Reference Count Imbalance Detection without Semantic Assumptions

Junjie Mao

Tsinghua University

5/26/2015

Outline

- Introduction
- Related Work on Detecting Reference Count Bugs
- Inconsistency Analysis
- Implementation
- Preliminary Evaluation
- Future Work

Introduction

- Reference counts (refcount) are widely used in OS kernels from servers to mobile devices, …
  - memory management
  - power management
  - … [see backup pages]
- … but refcount imbalances can lead to critical system failures.
  - out of memory, dangling pointers, …
  - send requests to devices in low-power state
  - etc.

Related Work

- Related works have proposed algorithms for detecting possible refcount imbalances on affine programs with shallow alias [1,2], …

```
int foo(i, x1, x2) {
    if(i > 0) {
        x3 = x1;
    } else {
        x3 = x2;
    }
    if(i > 0) {
        x3->refcount ++;
        return 1;
    }
    return 0;
}
```

affine →

```
global x1, x2

foo() {
    if ? {
        x3 = x1;
    } else {
        x3 = x2;
    }
    if ? {
        x3->refcount ++;
    }
}
```

[1] A. Lal and G. Ramalingam. Reference count analysis with shallow aliasing. Information Processing Letters, 111(2):57–63, 2010
[2] S. Li and G. Tan. Finding reference-counting errors in python/C programs with affine analysis. In ECOOP 2014

Related Work (cont.)

- Related works have proposed algorithms for detecting possible refcount imbalances on affine programs with shallow alias, …
  - … with the assumption that refcounts used should always be balanced.

```
foo() {
    set all refcounts to 0
    ……
    check all refcounts are still 0
        (or other assumed value)
}
```

Related Work (cont.)

- The techniques above do not apply to an OS kernel, however.
  - The entry of an OS kernel is expected to be “noreturn”
  - Functions in an OS kernel may be designed to have non-balanced refcount operations

- How can we know a refcount imbalance is bound to happen **without the semantic assumption**?

Related Work (cont.)

- In [3] and [4], inconsistency among beliefs from different code segments are used to check potential errors statically.

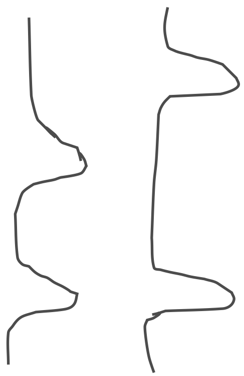
```
bar(x) {
    if (x != NULL) *x;
    ...
    *x;
    ...
}
```

```
lock l;
int a, b;
void foo0 {
    lock(l); a = a + b; unlock(l);
    b = b + 1;
}
void bar() {
    lock(l); a = a + 1; unlock(l);
}
void baz0 {
    a = a + 1;
    unlock(l); b = b - 1; a = a / 5;
}
```

[3] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In SOSP, pages 57–72, 2001.
[4] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In ACM SIGPLAN Notices, volume 42, pages 435–445, June 2007.

Inconsistency Analysis

① ②



x2++ x1++

```
void foo(x1, x2) {
    if ? {
        x3 = x1;
    } else {
        x3 = x2;
    }
    if ? {
        x3->refcount ++;
    }
}
```

a specific path

```
x2->refcount --;
  // x2->refcount should be 0
  // after the decrement
```

- In the affine program, a function with different refcount operations on different path will always raise refcount imbalance.
- However, this too imprecise to be practical.

Inconsistency Analysis (cont.)

```
int foo(i, x1, x2) {
    if(i > 0) {
        x3 = x1;
    } else {
        x3 = x2;
    }
    if(i > 0) {
        x3->refcount ++;
        return 1;
    }
    return 0;
}
```

- Is foo() really buggy? We are still no sure, because:
  - If i > 0, the refcount of x1 is incremented.
  - If i <=0, no refcount is changed.
  - Callers of foo() can determine how foo() will affect refcounts based on the actual parameter passing to foo() or the return value of foo().
  - It is the caller’s responsibility to handle both cases properly.

