Finding Resource-Release Omission Faults in Linux

Suman Saha¹, Julia Lawall², and Gilles Muller¹³
LIP6-Regal¹/DIKU²/INRIA³
Reliability of Linux

- Linux is a widely used OS.
- **Reliability** of code used in Linux is critical.
  - Handling transient run-time errors is essential

Error-Handling Code?
Error Handling Code

Error Handling code handles exceptions.
- Returns the system to a coherent state.

```
static int __init reipl_init(void) {
    ...
    reipl_kset = kset_create_and_add(...)
    ...
    if (rc) {
        kset_unregister(reipl_kset);
        return rc;
    }
    rc = reipl_ccw_init();
    if (rc)
        return rc;
    ...
}
```

- Mistakes cause deadlocks and memory leaks
- Key to ensuring reliability
Issues

- Faults in error-handling code cause deadlocks and memory leaks

- Error-handling code is not tested often
  - Research has shown there are many faults in error-handling code [OOPSLA:04]

- Fixing these faults requires knowing what kind of error-handling code is required
Best known approach: Data-Mining

- Data-mining is used to find protocols in source code.
  - For example, `kmalloc` and `kfree` often occur together

- Evaluate a potential protocol using threshold values

- Use statistics-based analysis to find probable protocols

- The identified protocols are used to find faults in source-code
Protocol with lower threshold values

- The data-mining based approach is not likely to detect this fault
- `wl1251_alloc_hw()` is used only twice
  - Once with this releasing operation and once without

```c
... hw = wl1251_alloc_hw();
... if(ret < 0) {
    ...
    goto out_free;
}
...
if(!w1->set_power) {
    ...
    return -ENODEV;
}
...
out_free:
    ieee80211_free_hw(hw);
return ret;
```

drivers/net/wireless/wl12xx/wl1251_spi.c
Our Work

- **Goal**: Detect resource-release omission faults in error-handling code

- **Approach**: Use information about error-handling code found within the same function

- We may have false negatives, if there is no model of the correct error handling code in the same function
Detecting Resource-Release Omission Faults

1. Identify error-handling code

```c
x = kmalloc(...);
...
if(!y) {
    kfree(x);
    ff();
    return NULL;
}
a->b = x;
m = a;
...
if(!z) {
    ff();
    return NULL;
}
```
Detecting Resource-Release Omission Faults

1. Identify error-handling code
2. Collect all Resource-Release operations

Function list

... x = kmalloc(...);
... if(!y) {
    kfree(x);
    ff();
    return NULL;
}

a->b = x;
m = a;
...
... if(!z) {
    ff();
    return NULL;
}
Detecting Resource-Release Omission Faults

1. Identify error-handling code

2. Collect all Resource-Release operations

3. Compare each block of error-handling code to the set of all Resource-Release operations

<table>
<thead>
<tr>
<th>Function list</th>
</tr>
</thead>
<tbody>
<tr>
<td>kfree(x);</td>
</tr>
<tr>
<td>ff();</td>
</tr>
</tbody>
</table>

```
... x = kmalloc(...);
...
if(!y) {
    kfree(x);
    ff();
    return NULL;
}
a->b = x;
m = a;
...
if(!z) {
    ff();
    return NULL;
}
```
Detecting Resource-Release Omission Faults

1. Identify error-handling code

2. Collect all Resource-Release operations

3. Compare each block of error-handling code to the set of all Resource-Release operations

Function list

- kfree(x);
- ff();

\[
\cdots
x = kmalloc(...);
\]
\[
\cdots
\text{if}(!y) \{
\quad \text{kfree}(x);
\quad \text{ff}();
\quad \text{return NULL;}
\}
\]
\[
a->b = x;
\]
\[
m = a;
\]
\[
\cdots
\]
\[
\text{if}(!z) \{
\quad \text{ff}();
\quad \text{return NULL;}
\}
\]
Detecting Resource-Release Omission Faults

