze programs such as Linux kernel and Python/C programs.
3. We analyze the characteristics of bugs and false positives reported from RID by an evaluation against the Linux kernel and three Python/C programs. 83 new bugs are found in Linux kernel and confirmed by kernel developers. We also find an instance which shows an API can still be misused in many places in relatively mature software like Linux.

In the rest of this paper, Section 2 presents related work and discusses the novelty of our work. Section 3 introduces the characteristics of refcounts and describes inconsistent path pairs in more detail. Section 4 proposes our summary-based inter-procedural analysis on inconsistent path pairs, followed by Section 5 which presents the implementation details and limitations. An analysis on evaluation results are presented in Section 6. Section 7 concludes our work.

## 2. Related Work

### 2.1 Detecting Refcount Bugs

M. Emmi *et al.*have proposed Referee [4] which uses symbolic model checking to verify the absence of refcount bugs in closed programs where the complete control flow is known. Referee assumes that resources are arranged as an array and are managed independently. An unlimited number of threads have access to the resources. Referee further assumes that any thread manages any resource in exactly the same way (i.e. by the same piece of code). This assumption allows Referee to reduce the verification of all threads and resources to the verification of an arbitrary thread and an arbitrary resource, both of which are identified symbolically. RID, on the other hand, focuses on the analysis of programs in which refcounts are not necessarily independent or changed in the same way.

A. Lal and G. Ramalingam [12] propose a polynomial time algorithm to statically detect refcount bugs in closed programs with only shallow aliasing. They identify the entry function of the program (usually `main()`), initialize all refcounts to 0 at the entry point of the function, insert assertions at the exit claiming that all refcounts must be 0, and verify that the assertions are never violated. Their methodology cannot be applied to open programs either, because it is too strong to assume that all entry functions in open programs like libraries must leave all refcounts unchanged.

Cpychecker, proposed by D. Malcom [17], and Pungi, proposed by S. Li *et al.*[13], analyze refcount bugs in the native implementations of Python/C programs. They rely on the rule that, in any function, the change of a refcount must be equal to the number of references escaped from the function. A reference can escape by the return value of the function or by Python/C APIs that steal references [23]. Pungi adopts the idea of affine abstraction in [12] and analyzes on Static Single Assignment (SSA) form of the program. Note that wrappers to the basic refcount APIs, which is common in subsystems of Linux, are always considered as an error according to the rule above. Thus, to apply the above rule to Linux kernel, a comprehensive list of refcount API wrappers must be given and maintained along with the rapid evolution of the Linux kernel.

Compared to Pungi and Cpychecker, RID checks programs against a weaker property which does not rely on any assumption on how refcounts should be changed in a function. Theoretically any bug found by RID (using a weaker property) should be detectable by the methods of Pungi or Cpychecker (using a stronger property) if the same analysis techniques (e.g. SSA form) are adopted. However, using a weaker property allows RID not to generate false alarms which are raised by Pungi or Cpychecker when the assumption they adopt is too strict for a function, which is common due to the reason presented above. No false alarm suppression techniques is mentioned in Pungi. In Cpychecker, suppressing these false alarms must be done by manually adding GCC attributes to each of these functions. Meanwhile, stronger properties can be easily integrated into RID by introducing corresponding checks on function summaries introduced in Section 4.

### 2.2 Specification Inference

Specification inference is another category of techniques that can check programs without rules defined by developers. Engler *et al.*use belief analysis to extract API usage patterns [6] and implement checkers based on state machine abstraction [5, 9]. Specification mining [8, 14, 15, 30] is another category of techniques which adopt statistical analysis to infer API usage rules that are commonly followed by a sufficient number of occurrences in the code. Different thresholds are used to filter out less likely rules and the remaining are used for checking. Hector [25], on the other hand, infers the acquisition and release of a specific resource in a function instead of seeking for general usage patterns of resource management APIs.

S. Saha *et al.*have shown that a significant part of API usage rules are under the thresholds used in recent specification mining work and thus cannot be found by these tools [25]. Different from the work above, RID analyzes a program in a bottom-up manner. The behavior of a function is calculated from the function body, rather than from how the function is used. This allows RID to detect bugs involving APIs that are used occasionally or are commonly misused. Examples are presented in Section 6.

### 2.3 Inconsistency Checking

In literature, various kinds of inconsistencies have been defined and used in static analysis.

D. R. Engler *et al.*define inconsistencies as incompatible beliefs implied by multiple uses of a value [6]. I. Dillig *et al.*[3] further propose a formal description of inconsistent beliefs by considering inconsistency checking as a variance of type inference and show that inconsistency checking is a complement to standard semantic-based approaches such as source-sink analysis. A. Tomb *et al.*[28] detect pieces of program code that have incompatible beliefs but can execute in sequence by solving the reachability problem.

S. Lu *et al.*propose MUVI to infer the access correlations of multiple fields in a data structure [16]. An update to some of these fields is considered inconsistent if the correlated fields are not updated accordingly. The access correlations are inferred by the patterns how multiple data structure fields are accessed in the same function.

M. Gabel *et al.*detect inconsistencies among changes to duplicated code syntactically [7]. Textual similarity of clone code and syntactic filters to change operations are used to extract likely bugs from a large set of similar code fragment pairs.

M. Schäf *et al.*consider a code fragment inconsistent if *it is not part of any normally terminating execution* [26]. Examples include a infeasible path, where the conjunction of branch conditions is not satisfiable, and a branch never taken because the branch condition always evaluates to true at runtime. Instead of detecting new bugs, M. Schäf *et al.*use inconsistency checking for fault localization.

Different from existing work, RID adopts a different notion of consistency, i.e. different refcount changes that are made in two paths in the same function and are indistinguishable outside the function at runtime. With IPP detection, RID can reveal bugs that are challenging to find using inferred beliefs or common usage patterns (illustrated in section 3.4 by example), showing that IPP detection is complementary to existing inconsistency checking techniques.

## 3. Refcounts and Inconsistent Path Pairs

### 3.1 Characteristics of Refcounts

A *refcount* is an integer encapsulated in a structure and is used to track the number of references to the structure. Note that a *reference* can be defined in different ways. For example, a refcount of a dynamically allocated object tracks the number of pointers that point to this object in the system. A refcount of a device structure, on the other hand, may track the number of threads that are using the device.

From the analysis and statistics on over 270k functions in Linux kernel, we observe that the following characteristics are commonly followed by the refcount APIs.