Inconsistency Analysis (cont.)

- Callers to foo() may determine the refcount operations of foo() by:
  - parameters passed to foo()
  - the return value of foo()
  - side effects of foo() on global data structures

- In this work we focus on the first two cases which assumes all functions are pure except their refcount operations.

Inconsistency Analysis (cont.)

C construct	affine translation in previous work	our affine translation
int foo(a1, …, ak) {…}	foo() {…}	int foo(a1, …, ak) {…}
if (i > 0) {…} else {…}	if ? {…} else {…}	if (i > 0) {…} else {…}
if (bar(…)) {…} else {…}	if ? {…} else {…}	if ? {…} else {…}
return v;	(none)	return v;
x = f(x1, …, xk)	f()	x = f(x1, …, xk)
x = y	x = y	x = y

- We adopt a summary-based inter-procedural function analysis to implement the detection process. Functions are visited in reverse topological order.

Function/ summary Based IPA

- We summarize a function by recording
  - modified refcounts
  - under what path conditions are these refcounts modified, and
  - how the refcounts are modified.
- This form of summary allows us to carry out optimiations on path enumeration which is discussed later.

```
int foo(i, x1, x2) {
    if(i > 0) { x3 = x1; }
    else { x3 = x2; }
    if(i > 0) {
        x3->refcount ++;
        return 1;
    }
    return 0;
}
```

summarize

Summary

refcount	cond / op	
x1	[i]>0 ∧ [1]* =	+1
x1	[i]<=0 ∧ [0] =	0

* [_] represents the return value

Function/ summary Based IPA (cont.)

- Summaries are applied in call sites based on the condition of actual argument and return values

Summary of foo()

refcount	cond / op	
x1	[i]>0 ∧ [1]* =	+1
x1	[i]<=0 ∧ [0] =	0

```
int bar(k, x1, x2) {
    ret = foo(k, x1, x2);
    if (ret == 0) {
        x1->refcount ++;
    }
}
```

- path 1: [k] <= 0 ∧ ret = 0
  - x1
- path 2: [k] > 0 ∧ ret = 1
  - x1 +1

Bug Reporting

- The summaries are calculated in the reverse topological order of the call graph.
- Bugs are reported in the following case

refcount	cond / op	
x1	$[i] > 0 \land [1] =$	0
x1	$[i] > 0 \land [1] =$	+1

- Note that the following case may not be a bug

refcount	cond / op	
x1	$[i] > 0$	0
x1	$[i] = 0$	+1

applied to a path whose condition is $[i] >= 0$

Implementation

- The IPA analysis is implemented based on the Static Single Assignment (SSA) form of Control Flow Graph (CFG)

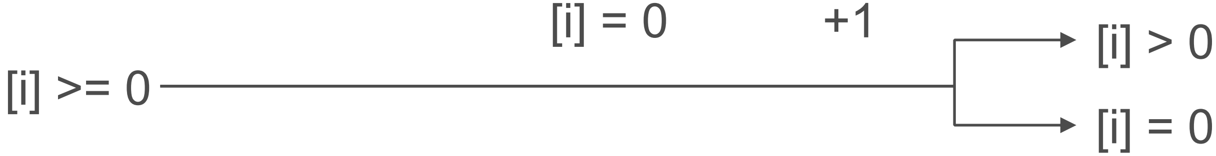
- For each function, all *trails* in the CFG are enumerated and function summaries are applied to the call sites.
  - A *trail* in a CFG is a path from the entry point to the exit point with no repeated edges.

