Empirical study of inter-procedural data flow (IDF) patterns for memory leak analysis in linux

by

Damanjit Singh

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Suraj Kothari, Major Professor

Srikanta Tirthapura

Wensheng Zhang

Iowa State University

Ames, Iowa

2014

DEDICATION

I would like to dedicate this thesis to my parents and my sister. Without their support I would not have been able to complete this work. Also, I would like to thank my friends for their loving guidance during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Suraj Kothari for his guidance, patience and support throughout this research. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education.

Second, Ahmed Tamrawi for his great help and guidance throughout this research and the writing of this thesis. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Srikanta Tirthapura and Dr. Wensheng Zhang. I am also grateful to all the members of my research group Akshay Deepak,Tom Deering,Ben Holland for their support. Finally, I would like to thank all my family members and friends for their love and support.

ABSTRACT

Analysis of inter-procedural data flow (IDF) is a commonly encountered challenge for verifying safety and security properties of large software. In order to address this challenge, a pragmatic approach is to identify IDF patterns that are known to occur in practice, and develop algorithms to detect and handle those patterns correctly. We perform an empirical study to gather the IDF patterns in Linux, which is essential to support such a pragmatic approach.

In our study, we first analyzed the Linux code to study how reference to dynamically allocated memory in a function flows out of the function. We analyzed instances of memory allocation and identified 6 IDF patterns. Second, we mined and analyzed memory leak bug fixes from the Linux git repository. Third, we surveyed the literature for static analysis tools that can detect memory leaks. Based on these studies, we found that the set of IDF patterns associated with the memory leak bug fixes in Linux and those that can be detected by the current static analysis tools is a subset of the 6 IDF patterns we identified.

CHAPTER 1. INTRODUCTION

In program analysis, Data Flow analysis is the process of collecting information about the way the variables are used, defined in the program. In many cases it is of interest to know how a particular data item is used after it is defined in a function. For example the usage of pointer to the allocated memory in a function is of interest to find memory leak. Also, data-flow analysis techniques play an important role in tools for performing optimization, program understanding and maintenance, software testing, and verification of program properties.

If the analysis of the usage of data item is done within the function where the data item is defined then it is called Intra-procedural Data Flow analysis. On the other hand, Inter-procedural data flow analysis extends the scope of data flow analysis across function boundaries. Figure 1.1 shows function *computeAverage*() which computes the average of two numbers and return the result back to the caller function *main*(). The data in the variable *average* which is defined in *computeAverage*() function flows from it to *main* function through return statement, thus requires Inter-procedural data flow analysis to track the data present in *average* variable. The data in variable *avg* which is defined in *computeAveragePrint*() remains within the function. Only Intra-procedural data flow analysis is required to track the data present in *avg* variable.

```java
package com.java;

public class MyClass {
    public static void main(String[] arg){
        int avg = computeAverage(2, 6);

        if(avg>4){
            System.out.println("High Average!!");
        }else{
            System.out.println("Low Average!!");
        }

        computeAveragePrint(2, 6);
    }

    private static int computeAverage(int a, int b){
        int average = 0;
        average = (a+b)/2;
        return average;
    }

    private static void computeAveragePrint(int a, int b){
        int avg = 0;
        avg = (a+b)/2;

        if(avg>4){
            System.out.println("High Average!!");
        }else{
            System.out.println("Low Average!!");
        }

        return;
    }
}
```

Figure 1.1 Program data flow

The work in this empirical study is motivated to provide answers to the following questions:

- What are the different IDF patterns that must be considered for the automated static analysis of memory leaks in Linux kernel?
- How frequent are those patterns?

1.1 Thesis Contributions

This thesis provides the following key contributions:

1. The finding of Inter-procedural Data Flow(IDF) patterns involved in memory leak analysis and their frequency of occurrence in Linux Operating System.

2. Mining and analysis of memory leak bug fixes in Linux.

1.2 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 defines the various IDF patterns that have been found by our empirical study. Chapter 3 discusses each IDF pattern with examples from Linux and the challenges associated with each IDF pattern for memory leak analysis. Chapter 4 discusses our study of Linux memory leak bug fixes. Chapter 5 discusses our study of the literature survey of current static analysis tools. Chapter 6 discusses the related work. Chapter 7 summarizes and concludes our work.

CHAPTER 2. INTER-PROCEDURAL DATA FLOW(IDF) PATTERNS

When a memory is allocated in a function, a pointer to the memory can escape to other threads of execution, or to calling functions. There are different ways in which a pointer(p) to an allocated memory inside a function can escape to other functions. We categorize the escape of variable p from a function into six different IDF patterns. In this chapter we define each IDF pattern and discuss their detail in next chapter.

In all the following examples of IDF patterns, the memory is allocated in the function foo and p is a pointer to the allocated memory.

2.1 Parameter Escape(PEsc):

When p or any variable tainted (a variable p taints variable q when p is assigned to q after the variable p is defined) by it is passed to some other function through its parameter or returned to caller of foo through one of its arguments then we call it as parameter escape . Listing 2.1 and listing 2.2 shows an example of parameter escape.

Listing 2.1 Parameter escape to the caller of function foo()

```
void foo(int *a){
        //some code
        int *p = malloc(sizeof(int)*10);
        a=p;
        //some code
}
```

Listing 2.2 Parameter escape to the function called by foo()

```
void foo(){
        //some code
        int *p = malloc(sizeof(int)*10);
        bar(p);
        //some code
}
```

In listing 1 the pointer p is passed to the caller of foo when its argument a is tainted with p as shown by the statement $a = p$. In listing 2 the pointer p is escaped to function bar when function bar is called and p is passed as one of its parameter.

2.2 Return Escape(REsc):

When p or any variable tainted by it is *returned* from function foo then we classify such an escape of p as return escape. It is called so because in this case the allocated memory escapes to the caller of *foo* through the returned value. Listing 2.3 shows an example of return escape.

Listing 2.3 Return escape

```
int* foo(){
        //some code
        p = malloc(sizeof(int)*10);
        //some code
        return p;
}
```

2.3 Global Escape(GEsc):

When p or any variable tainted by it is assigned to a global variable, we classify such an escape as *global escape*. Listing 2.4 shows an example of global escape.

Listing 2.4 Global escape

```
int *g;//global variable
void foo(){
        //some code
        int *p = malloc(sizeof(int)*10);
        g=p;
        //some code
}
```

2.4 Escape Through Function Pointer(FPEsc):

In this type of IDF pattern, the reference to the allocated memory is passed to the function(f) parameter. The function f is called using function pointer. Listing 2.5 shows the example of such IDF pattern. This pattern uses only PEsc IDF pattern to escape p.

Listing 2.5 Escape through function pointer

```
struct{
int a;
int (*fp)(int*);//fp is function pointer
}myStruct;

void foo(myStruct *s){
        int p = malloc();
        //some code
```

```
        s->fp(p);
        //some computational code
        return;
}
```

In listing 5 the highlighted line shows the call through a function pointer and the pointer to the allocated memory is passed as a parameter.

2.5 Escape By A Pointer To A Field Of Structure(SFEsc):

In this type of IDF pattern, the memory is allocated to a structure s in function foo and instead of a pointer to the s, a pointer to one of its fields is passed using any one of the base IDF pattern from foo. Listing 2.6 shows an example of such IDF pattern.