1. A refcount is mostly incremented or decremented by 1 at a time.
2. The exact value of a refcount is seldom accessed. Especially, the exact value is hardly used in any branch condition.
3. From any system state, it is possible for the refcount to be decremented to 0.
4. The value of a refcount must always be a non-negative.

The characteristics above applies to over 800 sets of refcount APIs, consisting of more than 1600 functions in Linux kernel. These APIs are found by a syntactical search for functions with similar names except some common antonyms such as 'inc'-'dec' and 'get'-'put'. A similar approach is used to discover paired functions in [15]. 10987 out of 11755 (93.5%) files in Linux kernel release 3.17 have functions calling these APIs directly or indirectly. In this paper, we only consider refcounts used in the common way, i.e. refcounts are changed by 1 at a time and the exact value is not accessed.

We use the Power Management (PM) counts in Linux DPM subsystem to illustrate the characteristics above. The counts are not designed to be accessible outside DPM subsystem. When a thread (typically created or scheduled by a device driver) needs to carry out some operations on a device, it increments the count of the device by calling an API provided by DPM and decrements the count when the operations are completed. When the count reaches 0, the corresponding device is scheduled to be suspended to power-saving mode. Each time the count is incremented, the device is resumed if it is in power-saving mode. A PM count that can never reach 0 under certain system state indicates that the corresponding device will keep active forever, leading to abnormal power drainage. A PM count with a negative value indicates that it is possible for a thread to carry out device operations when the device is in power-saving mode, which can lead to unexpected results.

In this paper, we call any violation to characteristic 3 or 4 above as a *refcount bug*.

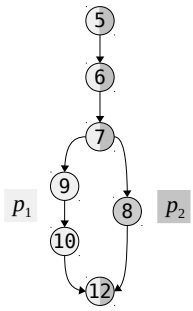
### 3.2 Inconsistent Path Pairs

Our key observation is that inconsistent path pairs often indicates violations to one of the two characteristics of refcounts. We illustrate the point by the example function `foo()` in Figure 1(a). The function `foo()` determines if the device is usable by checking the value of register located at 0x54 (the register is accessible even the device is suspended). The PM count of the device is incremented and more device operations are carried out if the register contains a positive.

Figure 1(b) shows the CFG (control flow graph) of `foo()` with each vertex representing a statement. The two paths in `foo()`, i.e. $p_1$ and $p_2$, are shown with different gray scale in the CFG. Note that the PM count of `dev` is incremented in $p_1$ but not in $p_2$. Furthermore, the only difference between $p_1$ and $p_2$ besides the change to PM count of `dev` is that $p_1$ carries out some device operations (line 8) while $p_2$ does not.

```
1  int reg_read(device *d, int reg);
2  void inc_pmcount(device *d);
3
4  int foo(device *dev) {
5    assert(dev != NULL);
6    int v = reg_read(dev, 0x54);
7    if (v <= 0)
8      goto exit;
9    inc_pmcount(dev);
10   // more register reads/writes
11 exit:
12   return 0;
13 }
```

(a)




(b)

Figure 1: (a) an example program for illustration, and (b) the control flow graph of the example program with each statement as a vertex. The exception path handling assertion failure is ignored for brevity. The numbers in vertices are the line numbers of the statement the vertices represent. $(p_1, p_2)$ is an inconsistent path pair.

As the exact value of the PM count is not accessible, a caller to `foo()` can hardly tell which of the two paths is executed at runtime. Especially, given the same structure representing a device, both the paths are feasible and return 0. As a result, it is not possible to determine outside `foo()` which path is executed by checking the return value at runtime. We call the path pair $(p_1, p_2)$ an *inconsistent path pair*.

The example above leads us to a general definition to *inconsistent path pairs*. An inconsistent path pair, or IPP, is a pair of paths satisfying the following conditions.

1. Both paths are in the same function.
2. Both paths start from the entry of the function and end at the exit.
3. The two paths have different changes to a refcount.
4. It is possible that, given the same arguments, both the paths are feasible and return the same value.

We informally explain why IPPs can reveal refcount bugs with the example function `foo()`. Assume that, at runtime, the program reaches the entry of `foo()` with the PM count of `dev` being $n$. After $p_1$ is executed, the PM count is incremented to $n + 1$. If the PM count of `dev` can never be decremented to 0 from the exit of `foo()`, then characteristic 3 is violated. Otherwise, let $p^{'}$ be a runtime path which starts at the exit of `foo()` and decrements the PM count of `dev` to 0. As $p_1$ and $p_2$ are indistinguishable outside `foo()` by checking the arguments passed to `foo()` or the value returned by `foo()`, it is possible for the program to follow $p^{'}$ even if $p_2$ is executed instead of $p_1$ in `foo()`. As a result, the PM count of `dev` is still $n$ at the exit of `foo()`. After the program executes $p^{'}$, the PM count of `dev` will be decremented to $-1$, which violates characteristic 4. In summary, a refcount bug can exist no matter which path in the IPP is taken at runtime.

### 3.3 Detecting IPP: An Example

To check IPPs in a program, we need to compare any two paths in a function and determine whether the third and fourth condition in the definition of IPP can be satisfied. A precise understanding of refcount changes in a path requires the information of how functions called in the path change refcounts. We use a function summary to record the refcount changes and the return values under different constraints. A summary based inter-procedural analysis is then needed for checking IPPs in programs.

We first introduce how `foo()` in Figure 1(a) is analyzed. A general introduction to the analysis follows in section 4. Figure 2 shows how we check IPPs in `foo()`. From a high level of view, the analysis consists of three steps. (I) Enumerate all paths in `foo()`. (II) Summarize each path independently. (III) Check consistency among summaries of paths, report any inconsistency found and use the union of the consistent summaries as the function summary for `foo()`.

#### 3.3.1 Summaries of the Called Functions

We summarize a function by a set of *summary entries* (referred to as entries hereinafter). Each entry records 1) how refcounts will be changed (by the field *changes*), 2) what can be returned (by the field *return*) and 3) under what constraints can this entry be applied (by the field *cons*). The constraints are on the arguments and the return value of the function.

For example, the summary of `reg_read()`, shown at the bottom of Figure 2, consists of two entries. The first entry applies when the first argument of `reg_read()` (written as $[d]$) is not a null pointer and the return value (written as $[0]$) is always nonnegative. The second entry applies when the return value is -1, without any constraints on the arguments. Neither of the entries changes any refcount. The implementation of `reg_read()` is given in Figure 2 for reference.