Analysis on a Single Path

- An entry in the summary may correspond to multiple paths in the function.
  - Different paths can have the same path condition and call sites

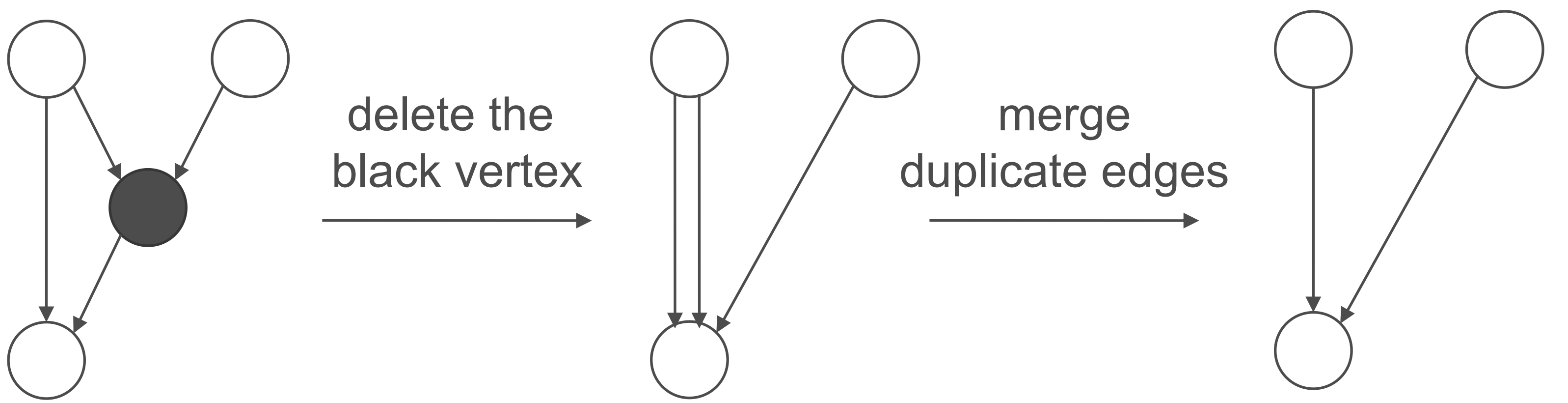
- A path in the function may also correspond to multiple entries in the summary.

refcount	cond / op	
x1	[i] > 0	0
	[i] = 0	+1



Implementation

- However, the number of paths in a CFG is exponential to the number of branches in general.
    - We have met a function with over 1 billion paths in the Linux kernel source
- Before path enumeration, we simplifies the CFG by removing *ineffective* vertices (i.e. basic blocks).



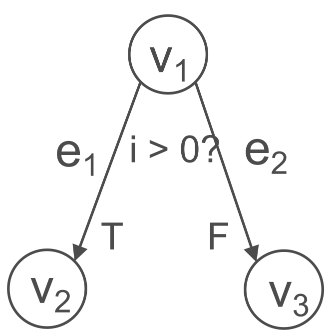
Remove Ineffective Vertices

- We first determine the set of *effective* instructions in a CFG. An instruction is *effective* if one of the following holds
  - It is a call to a function with refcount operations (this instruction defines a variable holding the return value at the same time)
  - It defines a variable which is used in an effective instruction
  - It uses a variable defined by an effective instruction to define another variable
  - It is a comparison on an effective instruction which is used in a branch condition
- Vertices which covers no instruction in the effective instruction set are ineffective, and are deleted on after another during the graph simplification.

Preserving Indirect Data Dependency

- Two sets are attached to each edge
  - pred: set of predicates that held by walking from its source vertex to the target
  - edge: set of edges in the original CFG that will always be passed when walking from its source vertex to the target