Listing 2.6 Escape through a field of structure

```
struct{
int a;
struct1 m;
}myStruct;

void foo(){
        myStruct* s = malloc(sizeOf(myStruct));
        //initialize other members of the structure
        return &s->m ;
}
```

In listing 6 first the memory of structure of type $myStruct$ is allocated and assigned to pointer s. Then the address of field m of the structure is returned to the caller of foo.

2.6 Escape By A Pointer Inserted In A Linked List(LLEsc):

When p or any variable tainted by it escapes from function foo by inserting it into a linked list, we classify such an escape as escape through linked list. Pointer p which is inserted to the linked list can escape from foo using any of the base IDF pattern. In listing 2.7 function foo allocates memory to structure of type *myStruct* and assigns the address of the allocated memory to variable s. The pointer s is then added to the linked list l using list_add function.

Listing 2.7 Escape to linked list

```
//list defined globally
struct list{
        list *next, *previous;
}l;
struct myStruct{
int a;
list m;
}

static inline void list_add(struct list_head *new,struct list_head *
   prev,struct list_head *next) {
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}

void foo(){
        myStruct *s = malloc(sizeOf(myStruct));
        //intialise other members of structure myStruct
        list_add(s,l.previous,l.next);
}
```

CHAPTER 3. CHALLENGES ASSOCIATED WITH IDF PATTERNS FOR MEMORY LEAK ANALYSIS IN LINUX KERNEL

In this chapter, we will show the result of our empirical study on Linux. We will show the findings by presenting the example cases of each type of IDF pattern. Also we will show the frequency of occurrence of these patterns in Linux.

For finding the IDF patterns we used Linux (version 3.12) as our base software. We initially used static analysis tool for finding how the pointer to the allocated memory in a function escapes out of it. Using our analysis based on static analysis tool, generated significant number of false positives like the one shown in listing 3.1. In the example shown in listing 3.1 pointer wq points to the allocated memory and taints variable $ei->socket.wq$ at line 14. The last statement of the function returns the address of pointer $ei->vfs\_inode$. Further listing 3.2 shows the recovery of pointer wq from the variable that was returned by the allocating function $sock\_alloc\_inode()$,which is then freed finally. Our tool fails to detect such data flow and shows such type of cases as if the pointer to the allocated memory has not escaped at all.

To more accurately determine IDF patterns we manually analyzed how the pointer to the allocated memory escapes out of the function. We have done our study on 838 instances of $kmalloc()$. Linux uses $kmalloc()$ function call to allocate memory dynamically.

Listing 3.1 False positive from static analysis tool

```
1  static struct inode *sock_alloc_inode(struct super_block *sb) {
2          struct socket_alloc *ei;
3          struct socket_wq *wq;
4          ei = kmem_cache_alloc(sock_inode_cachep, GFP_KERNEL);
5          if (!ei)
6                  return NULL;
7          wq = kmalloc(sizeof(*wq), GFP_KERNEL);
8          if (!wq) {
9                  kmem_cache_free(sock_inode_cachep, ei);
10                 return NULL;
11         }
12         init_waitqueue_head(&wq->wait);
13         wq->fasync_list = NULL;
14         RCU_INIT_POINTER(ei->socket.wq, wq);
15         ei->socket.state = SS_UNCONNECTED;
16         ei->socket.flags = 0;
17         ei->socket.ops = NULL;
18         ei->socket.sk = NULL;
19         ei->socket.file = NULL;
20         return &ei->vfs_inode;
21 }
```

Listing 3.2 Deallocating memory in *sock_destroy_inode*()

```
static void sock_destroy_inode(struct inode *inode) {
        struct socket_alloc *ei;
        struct socket_wq *wq;
        ei = container_of(inode, struct socket_alloc, vfs_inode);
        wq = rcu_dereference_protected(ei->socket.wq, 1);
        kfree_rcu(wq, rcu);
        kmem_cache_free(sock_inode_cachep, ei);
}
```

3.1 Example Cases Of IDF Patterns From Linux

In chapter 2 we had defined the IDF patterns, in this chapter we will discuss the IDF patterns in detail by showing the actual examples of each IDF pattern from Linux.

1. **Parameter Escape**(PEsc): There are 501 memory allocation instances found in Linux kernel as shown in figure 3.3 in which the variable pointing to the address of allocated memory is escaped through parameter only. Listing 3.3 shows example case of PEsc in which the pointer to the allocated memory *data* is passed to the function *usb_control_message*() function. We call it as PEsc to child. For memory leak analysis it is required to track the pointer in *usb_control_message*() as well. This case as we can see is more challenging than Intra-procedural data flow analysis for detecting memory leaks.

Listing 3.3 Parameter escape to child function

```
static void ntrig_report_version(struct hid_device *hdev) {
        int ret;
        char buf[20];
        struct usb_device *usb_dev = hid_to_usb_dev(hdev);
        unsigned char *data = kmalloc(8, GFP_KERNEL);
        if (!data)
                goto err_free;
        ret = usb_control_msg(usb_dev, usb_rcvctrlpipe(usb_dev, 0)
            ,USB_REQ_CLEAR_FEATURE,USB_TYPE_CLASS |
            USB_RECIP_INTERFACE | USB_DIR_IN,0x30c, 1, data, 8,
                                    USB_CTRL_SET_TIMEOUT);
        if (ret == 8) {
                ret = ntrig_version_string(&data[2], buf);
        }
        err_free:
        kfree(data);
}
```

Listing 3.4 shows another example of PEsc IDF pattern in which the pointer to the allocated memory is passed to the parent(caller) of the function.

Listing 3.4 Parameter escape to parent function

```
static int rock_continue(struct rock_state *rs) {
        int ret = 1;
        int blocksize = 1 << rs->inode->i_blkbits;
        const int min_de_size = offsetof(struct rock_ridge, u);
        //some code

        if (rs->cont_extent) {
                struct buffer_head *bh;
                rs->buffer = kmalloc(rs->cont_size, GFP_KERNEL);
                if (!rs->buffer) {
                        ret = -ENOMEM;
                        goto out;
                }
        ret = -EIO;
        bh = sb_bread(rs->inode->i_sb, rs->cont_extent);

        if (bh) {
                memcpy(rs->buffer, bh->b_data + rs->cont_offset,
                   rs->cont_size);
                put_bh(bh);
                rs->chr = rs->buffer;
                rs->len =       rs->cont_size;
                rs->cont_extent = 0;
                rs->cont_size = 0;
                rs->cont_offset = 0;
                return 0;
        }
                printk("Unable␣to␣read␣rock-ridge␣attributes\n");
```

```
        } out:
        kfree(rs->buffer);
        rs->buffer = NULL;
        return ret;
}
```

In listing 3.4 the highlighted lines shows that memory is allocated to the $rs->buffer$. As pointer rs is the argument of the *rock_contine* function so the reference to the memory escapes to the callers of *rock_contine*. Once the pointer to the allocated memory passes to the caller of the function, it is necessary to check all the callers for memory free. Figure 3.1 shows the reverse call graph (RCG) of *rock_contine* function. Thus the challenge involves traversing each path of the RCG of *rock_contine* function to check if allocated memory is freed.

Figure 3.1 Reverse call graph

2. **Return Escape**(REsc):There are 88 memory allocation instances found in Linux kernek as shown in figure 3.3 in which the variable pointing to the address of allo-

cated memory is escaped through return statement. Listing 3.5 shows an example case of Return Escape.

