Improving the Quality of Error-Handling Code in Systems Software using Function-Local Information

Suman Saha
Laboratoire d'Informatique de Paris 6
Regal

25th March, 2013
Outline

• Motivation

• Contribution 1:
  Understanding error-handling code in systems software

• Contribution 2:
  Improving the structure of error-handling code

• Contribution 3:
  Finding omission faults in error-handling code

• Future work and Conclusion
Motivation

Research Questions:

1. Why are the faults in error-handling serious?
2. Why is it difficult to identify them?
Reliability of Systems Code

Reliability of systems code is critical
- Handling transient run-time errors is essential
- Cause deadlocks, memory leaks and crashes
- Key to ensuring reliability
Issues

Error-handling code is not tested often

- Research has shown there are many faults in error-handling code [Weimer OOPSLA:04]
- Fixing these faults requires knowing what kind of error-handling code is required
Existing work on Error-Handling Code

- Proposing new language features [Bruntink ICSE:06]
  - introducing macros

- Finding faults in Error-Handling Code
  - focused on error-detection and propagation [Gunawi FAST:08, Banabic EuroSys:12]
Error-Handling Code in C Programs

Error-Handling code handles exceptions.

- Returns the system to a coherent state.

```c
param = copy_dev_ioctl(user);
...
err = validate_dev_ioctl(command, param);
if (err)
    goto out;
...
fn = lookup_dev_ioctl(cmd);
if (!fn) {
    AUTOFS_WARN("...", command);
    return -ENOTTY;
}
...
out:
    free_dev_ioctl(param);
return err;
```

Autofs4 code containing a fault
Understanding Error-Handling Code in Systems Software

Research Questions:

1. How is error-handling code important for systems software?
2. What are the typical ways to write error-handling code in systems software?
## Considered Systems Software

<table>
<thead>
<tr>
<th>SL</th>
<th>Project</th>
<th>Lines of code</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linux drivers</td>
<td>4.6 MLoC</td>
<td>2.6.34</td>
<td>Linux device drivers</td>
</tr>
<tr>
<td>2</td>
<td>Linux sound</td>
<td>0.4 MLoC</td>
<td>2.6.34</td>
<td>Linux sound drivers</td>
</tr>
<tr>
<td>3</td>
<td>Linux net</td>
<td>0.4 MLoC</td>
<td>2.6.34</td>
<td>Linux networking</td>
</tr>
<tr>
<td>4</td>
<td>Linux fs</td>
<td>0.7 MLoC</td>
<td>2.6.34</td>
<td>Linux file systems</td>
</tr>
<tr>
<td>5</td>
<td>Wine</td>
<td>2.1 MLoC</td>
<td>1.5.0</td>
<td>Windows emulator</td>
</tr>
<tr>
<td>6</td>
<td>PostgreSQL</td>
<td>0.6 MLoC</td>
<td>9.1.3</td>
<td>Database</td>
</tr>
<tr>
<td>7</td>
<td>Apache httpd</td>
<td>0.1 MLoC</td>
<td>2.4.1</td>
<td>HTTP server</td>
</tr>
<tr>
<td>8</td>
<td>Python</td>
<td>0.4 MLoC</td>
<td>2.7.3</td>
<td>Python runtime</td>
</tr>
<tr>
<td>9</td>
<td>Python</td>
<td>0.3 MLoC</td>
<td>3.2.3</td>
<td>Python runtime</td>
</tr>
<tr>
<td>10</td>
<td>PHP</td>
<td>0.6 MLoC</td>
<td>5.4.0</td>
<td>PHP runtime</td>
</tr>
</tbody>
</table>
Error-Handling in Python
Percentage of code found within functions that have 0 or more blocks of error-handling code.
Error-Handling: Basic Strategy

A typical, initial way to write error-handling code

Basic strategy

```c
x = alloc();
...
if(!y) {
    free(x);
    return -ENOMEM;
}
...
if(!z) {
    free(x);
    return -ENOMEM;
}
...
```
Basic Strategy: Problems

Basic strategy

```c
x = alloc();
...
if(!y) {
    free(x);
    return -ENOMEM;
}
...
if(!z) {
    free(x);
    return -ENOMEM;
}
...
```

Problems

- Duplicates code
Basic Strategy: Problems

Basic strategy

```c
x = alloc();
...
if(!y) {
    free(x);
    return -ENOMEM;
}
if(!m) {
    free(x);
    return -ENOMEM;
}
...
if(!z) {
    free(x);
    return -ENOMEM;
}
...
```

Problems

- Duplicates code
- Obscures what error handling code to use for new operations
x = alloc();
...
n = alloc();
...
if(!y) {
    free(n);
    free(x);
    return -ENOMEM;
}
if(!m) {
    free(n);
    free(x);
    return -ENOMEM;
}
if(!z) {
    free(n);
    free(x);
    return -ENOMEM;
}