The summary of `inc_pmcount()` has the same form as the summary of `reg_read()`. The main difference is that `inc_pmcount()` 1) has no return value, and 2) increments the PM count of its only argument (written as $[d].pm$) when the argument is not null.

#### 3.3.2 STEP I: Enumerating Paths

The first step to summarize `foo()` is to enumerate all paths in the function and attach a path constraint to each of them. In the example function `foo()`, there are two paths, i.e. $p_1$ and $p_2$ in Figure 1(b). The corresponding path constraints are $v > 0$ and $v \leq 0$, respectively, where $v$ is the local variable in `foo()`. Both of the paths assert that the first

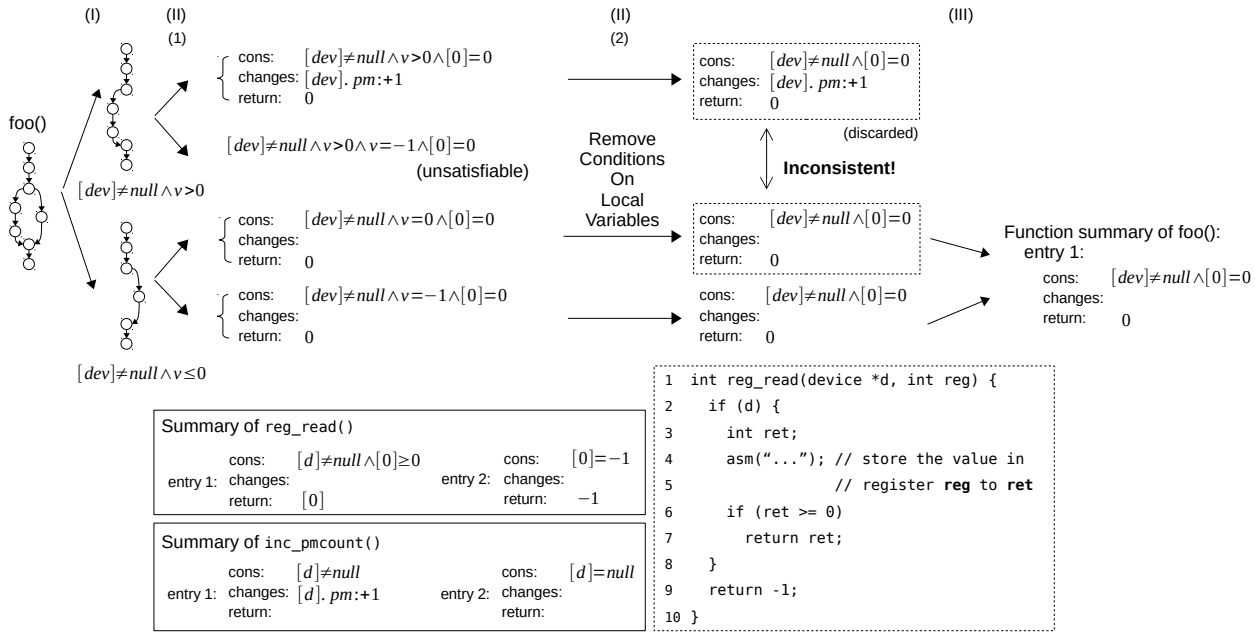



Figure 2: The complete analysis of foo() in figure 1.

formal argument of `foo()` (written as $[dev]$) is not null due to the assertion at line 5.

#### 3.3.3 STEP II: Calculating Summaries for Each Path

Each path enumerated in Step I is summarized independently. The summary of a path is a set of entries, each of which records how refcounts are changed in the path under the constraint in the entry. We ensure that the constraint in an entry can uniquely determine the refcount changes in and the return value of all functions called in the path. As a result, multiple entries may be needed to summarize a path.

We describe as an example how $p_2$ is analyzed. In $p_2$, $v$ is non-positive and no refcount is changed. Note that the path constraint of $p_2$ cannot uniquely determine the return value of `reg_read()`. As a result, two entries are needed to summarize the path, one for each entry in the summary of `reg_read()`. We denote the path constraint of $p_2$ as $c_p$ and the constraints in the summary of `reg_read()` as $c_1$ and $c_2$, respectively. Then one entry summarizing $p_2$ has the constraint $c_p \wedge c_1$, asserting the local variable $v$ is non-positive and non-negative, i.e. equals to 0, and the other entry has the constraint $c_p \wedge c_2$, asserting $v$ is -1. Both of the entry has no refcount changes recorded since `reg_read()`, the only function called in $p_2$, does not change any refcounts.

After summaries of paths are calculated, we remove conditions on local variables ($v$ in the example) from the constraints because local variables are discarded when the function returns and thus can not be accessed outside the function.

#### 3.3.4 STEP III: Checking and Merging Summaries of Paths

Any two path summaries calculated in step II are checked for consistency. In Figure 2, the path summaries in dashed boxes are inconsistent because they share the same constraints but have different refcount changes.

The function summary of `foo()` is the union of entries calculated in step II. If two path summaries are inconsistent, we randomly drop one of them.

### 3.4 Difference Between IPP and other Notions of Inconsistency

The example above also shows that IPP detection is able to reveal bugs that cannot be found by detecting other kinds of inconsistency. Belief analysis can infer that the refcount of `dev` must be non-negative at the entry of `foo` from how the refcount is operated in the function. However, it is still challenging to deduce the bug in `foo` from such beliefs. With only the definition of `foo`, inconsistency detection techniques based on common usage patterns are not able to find the bug, either, as no common usage pattern can be extracted.

## 4. Inter-procedural Analysis for IPP Checking

As is stated in Section 3.3, how a function changes refcounts depends on the behavior of its callees. Thus an inter-procedural analysis is required. In this section, we describe our inter-procedural analysis in general.

$$
\begin{array}{llll}
\textit{(Predicate)} & p & := & = \mid \neq \mid > \mid \geq \mid < \mid \leq \\
\textit{(Value)} & v & := & x \mid \textbf{\textit{numeral}} \mid true \mid false \\
\textit{(Instruction)} & inst & := & x = v \\
 & & \mid & x = y.\textbf{\textit{field}} \\
 & & \mid & x = \textbf{\textit{random}} \\
 & & \mid & \textbf{\textit{fn}}\ (v_1, ..., v_n) \\
 & & \mid & x = \textbf{\textit{fn}}\ (v_1, ..., v_n) \\
 & & \mid & \textbf{\textit{return}}\ v \\
 & & \mid & x = v_1\ p\ v_2 \\
 & & \mid & \textbf{\textit{branch}}\ x, l_1, l_2 \\
 & & \mid & \textbf{\textit{branch}}\ l
\end{array}
$$

Figure 3: Syntax of instructions in our abstract programs. $x, y$ are variables, ***numeral*** is a numeral constant, $l_i$ are labels marking positions where the control flow can jump to, ***field*** is a field name, ***random*** is a random number generator, and ***fn*** is a function identifier. Variables in conditional branches must be defined by an (in)equality expression.