Listing 3.5 Return escape Linux example 1

```
struct nfs_seqid *nfs_alloc_seqid(struct nfs_seqid_counter *
    counter, gfp_t gfp_mask) {  struct nfs_seqid *new;
        new = kmalloc(sizeof(*new), gfp_mask);
        if (new != NULL) {
                new->sequence = counter;
                INIT_LIST_HEAD(&new->list);
                new->task = NULL;
        }
        return new;
}
```

In listing 3.5 the variable *new* points to the allocated memory and is escaped to the caller of *nfs_alloc_seqid* function through return statement.The challenge in handling this IDF pattern for memory leak analysis is similar to that of PEsc to parent IDF pattern, as the pointer to the allocated memory escapes to the caller of the function.

Listing 3.6 shows another example case of return escape.

Listing 3.6 Return escape Linux example 2

```
static void *esp_alloc_tmp(struct crypto_aead *aead, int nfrags,
    int seqihlen) {        unsigned int len;
        len = seqihlen;
        len += crypto_aead_ivsize(aead);
        //some code
        len += sizeof(struct scatterlist) * nfrags;
        return kmalloc(len, GFP_ATOMIC);
}
```

3. **Global Escape**(GEsc):There are 25 memory allocation instances found in Linux kernel as shown in figure 3.3 in which the variable pointing to the address of allocated memory is escaped through global variable. Listing 3.7 shows one of example case of Global Escape.

Listing 3.7 Global escape Linux example 1

```
static struct usb_class {
        struct kref kref;
        struct class *class; } *usb_class;

static int init_usb_class(void) {
        int result = 0;
        if (usb_class != NULL) {
                kref_get(&usb_class->kref);
                goto exit;
        }

        usb_class = kmalloc(sizeof(*usb_class), GFP_KERNEL);
        if (!usb_class) {
                result = -ENOMEM;
                goto exit;
        }
        kref_init(&usb_class->kref);
        usb_class->class = class_create(THIS_MODULE, "usbmisc");

        if(IS_ERR(usb_class->class)) {
                result = PTR_ERR(usb_class->class);
                printk(KERN_ERR "class_create␣failed␣for␣usb␣
                   devices\n");
                kfree(usb_class);
                usb_class = NULL;
                goto exit;
```

```
        }

        usb_class->class->devnode = usb_devnode;
        exit:
        return result;
}
```

Listing 3.8 Memory deallocation function

```
static void release_usb_class(struct kref *kref) {
        class_destroy(usb_class->class);
        kfree(usb_class);
        usb_class = NULL;
}
```

In the example shown in listing 3.7 the allocated memory is assigned to variable *usb_class* which is a pointer to the global structure of type *usb_class*. Once the reference to the allocated memory is escaped through GEsc pattern, the scope of access of the memory reference becomes global and can be accessed by all the functions that can access the global pointer. In this case as the *usb_class* pointer is statically defined global variable so it can be accessed by all the functions present in the file where the pointer is defined. So the challenge for memory leak analysis not only involves analyzing the callers of the function(*init_usb_class*) but also to analyze all the functions that can refer the global pointer. Listing 3.8 shows the function the frees the allocated memory.

Listing 3.9 shows another example of global escape in which the address to the allocated memory is assigned to the global pointer irc_buffer.

Listing 3.9 Global escape Linux example 2

```
static char *irc_buffer;//global declaration
```

```
static int __init nf_conntrack_irc_init(void) {
        int i, ret;
        if (max_dcc_channels < 1) {
                return -EINVAL;
        }
        irc_exp_policy.max_expected = max_dcc_channels;
        irc_exp_policy.timeout = dcc_timeout;
        irc_buffer = kmalloc(65536, GFP_KERNEL);
        if (!irc_buffer)
                return -ENOMEM;
        /* If no port given, default to standard irc port */
        if (ports_c == 0)
                ports[ports_c++] = IRC_PORT;
        for (i = 0; i < ports_c; i++) {
                //some code
        }
        return 0;
}
```

4. **Escape through function pointer**(FPEsc): We have found 36 instances of memory allocation as shown in figure 3.4 in which the pointer to allocated memory is escaped to another function(f) . The function f is called using function pointer. Function pointers makes it difficult to track which function is called using conventional call flow graphs. It is easy to build a call graph of A-calls-B when the call statement explicitly mentions B. It is much harder to handle indirect calls. Listing 3.10 shows one of many such cases from Linux kernel.

Listing 3.10 Escape through function pointer Linux example 1

```
struct dm_dirty_log *dm_dirty_log_create(const char *type_name,
        struct dm_target *ti, int (*flush_callback_fn)(struct
            dm_target *ti),
```

```
        unsigned int argc, char **argv) {
        struct dm_dirty_log_type *type;
        struct dm_dirty_log *log;
        log = kmalloc(sizeof(*log), GFP_KERNEL);
        //some code
        log->flush_callback_fn = flush_callback_fn;
        log->type = type;
        if (type->ctr(log, ti, argc, argv)) {
                kfree(log);
                put_type(type);
                return NULL;
        }
        return log;
}
```

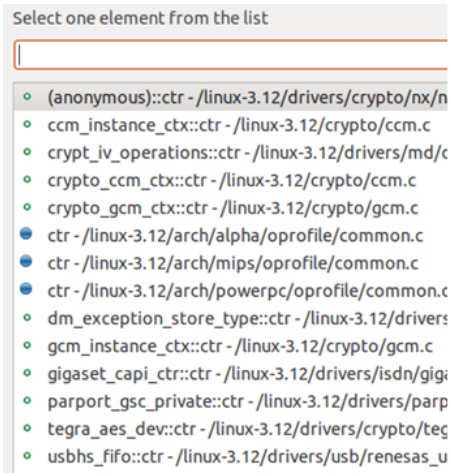

Figure 3.2 Possible calls by function pointer

In listing 3.10 the variable *log* is allocated and then passed as parameter to another function using call to function pointer $type->ctr$. Call through function pointer is not a direct(static) function call, in a way that the information about the function present in function pointer has to be extracted. Figure 3.2 shows the list of possible

functions that can be called by calling $type->ctr$ function pointer. Determining which specific function is called needs accurate data flow analysis of function pointer variable.

Listing 3.11 shows another example of such IDF pattern.

Listing 3.11 Escape through function pointer Linux example 2

```
static int fill_thread_core_info(struct elf_thread_core_info *t,
    const struct user_regset_view *view, long signr, size_t *total)
     {
     //some code
          for (i = 1; i < view->n; ++i) {
                    //some code
                    void *data = kmalloc(size, GFP_KERNEL);
                    ret = regset->get(t->task, regset, 0, size, data,
                        NULL);
                    //some code
          }
          return 1;
}
```

5. **Escape by a pointer to a Field of Structure**(SFEsc):

   (a) We have found 10 example cases of SFEsc pattern in Linux. Listing 3.12 shows an example of SFEsc pattern. In listing 3.12, structure *gcred* points to the allocated memory and only the address of its field $gcred->gc\_base$ is returned to the caller function. The caller of *generic_create_cred* now have the reference of one of the field of the structure *gcred*.