Basic Strategy: Problems

Basic strategy

Problems

- Duplicates code
- Obscures what error handling code to use for new operations
- Requires lots of changes when adding a new operation
Error-Handling: Goto-based Strategy

Goto-based strategy

```
x = alloc();
...
if(!y)
    goto out;
...
if(!z)
    goto out;
...
out:
    free(x);
    return -ENOMEM;
```

- State-restoring operations appear in a single labelled sequence at the end of the function
Goto-based Strategy: Benefits

Goto-based strategy

```c
x = alloc();
...
if(!y)
    goto out;
...
if(!z)
    goto out;
...
out:
    free(x);
    return -ENOMEM;
```

- State-restoring operations appear in a single labelled sequence at the end of the function
- No code duplication
Goto-based Strategy: Benefits

Goto-based strategy

```c
x = alloc();
...
if(!y)
    goto out;
...
if(!m)
    goto out;
if(!z)
    goto out;
...
out:
    free(x);
    return -ENOMEM;
```

- State-restoring operations appear in a single labelled sequence at the end of the function
- No code duplication
Goto-based strategy

```c
x = alloc();
...

n = alloc();
...

if(!y)
    goto out;
...

if(!m)
    goto out;
if(!z)
    goto out;
...

out:
    free(n);
    free(x);
    return -ENOMEM;
```

- State-restoring operations appear in a single labelled sequence at the end of the function
- No code duplication
Basic vs Goto-based strategies in Linux

![Graph showing the comparison between Basic and Goto-based strategies in Linux versions 2.0 to 3.6. The x-axis represents the Linux versions, and the y-axis represents the number of functions. The graph indicates the number of functions that only return in if branches, only goto in if branches, and both goto and return in if branches.]
Basic vs Goto-based strategies in Python

The chart compares the number of functions in different Python versions, categorized by whether they only return in if branches, only goto in if branches, or both goto and return in if branches. The chart shows a significant increase in the number of functions that use goto and return in if branches as Python versions progress.
Summary

- The number of functions with error-handling code is increasing version by version

- Many more functions use the basic strategy than use the goto strategy
  - Leads to a lot of duplicate code
  - Difficult to maintain
  - Error prone
  - Hard to debug
Improving the Structure of Error-Handling Code in Systems Software

Three steps:

1. Find error handling code
2. Identify operations for sharing
3. Perform transformation
1. Find Error Handling Code

- No recognizable error handling abstractions in C code
- Heuristics:
  - An if branch ending in a return
  - An if branch containing at least one non-debugging function call (something to share)

Examples:

```c
if(ns->bacct == NULL) {
    ...  
    if(acct == NULL) {
        filp_close(file, NULL);
        return -ENOMEM;
    }
}
```

```c
if(ns->bacct == NULL) {
    ns->bacct = acct;
    acct = NULL;
}
```
2. Identify Operations for Sharing

For each branch

1. Extract the code that is specific to the error condition
2. Extract the code that can be shared with other error handling code (cleanup code)

```c
if(x) {
    ret = -ENOMEM;
    h("s");
    f(a);
    return ret;
}
```

```c
if(x) {
    ret = -ENOMEM;
    h("s");
    goto out;
}
```

```c
out:
    f(a);
    return ret;
```
Reasons for no sharing

1. No state restoring operations

```c
if (x) {
    ret = -ENOMEM;
    h("s");
    return ret;
}
```

```c
out:
    return ret;
```
Reasons for no sharing

2. Only one branch to transform

```c
f() {
... 
  x = allocate();
... 
  y = noallocate();
  if(y) {
    ... 
    free(x);
    return ret;
  }
}
```
3. Nothing in common with other error handling code
3. Transformation

Classify the branches according to how difficult they are to transform

1. Simple
2. Hard
3. Harder
4. Hardest
3. Transformation: Simple

- Exactly same code in the branch and the label
- Reduce duplicate code by reusing the existing label

```c
... if (!sl->data) {
    clear_bit(n, sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
}
...
out:  
    clear_bit(n, sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

```c
... if (!sl->data)  
    goto out;
...
out:  
    clear_bit(n, sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```
3. Transformation: Hard

- Code in the branch is a subset of the code in the label
- Reduce duplicate code by creating a new label in existing code

```c
... if (!sl->data) {
    unlock_kernel();
    return ret;
}
...
out: clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

```c
... if (!sl->data) {
    unlock_kernel();
    goto out1;
}
...
out: clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```

```c
3. Transformation: Hard
```
3. Transformation: Harder