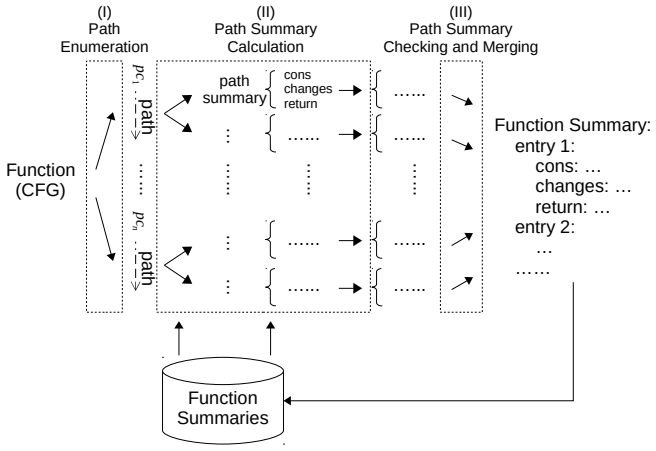



Figure 4: Overview of summary-based IPP analysis on a function

### 4.1 Program Abstraction

In this paper, we focus on the abstract program defined in Figure 3. The abstraction extends *affine programs* proposed in [12] and [13] by preserving formal arguments, branch conditions and return values, all of which are critical in IPP checking because formal arguments and return values are involved in the definition of IPPs. Another minor difference is that we assume refcounts can only be changed via refcount APIs. As a result, our abstraction ignores arithmetic operations which is mostly used to model changes to refcounts in previous work. A random generator is introduced as an abstraction of operations generating non-deterministic results, such as reading from device registers.

### 4.2 Summary-based Inter-procedural Analysis

Our summary-based inter-procedural analysis works on the aforementioned program abstraction (see Section 4.1). Functions are traversed and summarized in the reverse topological order of the call graph (recursive calls are randomly broken). A function summary is a set of summary entries, each of which summarizes the refcount changes and the return value of the function under certain constraints (see Section 4.3). Figure 4 shows the process of analyzing a single function. The process consists of the following three steps.

1. **Path Enumeration.** We first enumerate all paths in the given function. Loops are unrolled at most once in a path. Each path has a *constraint* showing when this path will be executed. The constraint is represented by first order formulas with LIA (linear integer arithmetic) theory [2].
2. **Path Summary Calculation.** Each path enumerated is summarized by symbolic execution (defined in section 4.4). After the summaries are calculated, conditions on local variables are removed from the constraints in the summary.
3. **Path Summary Checking and Merging.** After calculating all path summaries, we check whether these summaries are consistent and report any inconsistency found. The function summary is the set of consistent path summaries, and is added to the function summary database for future use.

### 4.3 Definition of the Summary

Formally, an entry in a summary of function is a triple

$$S = (cons, changes, return)$$

where $cons$ is a constraint on arguments and return values, $changes$ is a map from refcounts to integers, and $return$ is an expression representing the return value. A refcount is represented by an expression in our symbolic execution (see Section 4.4). We denote $cons$ in $S$ by $S.cons$, the map $changes$ in $S$ by $S.changes$, $return$ in $S$ by $S.return$, and the change of refcount $rc$ in $S$ by $S.changes[rc]$.

We restrict that the conjunction of the constraints of any two entries in a function is unsatisfiable. If not, assume there are two entries $e_1$ and $e_2$ where $e_1.cons \wedge e_2.cons$ is satisfiable. If the refcount changes in $e_1$ and $e_2$ are same for all refcounts, the entries can be merged into a single entry where the constraint is $e_1.cons \vee e_2.cons$. The return expression in the merged entry is $e_1.return$ if $e_1.return = e_2.return$ and $[0]$ otherwise. If the refcount changes in $e_1$ and $e_2$ are different, $e_1$ and $e_2$ are considered inconsistent and one of them must be dropped (refer to Section 4.5 on how inconsistency are detected).

### 4.4 Path Summary Calculation

Summary of a path is calculated by executing the paths symbolically. Valuation to variables are expressed by symbolic

| | | | |
|---|---|---|---|
| (*Predicate*) | $p$ | := | $= \mid \neq \mid > \mid \geq \mid < \mid \leq$ |
| (*Condition*) | $cond$ | := | $e_1\ p\ e_2$ |
| (*Argument*) | $arg$ | := | [ ***ident*** ] |
| (*Return Value*) | $ret$ | := | $[0]$ |
| (*Local Variable*) | $local$ | := | ***ident*** |
| (*Field*) | $field$ | := | $e.$***field*** |
| (*Expression*) | $e$ | := | ***const*** $\mid cond \mid arg \mid ret \mid local \mid field$ |

Figure 5: Syntax of expressions in the symbolic execution. ***field*** is a field name, ***const*** is a numeral or boolean constant, and ***ident*** is the name of an identifier.

| | |
|---|---|
| $x = v$ | $vmap[x] = vmap[v]$ |
| $x = y.$***field*** | $vmap[x] = vmap[y].$***field*** |
| $x =$ ***random*** $l$ | (none) |
| ***return*** $v$ | $return = vmap[v]$ |
| $x = v_1\ p\ v_2$ | $vmap[x] = vmap[v_1]\ p\ vmap[v_2]$ |
| ***branch*** $x, l_1, l_2$ | remove any condition previously added by this instruction from $cons$, then $cons = cons \wedge vmap[x]$ if $l_1$ is taken otherwise $cons = cons \wedge (\neg vmap[x])$ |
| ***branch*** $l$ | (none) |

Figure 6: How instructions are evaluated in the symbolic execution. ***field*** is a field name.

expressions defined in Figure 5. A *state* in the symbolic execution is a quintuple

$$St = (ip, cons, changes, return, vmap)$$

where $ip$ points to the next instruction to be executed, $cons$ is a path constraint, $changes$ is a map from refcounts to integers, $return$ is the value returned and $vmap$ is a map from variables to values. The initial state is defined as follows.