Listing 3.12 Escape by a pointer to a field of structure Linux example 1

```
static struct rpc_cred * generic_create_cred(struct rpc_auth *
    auth, struct auth_cred *acred, int flags) {
```

```
        struct generic_cred *gcred;
        gcred = kmalloc(sizeof(*gcred), GFP_KERNEL);
        if (gcred == NULL)
                return ERR_PTR(-ENOMEM);
        rpcauth_init_cred(&gcred->gc_base,acred,&generic_auth
            ,&generic_credops);
        gcred->gc_base.cr_flags = 1UL << RPCAUTH_CRED_UPTODATE
            ;
        gcred->acred.uid = acred->uid;
        gcred->acred.gid = acred->gid;
        gcred->acred.group_info = acred->group_info;
        gcred->acred.ac_flags = 0;
        if (gcred->acred.group_info != NULL)
                get_group_info(gcred->acred.group_info);
        gcred->acred.machine_cred = acred->machine_cred;
        gcred->acred.principal = acred->principal;
        dprintk("RPC:allocatedscredpforuiddgidd\n",gcred->acred.
            machine_cred?"machine":  "generic",gcred,
        from_kuid(&init_user_ns, acred->uid),
        from_kgid(&init_user_ns, acred->gid));
        return &gcred->gc_base;
}
```

The allocated memory is freed in the function generic_free_cred() as shown in listing 3.13. *container_of* is a macro defined in Linux which uses pointer arithmetic to determine address of allocated memory of structure type *s* from its field address. In the current example as shown in listing 3.13, the address of allocated memory of structure *generic_cred*(represented by *gcred*) is calculated from the variable *cred* (which holds the address of $gcred- > gcbase$)by subtracting its offset value from the *cred* .The offset value is simply the number of bytes a field of structure is away from the start of the structure.

Listing 3.14 shows the generic macro which is used in Linux for calculating the offset.

Listing 3.13 Freeing by offset calculation

```
static void generic_free_cred(struct rpc_cred *cred) {
        struct generic_cred *gcred = container_of(cred, struct
            generic_cred, gc_base);
        if (gcred->acred.group_info != NULL)
                put_group_info(gcred->acred.group_info);
        kfree(gcred);
}
```

Listing 3.14 Linux *container_of* macro example

```
#define container_of(ptr, type, member) ({\
         const typeof( ((type *)0)->member ) *__mptr = (ptr
            );\
         (type *)( (char *)__mptr - offsetof(type,member) )
            ;
})
```

Listings 3.15 and 3.16 shows more examples of SFEsc.

Listing 3.15 Escape through a field of structure Linux example 2

```
static struct cache_deferred_req *svc_defer(struct cache_req *
   req) {
        struct svc_rqst *rqstp = container_of(req, struct
           svc_rqst, rq_chandle);
        struct svc_deferred_req *dr;
        if (rqstp->rq_arg.page_len || !rqstp->rq_usedeferral)
                return NULL; /* if more than a page, give up
                    FIXME */
        if (rqstp->rq_deferred) {
```

```
                dr = rqstp->rq_deferred;
                rqstp->rq_deferred = NULL;
        } else {
                size_t skip;
                size_t size;            /* FIXME maybe discard
                    if size too large */
                size = sizeof(struct svc_deferred_req) + rqstp
                   ->rq_arg.len;
                dr = kmalloc(size, GFP_KERNEL);
                //some code
        }
                svc_xprt_get(rqstp->rq_xprt);
                dr->xprt = rqstp->rq_xprt;
                rqstp->rq_dropme = true;
                dr->handle.revisit = svc_revisit;
                return &dr->handle;
}
```

Listing 3.16 Escape through a field of structure Linux example 3

```
void alloc_acpi_hp_work(acpi_handle handle, u32 type, void *
   context, void (*func)(struct work_struct *work)) {
        struct acpi_hp_work *hp_work;
        int ret;
        hp_work = kmalloc(sizeof(*hp_work), GFP_KERNEL);
        if (!hp_work)
                return;
        hp_work->handle = handle;
        hp_work->type = type;
        hp_work->context = context;
        INIT_WORK(&hp_work->work, func);
        ret = queue_work(kacpi_hotplug_wq, &hp_work->work);
        if (!ret)
```

```
                kfree(hp_work);
}
```

(b) There is no need to calculate the offset if the pointer to the address of first field is escaped. This is because the address of first field of a structure is same as that of parent structure. As shown in listing 3.17 variable p of type proc_mount is allocated in mounts_open_common() function. One of the highlighted line shows that the allocated memory is escaped to $file$ parameter when address of first field of p is assigned to file->private_data.

Listing 3.17 Escape through a field of structure special case

```
static int mounts_open_common(struct inode *inode, struct
   file *file,int (*show)(struct seq_file *, struct
   vfsmount *)){
        struct proc_mounts *p;
    //some code//
        *p = kmalloc(sizeof(struct proc_mounts),
           GFP_KERNEL);
        file->private_data = &p->m;//the allocated
           variable is escaped to file structure
        //some code
}
```

Listing 3.18 shows the way allocated memory address is retrieved and freed. In line 2 pointer m points to the starting address of allocated proc_mount structure. This is because $file- > private\_data$ contains the address of the first field of proc_mount structure and the first field of any structure will have the same address as that of parent structure. So in line 4 when $kfree(m)$ is called this will actually free the memory allocated to pointer p of listing 3.17 although the pointer m is declared of type seq_file and not the original type

proc_mounts.

Listing 3.18 Deallocation of memory pointed by field of structure

```
1 int seq_release(struct inode *inode, struct file *file) {
2         struct seq_file *m = file->private_data;
3         kvfree(m->buf);
4         kfree(m);
5         return 0;
6 }
```

The difference between the previous two example cases discussed, lies in the way the pointer to the allocated memory is extracted from its field. In example shown in listing 3.13, using *container_of* macro the pointer to allocated memory is retrieved which is of same type as that of allocated structure(*generic_cred* in our case), but in case of second example the pointer of allocated variable when retrieved is not of the same type as that of allocated structure. This case uses the fact that the address of first field of a structure can be used to free the parent structure. So although the type of variable m in listing 3.18 is not that of *proc_mounts* still call to $kfree(m)$ will free the memory allocated to pointer p of type *proc_mounts*. An attempt to match the allocation and deallocation sites by matching the types of pointers involved may work (albeit with false positives) in example (a), however, such a strategy will fail in case (b).

For memory leak analysis SFEsc pattern involves additional complexity of recovering the address of allocated memory of structure from its field address.

6. **Escape by a Pointer inserted in a Linked List**(LLEsc): We have found 69 instances of memory allocation in which the pointer to allocated memory is escaped by adding it to Linked List. In the example shown in listing 3.19, it can be seen

that the address of List which is a field of allocated structure $\&new- > list$ is added to global list nfs_referral_count_list.

Listing 3.19 Escape by a pointer inserted in a linked list Linux example 1

```
static int nfs_referral_loop_protect(void) {
        struct nfs_referral_count *p, *new;
        int ret = -ENOMEM;
        new = kmalloc(sizeof(*new), GFP_KERNEL);
        if (!new)
                goto out;
        new->task = current;
        new->referral_count = 1;
        ret = 0;
        spin_lock(&nfs_referral_count_list_lock);
        p = nfs_find_referral_count();
        if (p != NULL) {
                if (p->referral_count >= NFS_MAX_NESTED_REFERRALS)
                        ret = -ELOOP;
                else
                        p->referral_count++;
        } else {
                list_add(&new->list, &nfs_referral_count_list);
                new = NULL;
        }
        spin_unlock(&nfs_referral_count_list_lock);
        kfree(new);
        out:
        return ret;
}
```

Listing 3.20 shows method *nfs_referral_loop_unprotect* which fetches the allocated variable from global list (corresponding to the memory allocation in listing

3.19, and frees it. Here, method $nfs\_find\_referral\_count$ is called which traverses the global list $nfs\_referral\_count\_list$ and returns the pointer to the current $nfs\_referral\_count$ variable. The returned pointer (variable p in Listing 3.20) points to the memory location to be freed. Listing 3.21 shows another example of such IDF pattern.