- Branches do have similar code but no label has
- Reduce duplicate code by creating a new label and moving code to that label

```c
... if (!sl->data) {
    clear_bit(n, sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
}
...
if (!ent) {
    goto out;
}
...
if (!sl->data) {
    clear_bit(n, sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
}
...
if (!ent) {
    goto out;
    kfree(sl->data);
    clear_bit(n, sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
}
...
out:
kfree(sl->data);
clear_bit(n, sbi-symlink_bitmap);
unlock_kernel();
return ret;
```
3. Transformation: Hardest

- Combination of **Simple** (common code in branch and label) and **Harder** (noncommon code in them).

```c
... if (!ent){
    kfree(sl->data);
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
}
...
return 0;
out:
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
...
```

```c
... if (!ent)
    goto out1;
...
return 0;
out1:
    kfree(sl->data);
    clear_bit(n,sbi-symlink_bitmap);
    unlock_kernel();
    return ret;
```
Results

- Applied to 7 widely used systems including Linux, Python, Apache, PHP and PostgreSQL
- **46%** of basic strategy functions have only one *if*. So, those are not transformed
- **54%** of basic strategy functions are taken for transformation
  - **59%** of these are not transformed due to lack of sharing
  - **41%** are transformed
Summary

We proposed an automatic transformation that converts basic strategy error-handling code to the goto-based strategy

- The algorithm identifies many opportunities for code sharing

What about possible defects in error-handling code?
Finding Omission Faults in Error-Handling Code

[PLOS11, DSN13]
Omission Faults in Error-Handling Code

param = copy_dev_ioctl(user);
...
err = validate_dev_ioctl(command, param);
if (err)
    goto out;
...
fn = lookup_dev_ioctl(cmd);
if (!fn) {
    AUTOFS_WARN("...", command);
    return -ENOTTY;
}
...
out:
    free_dev_ioctl (param);
    return err;

Autofs4 code containing an omission fault

Challenge

- Identify the needed code

Omission Fault
Best known approach: Data-Mining

- Use data mining to find protocols
  - For example, *kmalloc* and *kfree* often occur together
- Use the protocols satisfying threshold values or identified by statistics-based analysis
- The identified protocols are used to find faults in source code
- Engler *SOSP:01*, Ammons *POPL:02*, Li *FSE:05*, Yang *ICSE:06*
Problem: Protocols with low threshold values

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

```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;
```
Our approach: HECtor

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

- **Approach**: Use correct error-handling code (*exemplar*) found within the same function
  - What is needed nearby is likely to be needed in the current if as well
  - We may have false negatives, if there is no exemplar
Detecting Resource-Release Omission Faults

The algorithm has 4 Steps
Step 1: 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;
}
```
Step 2: Detecting Resource-Release Omission Faults

1. Identify error-handling code

2. Collect all Resource-Release operations

```c
x = kmalloc(...);
...
if(!y) {
    kfree(x);
    ff();
    return NULL;
}
a->b = x;
m = a;
...
if(!z) {
    ff();
    return NULL;
}
```
Step 3: 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

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

Function list:

- `kfree(x);`
- `ff();`

```c
if(!z) {
    ff();
    return NULL;
}
```
Step 3: 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;
}
```

Omitted: `kfree(x)`
Step 4: 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;
}
```

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

In some cases, omitted operations are not actually faults
Case 1: Variable with Different Definitions

The variable holding the resource is undefined or has a different definition at the point of the error-handling code.

```c
...  
x = kmalloc(...);
...
if(!y) {
    kfree(x);
    ff();
    return NULL;
}
...
x = y;
...
if(!z) {
    ff();
    return NULL;
}
```
Case 2: Return the Resource

The released resource is returned by the error-handling code.

```c
...
x = kmalloc(...);
...
if(!y) {
kfree(x);
ff();
return NULL;
}
...
if(z) {
ff();
return x;
}
```
Case 3: Alternate Ways to Release

**Scenario 1**

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

**Scenario 2**

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

**Scenario 3**

```c
... x = kmalloc(...);
... if(!y) {
    kfree(x);
    ff();
    return NULL;
}
a->b = x;
... if(!z) {
    cleanup(a);
    ff();
    return NULL;
}
```

**Scenario 4**