1. Identify error-handling code
2. Collect all Resource-Release operations
3. Compare each block of error-handling code to the set of all Resource-Release operations
4. Analyze the omitted operation to determine whether it is an actual fault

```c
... x = kmalloc(...);
... if(!y) {
    kfree(x);
    ff();
    return NULL;
}
a->b = x;
m = a;
...
if(!z) {
    ff();
    return NULL;
}
```
Analyze Omitted Releasing Operations

In some cases, omitted operations are not actually faults

- The variable holding the resource is undefined or has a different definition at the point of the error-handling code
- The released resource is returned by the error-handling code.
- The resource is released in an alternate way
4. Analyze Omitted Releasing Operations

Four alternate ways

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
</tr>
</thead>
</table>
| \[\ldots\]
  \[\ldots\]
  \[x = kmalloc(\ldots);\]
  \[\ldots\]
  \[if(!y) \{\]
  \[\quad \text{kfree}(x);\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\]
  \[\text{kfree}(x);\]
  \[\ldots\]
  \[if(!z) \{\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\] | \[\ldots\]
  \[\ldots\]
  \[x = kmalloc(\ldots);\]
  \[\ldots\]
  \[if(!y) \{\]
  \[\quad \text{kfree}(x);\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\]
  \[\text{free}(x);\]
  \[\ldots\]
  \[if(!z) \{\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\] | \[\ldots\]
  \[\ldots\]
  \[x = kmalloc(\ldots);\]
  \[\ldots\]
  \[if(!y) \{\]
  \[\quad \text{kfree}(x);\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\]
  \[\text{a->b = x;}\]
  \[\ldots\]
  \[if(!z) \{\]
  \[\quad \text{cleanup}(a);\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\] | \[\ldots\]
  \[\ldots\]
  \[x = kmalloc(\ldots);\]
  \[\ldots\]
  \[if(!y) \{\]
  \[\quad \text{kfree}(x);\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\]
  \[\text{ret} = \text{chk}(\ldots x \ldots);\]
  \[\text{if(ret)} \{\]
  \[\quad \text{ff}();\]
  \[\quad \text{return NULL;}\]
  \[\}\]
Results

<table>
<thead>
<tr>
<th></th>
<th>Total reports</th>
<th>Faults</th>
<th>FP</th>
<th>TODO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faults</td>
<td>126</td>
<td>103</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>Function</td>
<td>78</td>
<td>65</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

Table: Total number of Faults, False Positives (FP), and TODO. Experiments have been done on the *drivers* directory of Linux 2.6.34

Few false positives (16%)
The Benefit of Context Sensitivity

The tool found 331 resource allocations for which at least one releasing operation seems to be omitted.

- 22.4% of them are released by two different operations and 4.2% by three. (Scenario 2)

- 5.2% of them are released via another pointer. (Scenario 3)

- 15.7% of them are released by intervening functions. (Scenario 4)

- 14.8% of them are released by a call to another function that is defined in the same file.
Our Strategy VS Data-Mining Strategy

- Detected 103 faults are associated with 30 protocols
- Only 2 protocols are valid using the threshold values
- So, only 10 faults can be identified
## Impact of Detected Faults

Categorize associated functions in respect to frequencies of use

<table>
<thead>
<tr>
<th>Category</th>
<th>% of detected faults</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>61%</td>
<td>13% of the init functions is dynamic</td>
</tr>
<tr>
<td>open</td>
<td>17%</td>
<td>Attacker can execute repeatedly</td>
</tr>
<tr>
<td>ioctl</td>
<td>14%</td>
<td>Attacker can execute repeatedly</td>
</tr>
<tr>
<td>read/write</td>
<td>8%</td>
<td>Have the highest impact</td>
</tr>
</tbody>
</table>
Conclusion

• We have focused on context-sensitivity constraints on the choice of resource-releasing operations and used this as a guideline for finding faults.
• Taking context-sensitivity into account significantly reduces the number of false positive.
An Approach to improving the Structure of Error-Handling Code in the Linux

This work helps to reduce mistakes in error-handling code