- $ip$ points to the first instruction in the function.
- $cons$ is $True$.
- $changes$ is an empty map and $changes[x]$ is 0 by default if $x$ is not a key in $changes$.
- $return$ is an empty string.
- For each formal argument $arg$ of the function, the value of $vmap[arg]$ is $[arg]$. Otherwise $vmap[x]$ is $x$ by default, where $x$ can be a numeral constant, a boolean constant or a variable.

Figure 6 shows how instructions (except function calls) are executed in an informal manner. For example, $vmap[x] = vmap[v]$ means the value of key $x$ in $vmap$ is set to the value of key $v$ in $vmap$. Updates to the instruction pointer $ip$ are straightforward and are not shown for brevity.

**input** : The current state s
**output**: A set of states states

```
1  inst ← the call instruction pointed to by s.ip
2  summary ← summary of the function called by inst,
   instantiated with s.vmap
3  states ← {}
4  foreach entry in summary do
5      new_constraint ← s.cons ∧ entry.cons
6      if new_constraint is satisfiable then
7          s' ← allocate a new state
8          s'.ip ← pointer to the next instruction of inst
9          s'.cons ← new_constraint
10         s'.changes = s.changes
11         foreach key rc in entry.changes do
12             s'.changes[rc ] ← s'.changes[rc ] +
               entry.changes[rc ]
13         end
14         s'.vmap ← s.vmap
15         ret ← entry.return
16         var ← the variable to which the return value of
           inst is assigned to
17         s'.vmap[var ] ← ret
18         add s' to states
19     end
20 end
21 return states
```

**Algorithm 1:** Execution of a call instruction. A function summary is instantiated by replacing formal arguments and $[0]$ to actual arguments and the variable holding the return value.

Execution of a call instruction is more complicated as it uses the summary of the called function and can create new states. Algorithm 1 shows how a call instruction is executed in RID. Each entry in the summary of the called function is first *instantiated*, i.e. formal arguments are replaced by the expressions of actual arguments and $[0]$ is replaced by the name of the variable holding the return value of the call. For each entry in the instantiated summary, a new state is created if the conjunction of the constraints in the current state and in the entry is satisfiable (line 6). In the new state $s'$, $s'.cons$ is the conjunction above (line 9), the refcount changes in the entry are added to $s'.changes$ (line 10 to 13), and the expression of the variable holding the return value is updated (line 14 to 17). The execution generates a set of new states each of which will be executed independently.

Each time a return instruction is executed, an entry is created with the constraint, the refcount changes and the expression of the return value in the state. Conditions on local variables are then removed from the constraint.

### 4.5 Path Summary Checking

With a collection of summary entries $S_1, ..., S_n$, we check if there exist two different entries $S_i, S_j$ satisfying the following conditions:

- $S_i.cons \wedge S_j.cons$ is satisfiable.
- There exists a refcount $rc$ such that $S_i.changes[rc] \neq S_j.changes[rc]$.

Let $p_i$ and $p_j$ be the path represented by $S_i$ and $S_j$ respectively. A pair $S_i, S_j$ satisfying the above conditions indicates that $(p_i, p_j)$ is an IPP defined in section 3.2. This is because the satisfiability of $S_i.cons \wedge S_j.cons$ means we can find an initial state $St$ such that $p_i$ and $p_j$ are both feasible and the same value is returned (recall that $S_i.cons$ and $S_j.cons$ also contains conditions on the return value). Each refcount with different changes in the IPP is reported as a bug. One of the path in each IPP, randomly selected, will be discarded to avoid further reports at the call sites of the function. However, it is possible that RID drops a summary that does not meet the expected behavior of the function, as RID has no idea which summary reflects the *expected* behavior. As a result, RID may give redundant reports or fail to report bugs in callers of the function. A possible solution to this problem is discussed in section 5.4.

The function summary is defined as the set of the path summaries excluding the ones discarded during IPP checking. If the program under analysis respect other rules, a corresponding check on the refcount changes in the function summary can be added.

## 5. Implementation

We implement the analysis as a pass in the LLVM framework. All constraints are expressed in SMT formulas with linear integer arithmetic theory [1] and solved by Z3 [20].

### 5.1 Predefined Function Summaries

RID encodes the specifications of refcount APIs by predefined function summaries (referred to as predefined summaries hereinafter). If a predefined summary is given for a function, RID uses the predefined summary without analyzing the body of the function. RID, like existing tools [13, 17], expects that predefined summaries of APIs are given. Usually, predefined summaries are not changed frequently as the specifications of refcount APIs tend to be stable.

Figure 7 lists some examples of predefined summaries RID uses to analyze refcounts in Linux DPM and Python/C programs. The predefined summaries for DPM APIs in Linux is straightforward. API functions beginning with `pm_runtime_get` increment the refcount of the given device and functions beginning with `pm_runtime_put` do the opposite. Wrapper functions based on these APIs have no predefined summaries and are left to be analyzed by RID.

In Python/C, `Py_INCREF()` and `Py_DECREF()` are the basic interfaces to increment or decrement the refcount of a python object. In addition, other APIs may return borrowed reference or steal the caller's reference to an object passed to the function, as is explained in [23]. We briefly introduce some examples in the following paragraphs.

| Program | API | summary |
|---|---|---|
| Linux | pm_runtime_get<br>pm_runtime_get_sync | cons: $True$<br>change: $[dev].pm$: +1<br>return: [0] |
| | pm_runtime_put<br>pm_runtime_put_sync | cons: $True$<br>change: $[dev].pm$: -1<br>return: [0] |
| Python/C | Py_INCREF | cons: $True$<br>change: $[o].rc$: +1<br>return: |
| | Py_DECREF | cons: $True$<br>change: $[o].rc$: -1<br>return: |
| | Py_BuildValue<br>PyList_New<br>PyInt_FromLong | cons: $[0] \neq null$<br>change: $[0].rc$: +1<br>return: [0]<br>cons: $[0] = null$<br>change:<br>return: $null$ |
| | PyList_GetItem | cons: $True$<br>change:<br>return: [0] |
| | PyErr_SetObject | cons: $True$<br>change: $[type].rc$: +1<br>$[value].rc$: +1<br>return: |
| | PyList_SetItem | cons: $True$<br>change:<br>return: |

Figure 7: Examples of refcount APIs in Linux kernel and Python/C programs, and the predefined summaries of these APIs.