```c
... x = kmalloc(...);
... if(!y) {
    kfree(x);
    ff();
    return NULL;
}
... ret = chk(...x...);
if(ret) {
    ff();
    return NULL;
}
```
Example

param = copy_dev_ioctl(user);
...
err = validate_dev_ioctl(command, param);
if (err)
    goto out;
...
fn = lookup_dev_ioctl(cmd);
if (!fn) {
    AUTOFS_WARN("...", command);
    return -ENOMEM;
}
...
out:
    free_dev_ioctl (param);
    return err;

- **param** has the same definition in both blocks.
- No return statement with the resource.
- No alternate way to release the resource.
## Results

<table>
<thead>
<tr>
<th></th>
<th>Reports</th>
<th>Faults</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linux drivers</strong></td>
<td>293 (180)</td>
<td>237 (152)</td>
<td>56</td>
</tr>
<tr>
<td><strong>Linux sound</strong></td>
<td>32 (19)</td>
<td>19 (13)</td>
<td>13</td>
</tr>
<tr>
<td><strong>Linux net</strong></td>
<td>13 (13)</td>
<td>7 (7)</td>
<td>6</td>
</tr>
<tr>
<td><strong>Linux fs</strong></td>
<td>47 (34)</td>
<td>22 (17)</td>
<td>25</td>
</tr>
<tr>
<td><strong>Python (2.7)</strong></td>
<td>17 (13)</td>
<td>13 (11)</td>
<td>4</td>
</tr>
<tr>
<td><strong>Python (3.2.3)</strong></td>
<td>22 (13)</td>
<td>20 (12)</td>
<td>2</td>
</tr>
<tr>
<td><strong>Apache</strong></td>
<td>5 (5)</td>
<td>3 (3)</td>
<td>2</td>
</tr>
<tr>
<td><strong>Wine</strong></td>
<td>31 (19)</td>
<td>30 (18)</td>
<td>1</td>
</tr>
<tr>
<td><strong>PHP</strong></td>
<td>16 (13)</td>
<td>13 (10)</td>
<td>3</td>
</tr>
<tr>
<td><strong>PostgreSQL</strong></td>
<td>8 (5)</td>
<td>7 (4)</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>484 (314)</td>
<td>371 (247)</td>
<td>113 (23%)</td>
</tr>
</tbody>
</table>

**Table**: Total number of Faults, False Positives (FP).
### Higher Potential Impact of Detected Faults

<table>
<thead>
<tr>
<th>Category</th>
<th>Leaks</th>
<th>Transient Errors</th>
<th>No device and address</th>
<th>Invalid user value</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read/write</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leak</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Lock</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Debug</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Ioctl</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leak</td>
<td>12</td>
<td>3</td>
<td>16</td>
<td>5</td>
<td>36</td>
</tr>
<tr>
<td>Lock</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Debug</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leak</td>
<td>64</td>
<td>14</td>
<td>95</td>
<td>8</td>
<td>181</td>
</tr>
<tr>
<td>Lock</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Debug</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td><strong>Static init</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leak</td>
<td>12</td>
<td>2</td>
<td>14</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>Lock</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Debug</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leak</td>
<td>90</td>
<td>21</td>
<td>131</td>
<td>15</td>
<td>257</td>
</tr>
<tr>
<td>Lock</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Debug</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>6</td>
<td>19</td>
</tr>
</tbody>
</table>
### Reasons of False Positives

<table>
<thead>
<tr>
<th></th>
<th>FP</th>
<th>Heuristics Fail</th>
<th>Fail to recognize releasing operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Not EHC</td>
<td>Not Alloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Via Alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-local Call frees</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Caller frees</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other</td>
</tr>
<tr>
<td>Linux drivers</td>
<td>56</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Linux sound</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Linux net</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Linux fs</td>
<td>25</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Python (2.7)</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Python (3.2.3)</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Apache</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Wine</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>PHP</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>113</td>
<td>4 (4%)</td>
<td>29 (26%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>33 (29%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14 (12%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 (13%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>18 (16%)</td>
</tr>
</tbody>
</table>
Our Strategy VS Data-Mining Strategy

- Detected 371 faults associated with 150 protocols
- Threshold values are taken from PR-Miner [Li FSE:05]
- Only 7 protocols are valid using the threshold values
- So, only 23 (6%) faults can be identified
Scalability

Analyzing time (in seconds) per line of code
Summary

- HECtor is an accurate and scalable approach to finding resource release omission faults in error-handling code.
- It has found 371 faults with the false positives rate of 23%.
- Some faults allow unprivileged malicious user to crash the entire system.
- 97 patches submitted for Linux drivers.
  - 74 are accepted
  - 23 are not accepted yet
Future work, Publications, and Conclusion
Future Work

- Relax the need for exemplars
- Find other memory related bugs
- Find shared variables
- Fix bugs
Related Publications

• Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calves, Julia Lawall, and Gilles Muller “Faults in Linux: Ten Years Later” in the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011, CA, USA.


Conclusion

- The goal of the work is to improve the quality of the error-handling code in systems software written in C language.
- The work used local information that is found within the same function.
- The first contribution is an empirical study on error-handling code.
- The second contribution helps to reduce making mistakes in the error-handling code.
- The third contribution helps to find existing faults in the error-handling code.