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

INVENTEURS DU MONDE NUMÉRIQUE
SORBONNE UNIVERSITÉS

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.

```
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

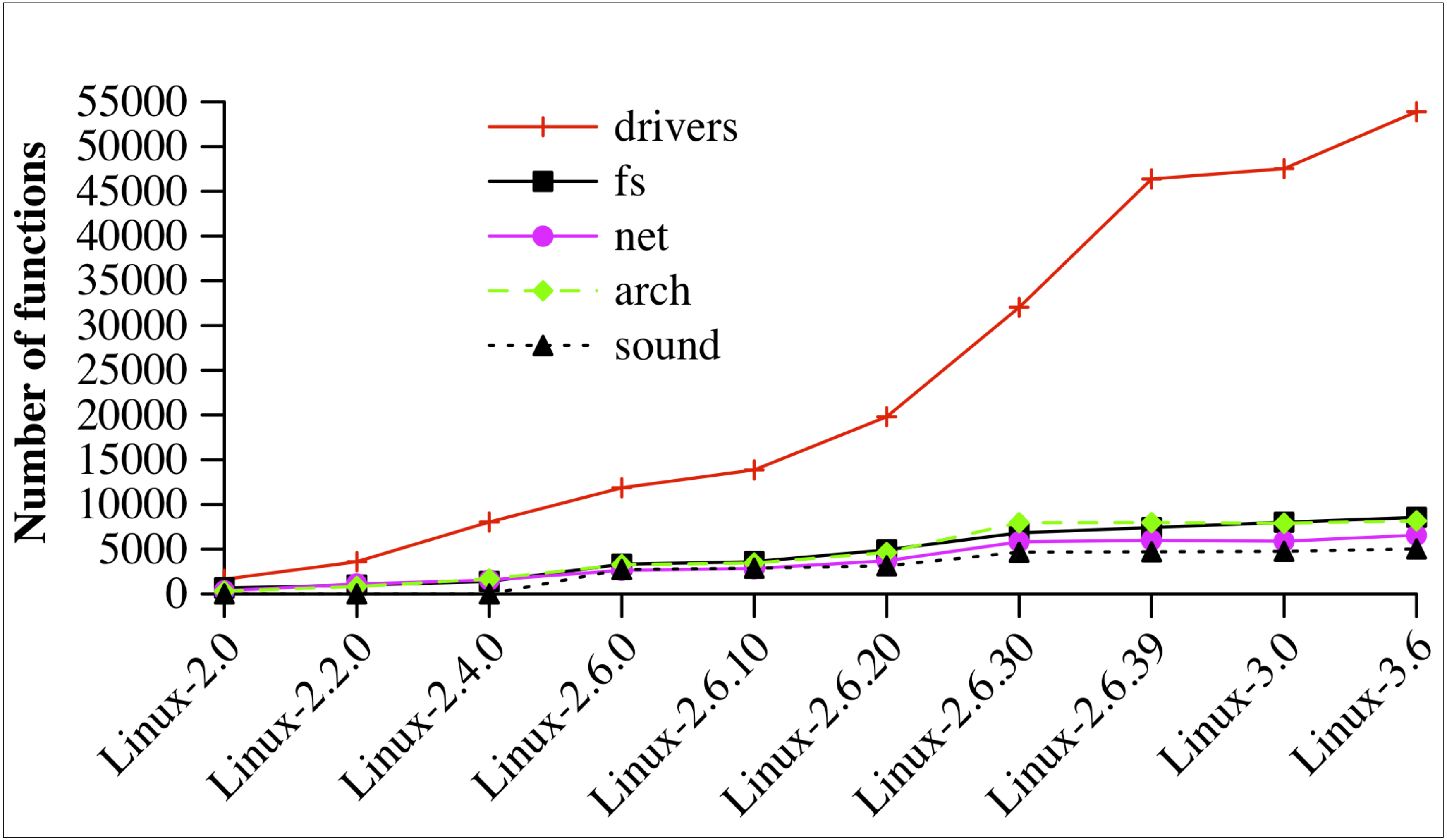
Research Questions:

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

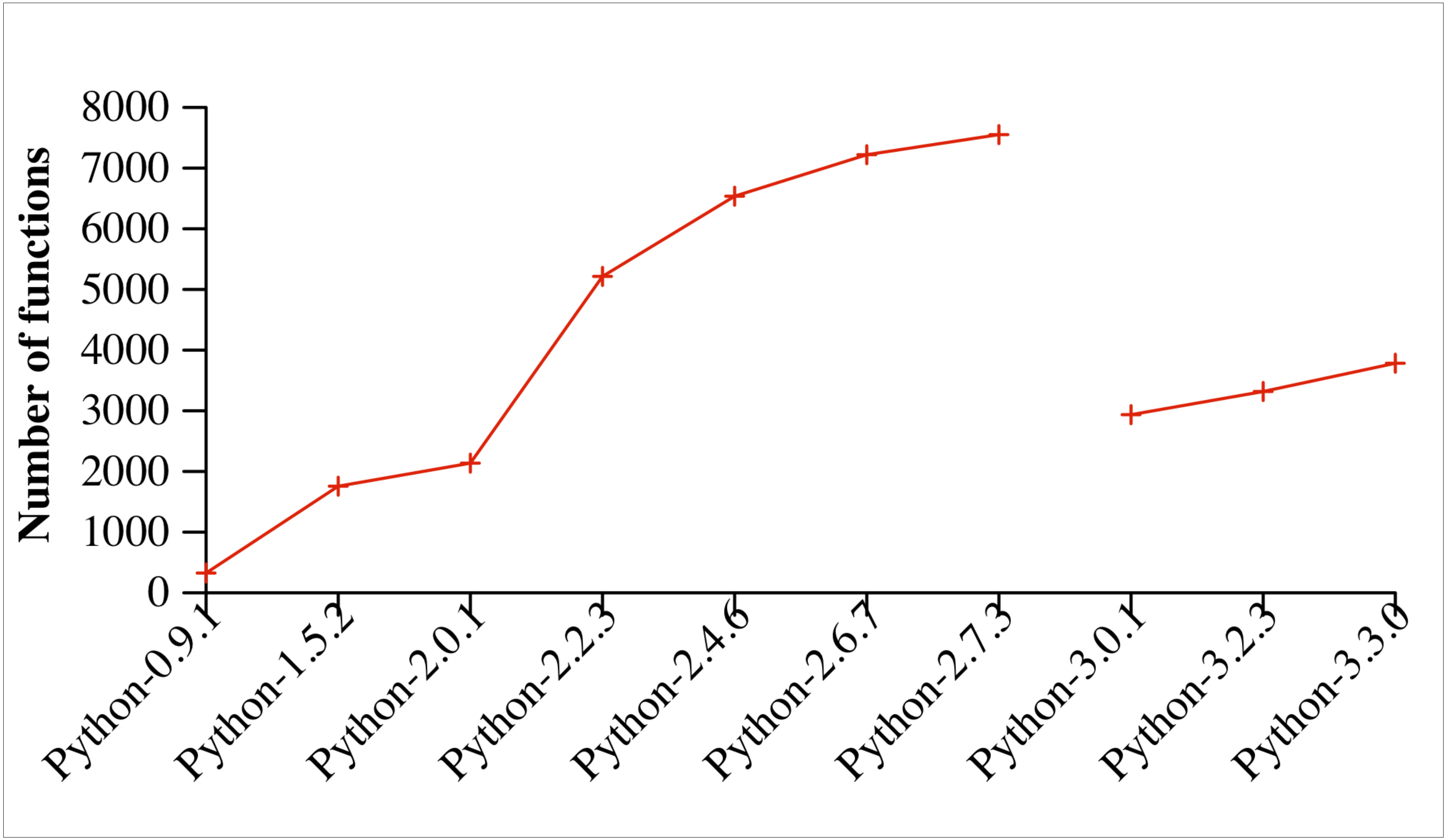
Considered Systems Software

SL	Project	Lines of code	Version	Description
1	Linux *drivers*	4.6 MLoC	2.6.34	Linux device drivers
2	Linux *sound*	0.4 MLoC	2.6.34	Linux sound drivers
3	Linux *net*	0.4 MLoC	2.6.34	Linux networking
4	Linux *fs*	0.7 MLoC	2.6.34	Linux file systems
5	Wine	2.1 MLoC	1.5.0	Windows emulator
6	PostgreSQL	0.6 MLoC	9.1.3	Database
7	Apache httpd	0.1 MLoC	2.4.1	HTTP server
8	Python	0.4 MLoC	2.7.3	Python runtime