Predefined summaries of APIs returning a new reference usually consist of two entries because these APIs involve memory allocation that can fail. Among the two entries, one returns a new reference successfully when the refcount of the returned object is incremented, and the other returns a null pointer and no refcount is changed. Examples include `Py_BuildValue()`, `PyList_New()` and `PyInt_FromLong()`.

APIs that create new references to the arguments increment the refcount in their summary. In Figure 7, `PyErr_SetObject()` is an API of this type and the corresponding predefined summary increment the refcount of $[type]$ and $[value]$, which are the two formal arguments of `PyErr_SetObject()`.

APIs that return a borrowed reference or steal a reference to an object passed to them have simpler summaries as these APIs do not change any refcount. For example, `PyList_GetItem()` returns a borrowed reference and `PyList_SetItem()` steals a reference to the object to be inserted.

The predefined summaries are manually written according to the documentation available in Python source [24]. It is also possible to generate the predefined summaries automatically because the documentation has a well-defined syntax.

### 5.2 Analysis of Large Scale Programs

Analyzing a complete OS kernel like Linux by symbolic execution with constraint solving is usually unacceptably expensive. To reduce the number of functions we need to analyze, we divide functions in Linux into the following three categories.

1. **Functions with refcount changes.** Functions in this category calls other functions with refcount changes. The basic functions in this category are refcount APIs with predefined summaries.
2. **Functions affecting those with refcount changes.** Functions in this category do not call any functions with refcount changes. However, at some call sites of the functions, return values of the functions can affect how functions in the first category change refcounts.
3. **The others.** This category consists of functions not belonging to the previous two categories.

We classify the functions by a two-phase analysis on the call graph. In the first phase, the set of functions with refcount changes is calculated by traversing all functions in the call graph in reverse-topological order (note that recursive calls are broken in the call graph), with an initial set including the refcount APIs. In the second phase, the functions are traversed in topological order of the call graph. For each function, a static backward slice [21, 31] is calculated. The slice criteria includes the return value and all actual arguments passed to functions with refcount changes. Thus a function called in the slice may affect the behavior of functions with refcount changes and falls into the second category if it is not in the first category.

With the above three categories in mind, we analyze functions from different categories in different ways. Functions with refcount changes are fully analyzed because any two paths in a function may be inconsistent. Functions affecting those with refcount changes are analyzed selectively. This is because these functions have no refcount changes and the only information useful is the conditions on their return values. We avoid analyzing complex functions in this category and assume these functions can return any possible value. In RID, we analyze functions in this category with no more than three conditional branches. Functions in the third category are ignored as no information RID summarizes will be used.

RID also limits the total number of paths to be enumerated in a function and the number of path summaries generated from a path. If a function is not fully analyzed due to the limits, a default summary entry, which has no refcount changes and no conditions on the return value, will be added to the summary of the function. Options are provided to adjust the limitations according to the program to be analyzed.

### 5.3 Preparation of the Program under Analysis

RID analyzes a single LLVM bitcode file a time. A program containing multiple source files can either be compiled into a single bitcode file, or into multiple bitcode files which can be analyzed separately.

When linking multiple bitcode files in LLVM, we meet an implementation issue that static functions defined in headers have multiple copies. This can significantly increase analysis time in large scale programs because all of the duplicate functions must be analyzed. To avoid the duplication, we mark all function defined in headers as *weak symbols* before linking, so that multiple weak symbols will be merged into a single symbol.

RID can also analyze multiple files separately. In this case, function summaries calculated in one source file will be saved to disk and used when another source file is analyzed. To determine a proper order of the sources, a dependency graph of the sources is built where A depends on B if and only if A uses any symbols defined in B. We then calculate strongly connected components (SCCs) in the dependency graph, link sources in the same SCC into a single file and analyze these SCCs in reverse topological order. Multiple SCCs can be analyzed in parallel as long as the SCCs they depend on have been analyzed. This technique helps reduce the total time needed to analyze a large scale program.

### 5.4 Limitations

We list the limitations of RID as follows and briefly discuss their influence on the results of the analysis.

First, the abstract program analyzed by RID does not include features such as function pointers, bit operations and stores to data structure fields. When a branch condition is out of the expressive power of linear arithmetic theory used by RID, the branch condition is forgetted. Forgetting some branch conditions or updates other than resources and the return value may make two path indistinguishable to RID though they are actually not. Thus the design decisions above can lead to false positives but no false negatives. In the future, we plan to enhance RID by adopting techniques such as alias analysis [11] and SMT BitVector Theory.

Second, RID unrolls loop at most once. Some bugs cannot be detected if they can only be triggered by executing a loop body multiple times.

Third, RID limits the number of paths analyzed in a function and may not visit all possible paths in a complex

function. In the future we plan to investigate whether static slicing is helpful in reducing the number of paths RID need to analyze.

Fourth, randomly dropping one of two inconsistent reference count update summaries in one function, say `bar()`, may hide some of inconsistencies exist in the callers of `bar()`. These inconsistencies in the callers can be revealed by an incremental recheck which uses previously calculated summaries of unaffected functions once the inconsistency in `bar()` is fixed.

## 6. Evaluation

### 6.1 Benchmark and Configuration

We evaluate RID against two kinds of sources, i.e. the Linux kernel and a set of native implementaion in Python/C programs.

For the Linux kernel, we use the 3.17 release and concentrate on the DPM (Dynamic Power Management) subsystem which is a necessity for device drivers to support runtime power management. We choose DPM as a representative of refcount APIs because refcount bugs in DPM APIs are known to cause severe errors including system crash, boot hang and abnormally high power consumption. One of the fundamental elements in DPM is a per-device refcount. The count of a device is incremented by 1 whenever the driver is about to execute some device operations and decremented by 1 after the operations are done. The bugs are then double-checked in the upstream kernel and reported to the kernel mail list if they exist in the latest code base.

For the Python/C programs, we focus on the reference count usage of Python objects in the native code, as is done in previous work [13]. Predefined summaries follow the official Python/C API reference manual [23]. We choose three Python/C programs, namely krbV, pyaudio and ldap, and compares the reports of RID with warnings given by cpychecker [17] (the source code of Pungi[13] is not yet available). Cpychecker is a rule-based checker relying on the extra rule discussed in section 2.1, i.e. the change of a reference count of an object must be equal to the number of references escaping from the function. Cpychecker also allows a user to add GCC attributes in the source to indicate that a function steals a reference to an argument or returns borrowed reference.