Listing 3.20 Dereferencing pointer from list

```
static void nfs_referral_loop_unprotect(void) {
        struct nfs_referral_count *p;
        spin_lock(&nfs_referral_count_list_lock);
        p = nfs_find_referral_count();
        p->referral_count--;
        if (p->referral_count == 0)
                list_del(&p->list);
        else
                p = NULL;
        spin_unlock(&nfs_referral_count_list_lock);
        kfree(p);
}
```

Listing 3.21 Escape by a pointer inserted in a linked list Linux example 2

```
static void quirk_awe32_add_ports(struct pnp_dev *dev,struct
   pnp_option *option, unsigned int offset) {
        struct pnp_option *new_option;
        new_option = kmalloc(sizeof(struct pnp_option), GFP_KERNEL
           );
        if (!new_option) {
                return;
        }
        *new_option = *option;
        new_option->u.port.min += offset;
        new_option->u.port.max += offset;
```

```
            list_add(&new_option->list, &option->list);
            pnp_option_set(option));
}
```

In addition to the challenges mentioned for base IDF patterns for memory leak analysis, the additional challenge involves in handling LLEsc IDF pattern is that the static analysis tool needs to check that all the references to the allocated memory that has been added to the collection(List) should be freed eventually. There is no easy way to track the individual memory reference once it is added to the list. For example in the example shown in listing 3.20 on completion of each task the pointer to the allocated memory p is removed from the list and then freed using $kfree()$. So each time when reference to the allocated memory is recovered from the list, it is freed immediately after deleting the reference from the list.Thus at the end when all the pointers to the allocated memory are deleted, their is no memory reference left unfree.

Figures 3.3 and 3.4 shows the frequency distribution of IDF patterns in Linux. Appendix at the end list all the functions in Linux analyzed for empirical study.

Note: There are 11 instances of memory allocation in which the pointer to the allocated memory is escaped through combination of PEsc and LLEsc. i.e. PEsc $\cap$ LLEsc = 11. Similarly, LLEsc $\cap$ REsc = 4, LLEsc $\cap$ GEsc = 1 and FPEsc $\cap$ PEsc = 30. Also we have found 13 instances of memory allocation in which the pointer to the allocated memory does not escape from the function. i.e. there is no inter-procedural data flow involved for such instances of memory allocation. So total memory instances analyzed are $708 + 117 + 13 = 838$.

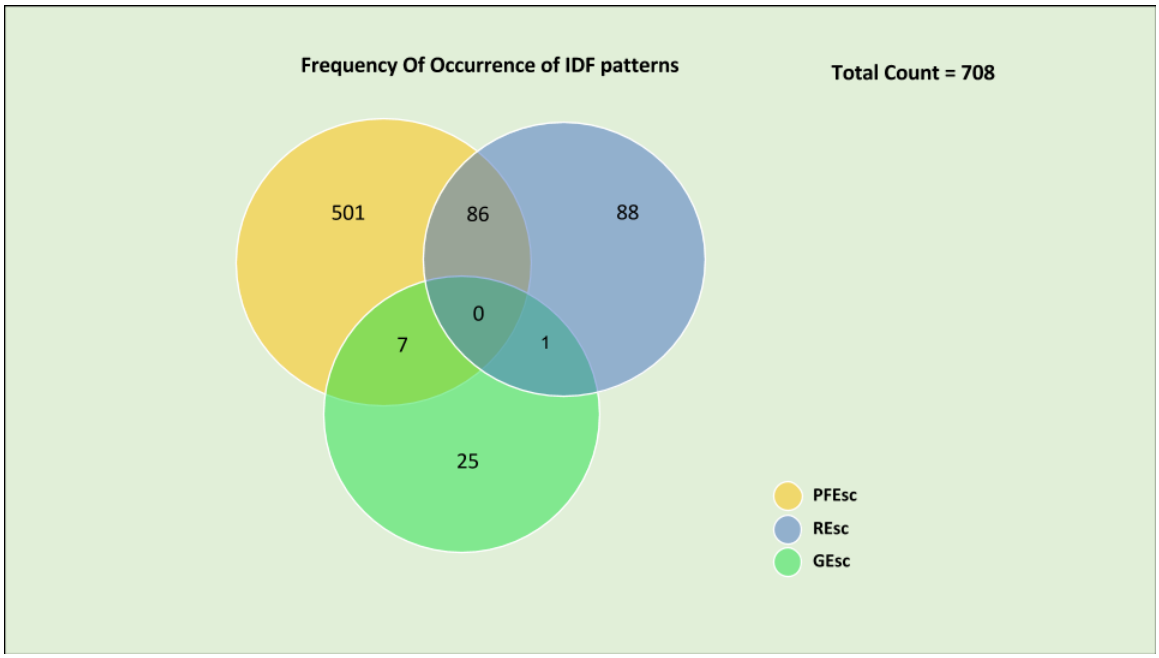

Figure 3.3 Frequency distribution of IDF patterns PEsc, REsc, GEsc

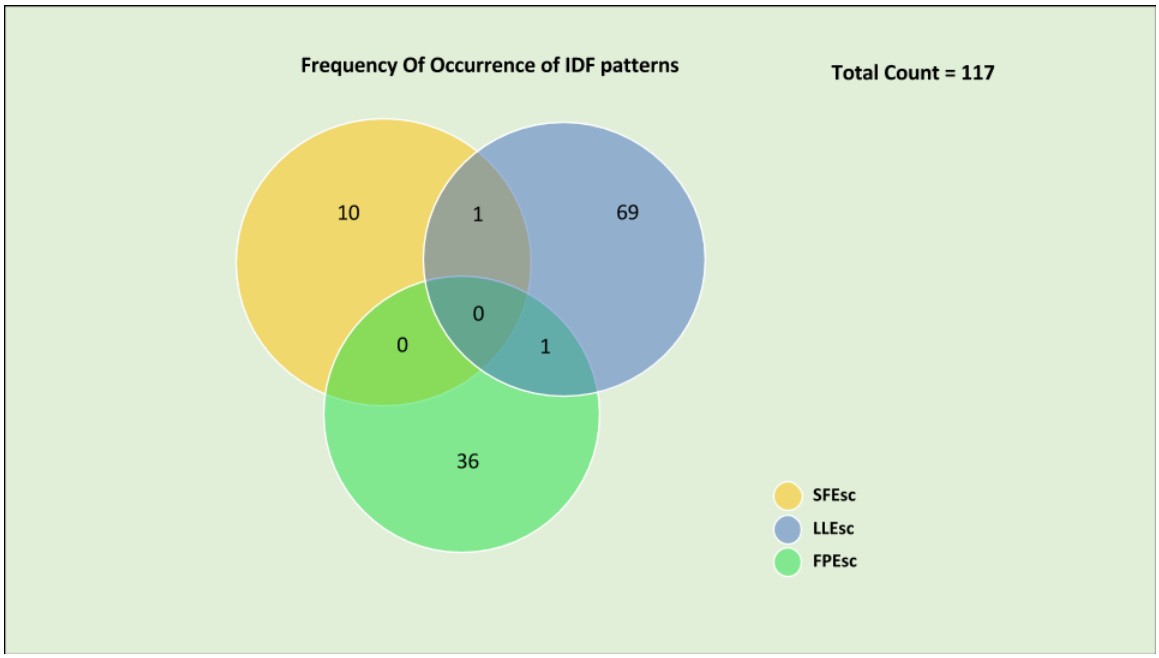

Figure 3.4 Frequency distribution of IDF patterns SFEsc, LLEsc, FPEsc

CHAPTER 4. STUDY OF MEMORY LEAK FIXES IN LINUX KERNEL

We performed this study to check the IDF patterns associated with memory leak bug fixes in Linux. In order to analyze memory leak related bug fixes in Linux we chose Linux git repository.