- Initialization

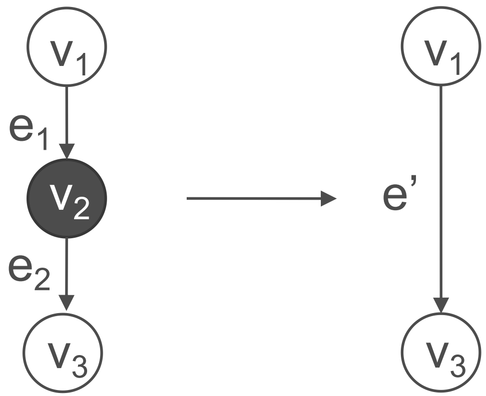


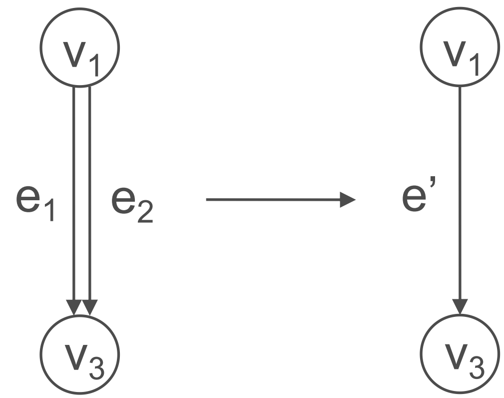
e_1.pred: {i > 0}
e_1.edge: {e_1}

e_2.pred: {i <= 0}
e_2.edge: {e_2}

Preserving Indirect Data Dependency

- For newly created edges during the simplification of the CFG:





e'.pred = $e_1 \cup e_2$
e'.edge = $e_1 \cup e_2$

e'.pred = $e_1 \cap e_2$
e'.edge = $e_1 \cap e_2$

Preserving Indirect Data Dependency

- A trail is represented by a sequence of edges:

$$e_1, e_2, \dots, e_k$$

- Path conditions of a trail (written as *PC*) in the simplified CFG is:

$$AP = \bigcup_{i=1}^{k} e_i.path$$

$$PC = \{p \mid p \in AP \wedge \neg p \notin AP\}$$

- For a ϕ variable in v_k whose in edge is e_k in the trail, the value of this variable is then determined by e_k.edge

Preliminary Evaluation

- Common error types
  - Missing resource release in error-handling path
  - Jumping to the wrong exception handler
  - Wrong assumption on function behavior

Preliminary Evaluation: Example

```
static void g2d_dma_start(struct g2d_data *g2d,
                          struct g2d_runqueue_node *runqueue_node)
{
        struct g2d_cmdlist_node *node =
                    list_first_entry(&runqueue_node->run_cmdlist,
                                     struct g2d_cmdlist_node, list);
        int ret;

        ret = pm_runtime_get_sync(g2d->dev);
        if (ret < 0)
                return;

        writel_relaxed(node->dma_addr, g2d->regs + G2D_DMA_SFR_BASE_ADDR);
        writel_relaxed(G2D_DMA_START, g2d->regs + G2D_DMA_COMMAND);
}
```

Summary

refcount	cond / op	
g2d->dev->power.usage_count	True	+1

Future Work

- Symbolic-execution based summary calculation
- Consideration on common side effects
- More precise analysis on loops

Thanks!

Backup

linux kernel 2.6.5 引入kref

```
struct foo {
  ...
  struct kref kref;
  ...
}
```

```
struct foo *foo;
foo = kmalloc(sizeof(*foo),GFP_KERNEL);
kref_init(&foo->kref, oo_release);
```

```
void foo_release(struct kref *kref)
{  struct foo *foo;
   foo = container_of(foo, struct foo,kref);
   kfree(foo);
}
```

```
struct kref {
atomic_t refcount;
void (*release)(struct kref *kref);
};
```

Backup

linux kernel 2.6.5 引入kref

```
void kref_init(struct kref *kref, void (*release)
(struct kref *kref))
{  WARN_ON(release == NULL);
   atomic_set(&kref->refcount,1);
   kref->release = release;
}
```

```
struct kref *kref_get(struct kref *kref)
{  WARN_ON(!atomic_read(&kref->refcount));
   atomic_inc(&kref->refcount);
   return kref;
}
```

```
void kref_put(struct kref *kref)
{   if (atomic_dec_and_test(&kref->refcount))
        kref->release(kref);
}
```