Examples of predefined summaries used in our evaluation are given in Section 5.1. We omit the full list of predefined summaries due to the large amount of Python/C APIs. The limits of paths enumerated in a function and subcases in a path are set to 100 and 10, respectively. All evaluations are carried out on a Fedora Linux box with eight 2.4GHz x86-64 cores and 8GB memory.

### 6.2 Bugs Detected and Their Characteristics

RID has found 83 new bugs out of 355 reports in Linux involving DPM. These bugs are reported to kernel mail lists and confirmed by kernel community developers. Similar bugs fixed in recent Linux development are known to cause system crashes, kernel boot hangs and abnormal power consumption. We show two representative examples to illustrate the characteristics of bugs found by RID.

```
1  ... radeon_crtc_set_config(dev) {
2    ...
3    ret = pm_runtime_get_sync(dev);
4    if (ret < 0)
5      return ret;
6    ...
7    ret = drm_crtc_helper_set_config(set);
8    ...
9    pm_runtime_put_autosuspend(dev);
10   return ret;
11 }
```

Figure 8: A DPM API misuse detected by RID. pm_runtime_get_sync() increments a refcount regardless of its return value. pm_runtime_put_autosuspend() decrements the refcount.

One typical category of bugs caught by RID reflects the difference between specifications of refcount APIs and programmers' understanding. For example, the function `radeon_crtc_set_config()` in Figure 8, before carrying out further device operations, increments the refcount of the device by calling `pm_runtime_get_sync()` and returns directly if `pm_runtime_get_sync()` returns an error code (which is a negative integer). This reflects that the developer assumes `pm_runtime_get_sync()` should do nothing if it returns an error. This assumption, though widely adopted in Linux, is not true for `pm_runtime_get_sync()` which always increment the refcount, as is shown in the predefined summary in section 5.1. As a result, the increment to the refcount is never balanced by a pairing decrement, which prevents the driver from suspending the device when the device is not in use.

Another category of bugs found reveals the improper error handling when invoking an API that may fail. Figure 9 shows a subsystem specific wrapper to DPM, namely `usb_autopm_get_interface()`, which is commonly used in USB drivers. This wrapper works in a different way from `pm_runtime_get_sync()`, i.e. no refcount is changed if the API returns an error code. With the predefined summary of `pm_runtime_get_sync()`, RID can automatically calculate the precise summary of `usb_autopm_get_interface()` and detect the bug in `idmouse_open()` which does not decrement the refcount when `idmouse_create_image()` fails.

The above examples also show why RID can detect bugs with only the refcount API specifications. The key point is that it is common for a function, which implement a high level functionality, to invoke multiple APIs. A well-written

```
int usb_autopm_get_interface
    (struct usb_interface *intf)
{
  int status;
  status = pm_runtime_get_sync(&intf->dev);
  if (status < 0)
    pm_runtime_put_sync(&intf->dev);
  if (status > 0)
    status = 0;
  return status;
}

int idmouse_open(struct inode *inode,
                 struct file *file)
{
  interface = ...;
  ...
  result =
    usb_autopm_get_interface(interface);
  if (result)
    goto error;
  result = idmouse_create_image (dev);
  if (result)
    goto error;
  usb_autopm_put_interface(interface);
error:
  return result;
}
```

Figure 9: A bug detected by RID. Different from pm_runtime_get_sync(), usb_autopm_get_interface(), a wrapper of DPM in USB subsystem, does not change the PM count when it returns an error code. How this subsystem specific API changes the PM count is automatically summarized in a precise manner.

function then has to deal with all the potential behavior of any API the function calls and to implement a consistent behavior under all these circumstances. In these cases, it is not surprising that the code can reflect the difference between the behavior of APIs and developers' understanding, as the consistent behavior from the developers' point of view is actually inconsistent. RID takes advantage of these different paths and reveals any inconsistency in them to detect refcount bugs without knowing the actual consistent behavior expected by the developers.

### 6.3 High Percentage of Incorrect API Usage

The example in Figure 8 is a representative of a class of bugs due to the uncommon specification of the `pm_runtime_get()` and `pm_runtime_get_sync()`. After searching the whole kernel for similar call sites using regular expressions, we find 96 calls to the function with error handling (wrapper functions are excluded). Among these call sites, 67 of them ( 70%) miss the decrement when `pm_runtime_get()` fails (RID has detected 40 of them and the reason why some bugs are missing is discussed in Section 6.4).

Reasons of this unexpectedly high percentage of incorrect uses include the following. (1) The specification of `pm_runtime_get()` is different from most other resource management functions. `pm_runtime_get()` always increments a refcount regardless of its return value, while the common practice for resource management functions is to change nothing if the function returns an error code. (2) Cases are observed where developers of a driver may copy the code from another driver in the same subsystem for common functionality. This allows errors in one driver to be propagated to some others.

By discussing with DPM designers, it is worth noting that the specification of `pm_runtime_get()` has been deliberated and designed in a reasonable way. Balancing the refcount increment on error in the APIs can introduce extra complexity because errors can happen in another thread scheduled by `pm_runtime_get()` after `pm_runtime_get()` has returned.

This clearly shows that, even in relative mature code bases, how an API is commonly used does not necessarily indicates how the API *should* be used. As a result, it is challenging for specification mining techniques to infer the correct pattern in this kind of cases. RID, on the other hand, can reveal these bugs because it checks for IPPs in a function locally, without relying on how APIs are used in other functions.

### 6.4 False Positives and Missed Bugs

The major source of false positives from RID is operations not covered by RID's program abstraction. For example, a function in a driver may have two paths with different changes to a refcount according to a specific bit in a bitmap which tracks options from the user. As we have not considered bit operations in our abstraction yet, conditions on these bits are dropped. This makes the paths look indistinguishable in RID though they can be distinguished by the value in the bitmap. Another example is the lack of data structure operations in our summary. This prevents us from distinguish two paths by whether they insert(remove) an element to(from) a list passed as an argument.

Figure 10 shows an example missed by RID. This bug is found by the brute force search discussed in Section 6.3. RID cannot detect this bug because no inconsistency exists among the paths in `arizona_irq_thread()`. One path in the function returns `IRQ_NONE` with the PM count of `arizona->dev` incremented and the other paths returns `IRQ_HANDLED` with no PM count changes. To detect this bug, RID needs to look at the callers to this function and determine whether every caller handles these two cases properly. However, `arizona_irq_thread()` is an interrupt handler which is called via function pointers, which is not yet covered by RID. It is possible to extend RID with alias analysis so that the above call relation is included in the call graph,

```
irqreturn_t arizona_irq_thread
  (int irq, void *data)
{
  int ret;
  ret = pm_runtime_get_sync(arizona->dev);
  if (ret < 0) {
    dev_err(...);
    return IRQ_NONE;
  }
  ..... // no return here
  pm_runtime_put(arizona->dev);
  return IRQ_HANDLED;
}
```

Figure 10: A bug missed by RID. One path in the function returns IRQ_NONE with a refcount incremented. The other paths returns IRQ_HANDLED with no refcount changes.