We use the git version control for Linux to check for the commits that are related to memory leaks. We have analyzed 1200 cases of memory leak related commits.

After setting up the Linux repository we searched all the commits that are related to memory leaks. We used 'git log –grep="memory leak"' command to search for all the logs of commits that contain the word memory leak. We then manually analyzed each commit to check if the fix provided for the bug is related to memory leak. With this we found 1200 cases of memory leak related bug fixes in the master branch. We then analyzed each case to check if there exists any pattern for such fixes.

4.1 Results

Out of 1200 bug fixes we found a pattern of bug fixes that are related to providing fix in error path of a method. The example for such cases is shown in listing 4.1.

Listing 4.1 Memory leak fix in error path example 1

```
struct vport *ovs_vport_alloc(int priv_size, const struct vport_ops *
    ops, const struct vport_parms *parms) {
        vport = kzalloc(alloc_size, GFP_KERNEL);
```

```
        //some code
        if (ovs_vport_set_upcall_portids(vport, parms->upcall_portids))
            {
                kfree(vport);
                return ERR_PTR(-EINVAL);
        }
        //some code
        return vport;
}
```

Listing 4.2 shows another case of memory leak fix provided in error path. The line in orange color is deleted, lines in yellow color are added after the fix.

Listing 4.2 Memory leak fix in error path example 2

```
static int pxa_ssp_probe(struct snd_soc_dai *dai) {
        //some code
        priv = kzalloc(sizeof(struct ssp_priv), GFP_KERNEL);
        //some code
        ssp_handle = of_parse_phandle(dev->of_node, "port", 0);
        if (!ssp_handle) {
                return -ENODEV;
                ret -ENODEV;
                goto err_priv;
        }
        //some code
        err_priv:
        kfree(priv);
        return ret;
}
```

Examples in listings 4.1 and 4.2 shows that the pointer to the allocated memory is either passed to another function through parameter which involves PEsc IDF pattern or it is not escaped at all which is the case of intra-procedural data flow.

There also exists a pattern of bug fixes in which memory allocated to a structure is freed without freeing the memory allocated to its field and vice versa, an example is shown in listing 4.3.

Listing 4.3 Partial memory free example 1

```
void kvm_arch_vcpu_free(struct kvm_vcpu *vcpu) {
        hrtimer_cancel(&vcpu->arch.comparecount_timer);
        kvm_vcpu_uninit(vcpu);
        kvm_mips_dump_stats(vcpu);
        kfree(vcpu->arch.guest_ebase);
        kfree(vcpu->arch.kseg0_commpage);
        kfree(vcpu);
}
```

Listing 4.4 shows an example in which developer frees the memory associated with structure without freeing its member's memory. So the colored statement shown in the figure is added to fix the associated memory leak.

Listing 4.4 Partial memory free example 2

```
void sta_info_free(struct ieee80211_local *local, struct sta_info *sta)
    {
        if (sta->rate_ctrl)
                rate_control_free_sta(sta);
        kfree(rcu_dereference_raw(sta->sta.rates));
        kfree(sta);
}
```

The IDF pattern involved in examples shown in listings 4.3 and 4.4 is PEsc because the pointer to the allocated memory is the argument of the function called to free the memory.

Listing 4.5 shows LLEsc IDF pattern involved in memory leak bug fix. The list $info- > zone\_list$ is traversed using the macro $list\_for\_each\_entry\_safe$ and the ref-

erence to the allocated memory is recovered from the list which is then freed in the statement $kfree(publ)$;

Listing 4.5 LLEsc IDF pattern associated with bug fix

```
static void tipc_purge_publications(struct name_seq *seq)  {
        struct publication *publ, *safe;
        struct sub_seq *sseq;
        struct name_info *info;
        if (!seq->sseqs) {
                nameseq_delete_empty(seq);
                return;
        }
        sseq = seq->sseqs;
        info = sseq->info;
        list_for_each_entry_safe(publ,safe,&info->zone_list,zone_list){
             tipc_nametbl_remove_publ(publ->type,publ->lower,publ->node,
           publ->ref, publ->key);
                kfree(publ);
        }
}
```

All the bug fixes that we have analyzed have IDF patterns out of the six patterns that we have characterized in Chapter 3.

CHAPTER 5. STUDY OF STATIC ANALYSIS TOOLS

In this section we present analysis of our literature survey on static analysis tools to see the IDF patterns associated with example cases in which tools claims to fix memory leaks.

The paper by Sui et al Sui et al. (2012) developed a tool called SABER which uses sparse value flow analysis to detect memory leaks in C programs. Sparse value flow analysis is different from data flow analysis. The later tracks the flow of values iteratively at each point through the control flow while the former tracks the flow of values sparsely through def-use chains or SSA(Static Single Assignment) form. Figure 5.1 shows a leaky code in "icecast" software in which the memory is allocated at lines 174 and 176. The pointer "*entry*" to the allocated memory is then passed as parameter to the function *avl_insert* function. The leak is in one of the error path at line 122 where the function simply returns without freeing the memory. This scenario involves PEsc IDF pattern.

Figure 5.2 shows a leaky code from "wine" software. In function $OLEPictureImpl\_LoadGif$, $GifOpen$ is called at line 1021 so that two heap objects are allocated at lines 898 and 905. One of the two objects is passed to gif and the other to the field private of GifFile. At the end of $OLEPictureImpl\_LoadGif$, there is a call to $DGifCloseFile$ to free the two objects. However, there is a test at line 1030 sitting between the two calls. The two objects are never freed when this test evaluates to true. This scenario involves REsc IDF pattern as the pointer $GifFile$ is assigned to the allocated memory in function $DGIFOPEN()$ and is returned to the caller of $DGIFOPEN()$ through return statement.

The paper by Yungbum Jung and Yi (2008) Jung developed a tool called SPARROW

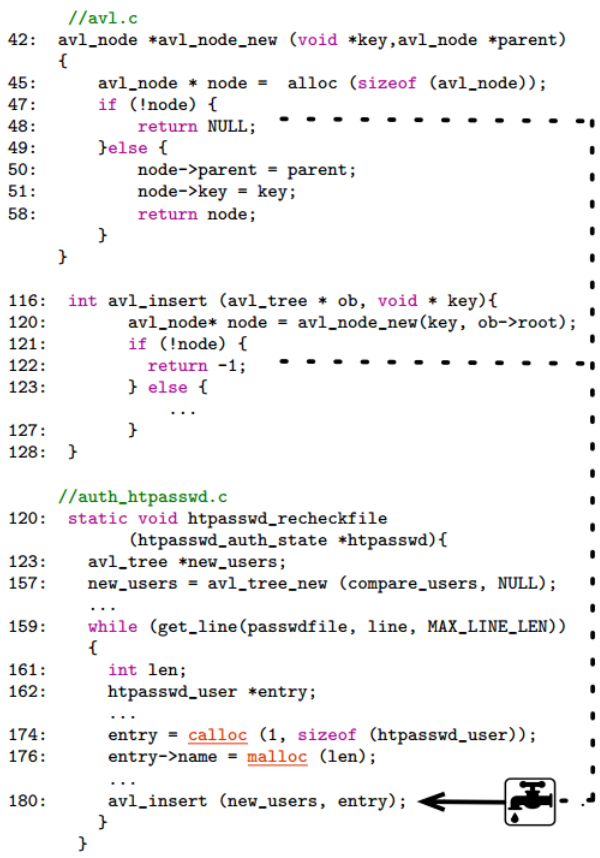

Figure 5.1 Memory leak handled by SABER tool example 1

that detects memory leaks in C programs using function summaries. The tool summarize each function while preserving its memory behavior. This function summary is then used at each call site of the function to analyze memory behavior inter-procedurally. There are many examples in the paper showing the representation of summaries when the allocated memory is returned to the caller function. Figure 5.3 shows a code from "mesa" program containing two memory leaks. First leak occurs at line 273 when in one of the error path, function returns without freeing the pointer *osmesa*. The second leaks occurs when some heap allocations by the function *g1_create_context* are not freed by the function *g1_destroy_context*. To detect such leak, analysis of both *g1_create_context* and *g1_destroy_context* functions is required. The tool in the paper claims to detect both memory leaks. It can be interpreted from the example that the tool can handle both

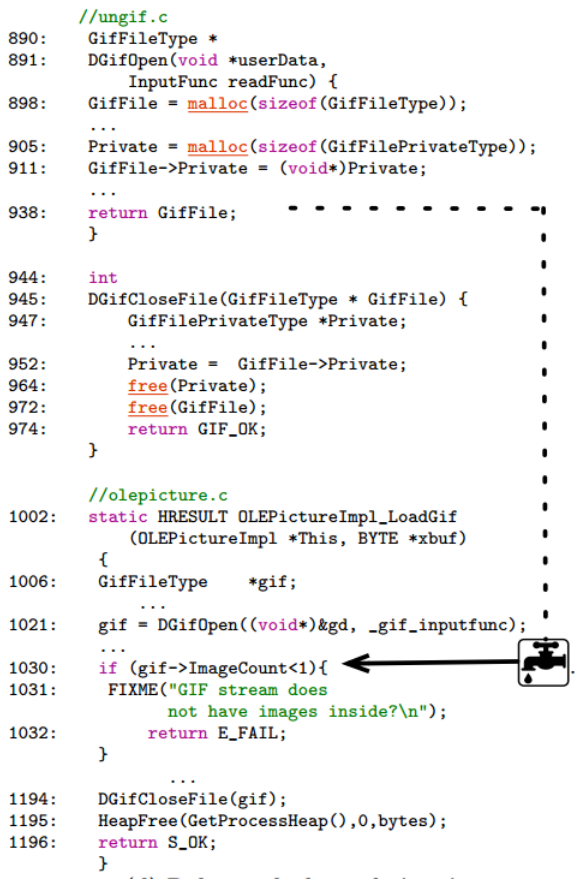

Figure 5.2 Memory leak handled by SABER tool example 2

REsc and PEsc IDF patterns because function *g1_create_context* **returns** the reference to the allocated memory and this reference is passed as **parameter** to the function *g1_destroy_context* for deallocation.

The paper by Yichen Xie and Aiken (2005) developed a tool called SATURN that can perform path and context sensitive memory leak analysis, in addition to detecting memory leaks in error paths of a function's control flow. The tool uses abstraction and boolean satisfiability to achieve inter-procedural path sensitivity. Figure 5.4 shows example cases of memory leaks that SATURN claims to detect. Part (a) of the code in figure 5.4 shows the memory leaks occurs in error path which does not involve any IDF pattern. In part (b) the reference to the allocated memory is returned and is assigned to the pointer *longfilename*. In part (c) the pointer to the allocated memory is passed

```
261: osmesa = (OSMesaContext) calloc( 1, sizeof( ...
262: if (osmesa) {
263:   osmesa->gl_visual = gl_create_visual( rgbmode,
     ...
272:   if (!osmesa->gl_visual) {
273:     return NULL;
274:   }

276:   osmesa->gl_ctx = gl_create_context( ...
     ...
279:   if (!osmesa->gl_ctx) {
280:     gl_destroy_visual( osmesa->gl_visual );
281:     free(osmesa);
282:     return NULL;
283:   }
284:   osmesa->gl_buffer = gl_create_framebuffer( ...
285:   if (!osmesa->gl_buffer) {
286:     gl_destroy_visual( osmesa->gl_visual );
287:     gl_destroy_context( osmesa->gl_ctx );
288:     free(osmesa);
289:     return NULL;
290:   }
```

Figure 5.3 Memory leaks handled by SPARROW tool

to another function through parameter. Clearly the example shown in part (b) and (c) involves REsc and PEsc IDF pattern respectively.

```
1 /* Samba – libads/ldap.c:ads_leave_realm */
2 host = strdup(hostname);
3 if (...) { ...; return ADS_ERROR_SYSTEM(ENOENT); }
```

(a) The programmer forgot to free host on error.

```
1 /* Samba – client/clitar.c:do_tarput */
2 longfilename = get_longfilename(finfo);
3 ...
4 return;
```

(b) get_longfilename allocates new memory.

```
1 /* Samba – utils/net_rpc.c:rpc_trustdom_revoke */
2 domain_name = smb_xstrdup(argv[0]);
3 ...
4 if (!trusted_domain_password_delete(domain_name))
5     return -1;
6 return 0;
```

(c) trusted_domain_password_delete does not free.

Figure 5.4 Memory leaks handled by SATURN tool

The paper by Sigmund Cherem et al. (2007) et al presents a technique that tracks the flow of values from allocation points to deallocation points using a sparse representation of the program consisting of a value flow graph. This graph captures def-use relations

and value flows via program assignments. Figure 5.5 shows an example of leaky code which is claimed to be handled by the tool. At line 292 call to *concat*() function allocates the memory(as shown at line 58), the reference to which is returned and stored in object *f_list* . Next line 293 calls *concat*() again which overwrites *f_list* with the reference to the new memory location, the reference to the old memory is thus lost resulting in memory leak. Fixing this memory leak involves handling of REsc IDF pattern.

```
     /* file "c-aux-info.c" */
53:  char* concat (char* s1, char* s2) {
54:      if (!s1) s1 = "";
55:      if (!s2) s2 = "";
56:      int size1 = strlen (s1);
57:      int size2 = strlen (s2);
58:      char* ret_val = malloc (size1 + size2 + 1);
59:      strcpy (ret_val, s1);
60:      strcpy (&ret_val[size1], s2);
61:      return ret_val;
62:  }
...  ...
281: char* gen_formal_list_for_func_def(tree fdecl) {
282:    char* f_list = "";
...     ...
290:    while (fdecl) {
291:        if (...)
292:            f_list = concat(f_list, ",");
293:        f_list = concat(f_list, formal);
...         ...
302:    }
303:    return f_list;
304: }
```

Figure 5.5 Leaky code example

It can be seen from the above analysis that no new IDF pattern is involved in the examples covered by the papers on static analysis tools.

CHAPTER 6. RELATED WORK

Kang Gui and Suraj Kothari Gui and Kothari (2010) identified two patterns indicative of good software design with respect to matching pair property such as memory leak. The empirical study of Linux kernel is done to check the existence of such patterns.

Andy Chou et al. Chou et al. (2001) presented the result of the empirical study of Operating System errors in Linux kernel. The study discussed various parameters of software bugs like life time of a bug, distribution of software bugs in Linux Kernel etc. Also the study presented how the bugs like memory leaks, null pointer de-referencing etc are distributed across the various modules of Linux kernel.