Table 1: Function in different categories and paths analyzed in functions

| Category | | Count |
|---|---|---|
| Functions with refcount changes | | 2133 |
| Functions affecting those with refcount changes | analyzed | 1889 |
| | not analyzed | 2803 |
| The others | | 261391 |

Table 2: Comparison between RID and Cpychecker.

| Test Program | Common | Specific to... | |
|---|---|---|---|
| | | RID | Cpychecker |
| krbV-1.0.90 | 48 | 86 | 14 |
| ldap-2.4.20 | 7 | 13 | 1 |
| pyaudio-0.2.8 | 31 | 15 | 1 |
| total | 86 | 114 | 16 |

allowing RID to detect bugs of this kind by checking callers of the functions.

### 6.5 Performance

RID takes 64 minutes to classify the 270k functions in Linux and 67 minutes to analyze the complete Linux kernel with the help of selective analysis presented in Section 5.2. Table 1 shows the number of functions in different categories. The statistics illustrate that we can concentrate on a small portion of the Linux kernel as the other parts have no effects in our analysis. This is because Linux is composed of many functionalities while power management of devices is only one of them. The 2133 functions with refcount changes cover 574 out of 3623 drivers in the Linux kernel we build for evaluation.

### 6.6 Sanity Check: Refcounts in Python/C Programs

Table 2 lists the number of common bugs found by Cpychecker and RID, along with number of bugs detected by only one of the two tools. We check the reports from these tools manually, as is done in [13]. The result shows RID can detect more bugs than Cpychecker, mainly because of the adoption of SSA form which allows RID handle variables with multiple statical assignment. The same phenomenon is observed in the evaluation of Pungi [13]. Some bugs reported by Cpychecker are missed by RID because of the same reason discussed in section 6.4.

## 7. Conclusion

In this paper, we propose inconsistent path pair detection, a novel technique to discover refcount bugs. A prototype, called RID, is implemented based on the method. RID looks for a pair of paths in the same function that, given the same argument, have the same return value but different changes to a refcount. At the core of RID is a summary-based interprocedural analysis that calculates the refcount changes and return value of a function under different constraints on arguments. Our evaluation shows RID detects 83 new bugs in Linux and tends to find bugs caused by developers' misunderstanding on API semantics or improper error handling. RID can also be applied to libraries such as the native implementation of Python/C programs as long as the specifications of refcount APIs are given.

The future work of RID focuses on relaxing the limitations discussed in Section 5.4. Alias analysis can be adopted to complement the call graph. Operations on abstract data structures can be included in summaries to improve the precision of RID on functions with data structure operations. Symbolically executing multiple paths in parallel can be adopted to improve the performance of RID.

## Acknowledgments

This research is supported by Natural Science Foundation of China under Grant No. 61170050, National Science and Technology Major Project of China (2012ZX01039-004) and the National High Technology Research and Development Program of China (2015AA011505).

## References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard version 2.5. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf`.

[2] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. (lia) - model evolution with linear integer arithmetic constraints. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 258–273. Springer Berlin Heidelberg, 2008.

[3] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *ACM SIGPLAN Notices*, volume 42, pages 435–445, June 2007.

[4] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2009.

[5] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In Michael B. Jones and M. Frans Kaashoek, editors, *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, pages 1–16. USENIX Association, 2000.

[6] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[7] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. Scalable and systematic detection of buggy inconsistencies in source code. *ACM SIGPLAN Notices*, 45(10):175–190, October 2010.

[8] Claire Le Goues and Westley Weimer. Specification mining with few false positives. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2009.

[9] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A system and language for building system-specific, static analyses. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 69–82. ACM, 2002.

[10] Mateus Jurczyk. Windows kernel reference count vulnerabilities - case study. `http://j00ru.vexillium.org/dump/zn_slides.pdf`.

[11] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 249–259. ACM, 2008.

[12] Akash Lal and Ganesh Ramalingam. Reference count analysis with shallow aliasing. *Information Processing Letters*, 111(2):57–63, 2010.

[13] Siliang Li and Gang Tan. Finding reference-counting errors in python/C programs with affine analysis. In Richard Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 80–104. Springer, 2014.

[14] Zhenmin Li and Yuanyuan Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 306–315. ACM, 2005.

[15] Huqiu Liu, Yuping Wang, Lingbo Jiang, and Shimin Hu. PF-miner: A new paired functions mining method for android kernel in error paths. In *COMPSAC*, pages 33–42. IEEE, 2014.

[16] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 103–116. ACM, 2007.

[17] D. Malcom. a static analysis tool for cpython extension code. `https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html`.

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

[19] Paul E. McKenney and Jack Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *10th IASTED International Conference on Parallel and Distributed Computing and Systems*, October 1998.

[20] Leonardo Mendona De Moura and Nikolaj Bjrner. *Z3: An Efficient SMT Solver*. Springer, 2008.

[21] Robert Oehlmann. Static single-assignment for program slicing on binary intermediate language. Master's thesis, Hamburg University of Technology, 2013.

[22] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In Nigel Davies, Srinivasan Seshan, and Lin Zhong, editors, *The 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, Ambleside, United Kingdom - June 25 - 29, 2012*, pages 267–280. ACM, 2012.

[23] Python/c api reference manual. `https://docs.python.org/2/c-api/`.

[24] Refcount behavior of python/c apis. `http://svn.python.org/projects/python/trunk/Doc/data/refcounts.dat`.

[25] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia Lawall, and Gilles Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2013*, June 2013.

[26] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Explaining inconsistent code. In Bertrand Meyer, Luciano

Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 521–531. ACM, 2013.

[27] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and ChengXiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[28] Aaron Tomb and Cormac Flanagan. Detecting inconsistencies via universal reachability analysis. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 287–297. ACM, 2012.

[29] Security Tracker. Linux kernel memory leak in inotify_init() lets local users deny service. `http://www.securitytracker.com/id/1025321`.

[30] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 461–476. Springer, 2005.

[31] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.