Neil Brown Brown (2009) discussed various design patterns used in the development of Linux kernel. The patterns deals with the life time of object. The article also relates how the understanding of design patterns is important for memory leak analysis.

Dor et al. use Dor et al. (2000) TVLA, a shape analysis tool based on 3-valued logic, to prove the absence of memory leaks and other memory errors in several list manipulation programs. The paper also presents challenges for memory leak analysis if list manipulation is involved, similar to the LLEsc pattern we discussed. Their analysis verifies these programs successfully, but is intra-procedural and cannot be applied to recursive and multi-procedure programs. Of these analysis [Heine and Lam (2003), Hackett and Rugina (2005)] target referencing leaks; and [Xie and Aiken (2005), Dor et al. (2000)] target reach-ability leaks.

Winter Winter et al. (2013) et al. presented path sensitive data flow analysis to verify memory leaks. Hind et al. Hind et al. (1999) presented flow sensitive algorithm

for Inter-procedural pointer alias analysis. The paper also presents techniques to track the pointer in the presence of function pointers.

In addition to the papers discussed in section five, Das et al. Das et al. (2002) and Engler at al. Engler et al. (2000) employs data flow analysis techniques to statically verify memory leaks. In addition to that there are some open source tools like [Marjamaki, Cla, Spl] that are used to detect memory leaks in software systems. Similarly Sparse static analysis Spa tool is specifically implemented to find fault in Linux Kernel.

Nathaniel et al Ayewah et al. (2007) evaluated the accuracy of static analysis tools by analyzing the warnings provided by the tools.

CHAPTER 7. CONCLUSION

We have observed various ways in which a pointer to the allocated memory escapes from one function to other. We have categorized such escape cases into six Inter-procedural Data Flow(IDF) patterns. We have discussed about the challenges involved in memory leak analysis in the presence of each pattern. Our study of memory leak bug fixes in Linux kernel and the leaks that can be detected by current state of the art static analysis tools reveals four out of six patterns which we have identified. We believe that such IDF patterns will serve as a reference to static analysis tools to achieve higher accuracy for performing automated memory leak analysis.

BIBLIOGRAPHY

Clang: Static analysis tool. `http://clang-analyzer.llvm.org/`. 39

Sparse: Static analysis tool for linux kernel. `http://kernelnewbies.org/Sparse/`. 39

Splint: Static analysis tool for c. `http://splint.org/`. 39

Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y. (2007). Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8, New York, NY, USA. ACM. 39

Brown, N. (2009). Comprehensive and efficient protection of kernel control data. *Information Forensics and Security, IEEE Transactions on*. 38

Cherem, S., Princehouse, L., and Rugina, R. (2007). Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 480–491, New York, NY, USA. ACM. 36

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. (2001). An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA. ACM. 38

Das, M., Lerner, S., and Seigle, M. (2002). Esp: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Notices*, volume 37, pages 57–68. ACM. 39

Dor, N., Rodeh, M., and Sagiv, S. (2000). Checking cleanness in linked lists. In *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*, pages 115–134. 38

Engler, D., Chelf, B., Chou, A., and Hallem, S. (2000). Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, Berkeley, CA, USA. USENIX Association. 39

Gui, K. and Kothari, S. (2010). An Empirical Study to Discover Patterns for Checking the Matching Pair Property. In *International Conference on Computational Intelligence and Software Engineering*. 38

Hackett, B. and Rugina, R. (2005). Region-based shape analysis with tracked locations. In *ACM SIGPLAN Notices*, volume 40, pages 310–323. ACM. 38

Heine, D. L. and Lam, M. S. (2003). A practical flow-sensitive and context-sensitive c and c++ memory leak detector. *ACM SIGPLAN Notices*, 38(5):168–181. 38

Hind, M., Burke, M., Carini, P., and deok Choi, J. (1999). Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21. 38

Jung, Y. and Yi, K. (2008). Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 131–140, New York, NY, USA. ACM. 33

Marjamaki, D. Cppcheck: Static analysis tool. 39

Sui, Y., Ye, D., and Xue, J. (2012). Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 254–264, New York, NY, USA. ACM. 33

Winter, K., Zhang, C., Hayes, I. J., Keynes, N., Cifuentes, C., and Li, L. (2013). Path-sensitive data flow analysis simplified. In *Formal Methods and Software Engineering*, pages 415–430. Springer. 38

Xie, Y. and Aiken, A. (2005). Context- and path-sensitive memory leak detection. *SIGSOFT Softw. Eng. Notes*, 30(5):115–125. 35, 38

APPENDIX. FUNCTIONS IN LINUX ANALYZED FOR EMPIRICAL STUDY

This appendix will list all the functions that have been analyzed in Linux to find IDF patterns, further we have categorized each function with the type of IDF pattern associated with the pointer to the allocated memory in the function.

- Functions in which the allocated variable escape through IDF Pattern *SFEsc* are:

Table .1 Functions involve SFEsc IDF pattern

Function	Pointer to the allocated memory
i915_gem_set_tiling	obj.bit_17
mounts_open_common	p
svc_defer	dr
rsc_alloc	rsci
ip_map_alloc	i
generic_create_cred	gcred
svcauth_gss_register_pseudoflavor	new
unix_domain_find	new
spi_schedule_dv_device	wqw
sock_alloc_inode	wq

- Functions in which the allocated variable escape through IDF Pattern *LLEsc* are:

Table .2 Functions involve LLEsc IDF pattern

Function	Pointer to the allocated memory
__i915_add_request	request
create_pid_cachep	pcache
drm_prime_add_buf_handle	member
serio_queue_event	event
scsi_complete_async_scans	data
open	bb.buffer
pm_vt_switch_required	entry
drm_add_fake_info_node	node
__hw_addr_create_ex	ha
acpi_add_id	id
region_chg	nrg
drm_addctx	ctx_entry
usb_driver_set_configuration	req
register_kretprobe	inst
read_cis_cache	cis
postfix_append_op	elt
usbhid_modify_dquirk	q_new
kcore_update_ram	ent
usb_hub_clear_tt_buffer	clear
add_conn_list	p
postfix_append_operand	elt
sunrpc_cache_pipe_upcall	buf

- Functions in which the allocated variable escape through IDF Pattern *FPEsc* are:

Table .3 Functions involve FPEsc IDF pattern

Function	Pointer to the allocated memory
ethtool_get_stats	data
seq_read	m.size
seq_read	m.buf
cgroup_write_string	buffer
genl_family_rcv_msg	attrbuf
setkey_unaligned	buffer
fifo_set_limit	nla
xfrm_user_policy	data
slave_update	uctl
e1000_dump_eeprom	data
pneigh_lookup	n
inode_doinit_with_dentry	context
soft_cursor	ops.cursor_src
ahash_def_finup	priv
con_font_get	font.data
ethtool_set_eeprom	data
hidinput_led_worker	buf
shash_setkey_unaligned	buffer
snd_mixer_oss_build_input	uinfo
rngapi_reset	buf
ethtool_self_test	data
snd_mixer_oss_build_test	info

Table .4 Functions involve multiple IDF patterns

Functions	Pointer to the allocated memory	IDF Pattern
__netpoll_setup	npinfo	LLEsc, FPEsc
agp_3_5_enable	cur	SFEsc, LLEsc

Complete list of functions for each IDF pattern is publicly shared at:

`https://drive.google.com/folderview?id=0B2krxqxu-hmXVEpxM01TVXdNN1E&usp=drive_web`