9	Python	0.3 MLoC	3.2.3	Python runtime
10	PHP	0.6 MLoC	5.4.0	PHP runtime

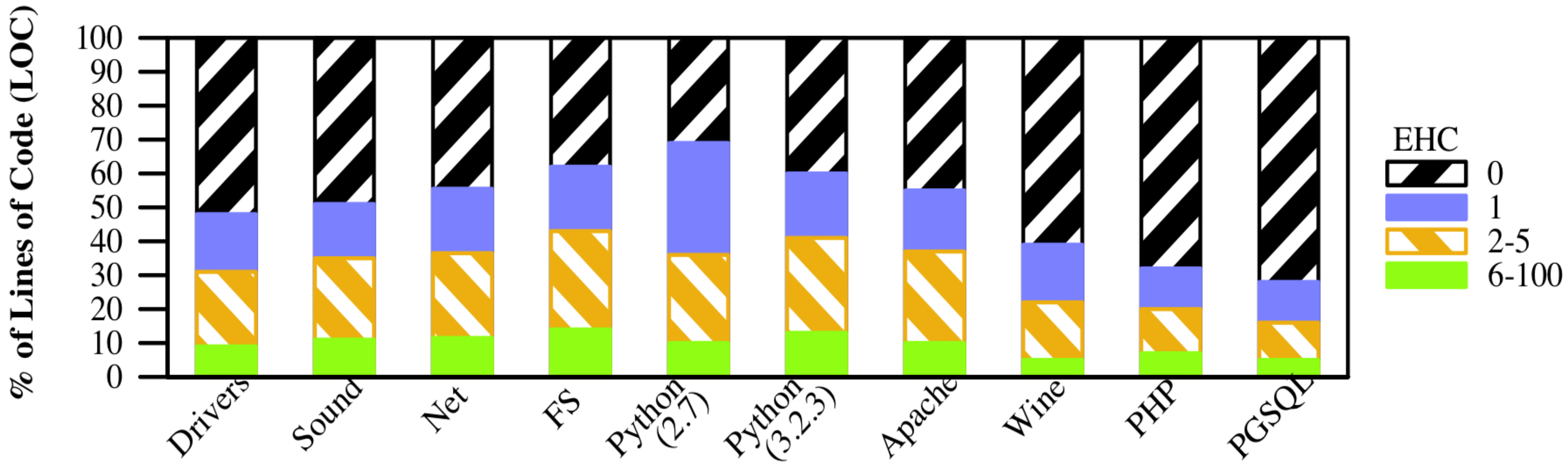
Error-Handling in Linux



Error-Handling in Python



Error-Handling in Systems Code



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

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

Basic Strategy: Problems

Basic strategy

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

Problems

- Duplicates code

Basic Strategy: Problems

Basic strategy

```
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

Basic Strategy: Problems

Basic strategy

```
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;
}
```

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

```
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

```
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

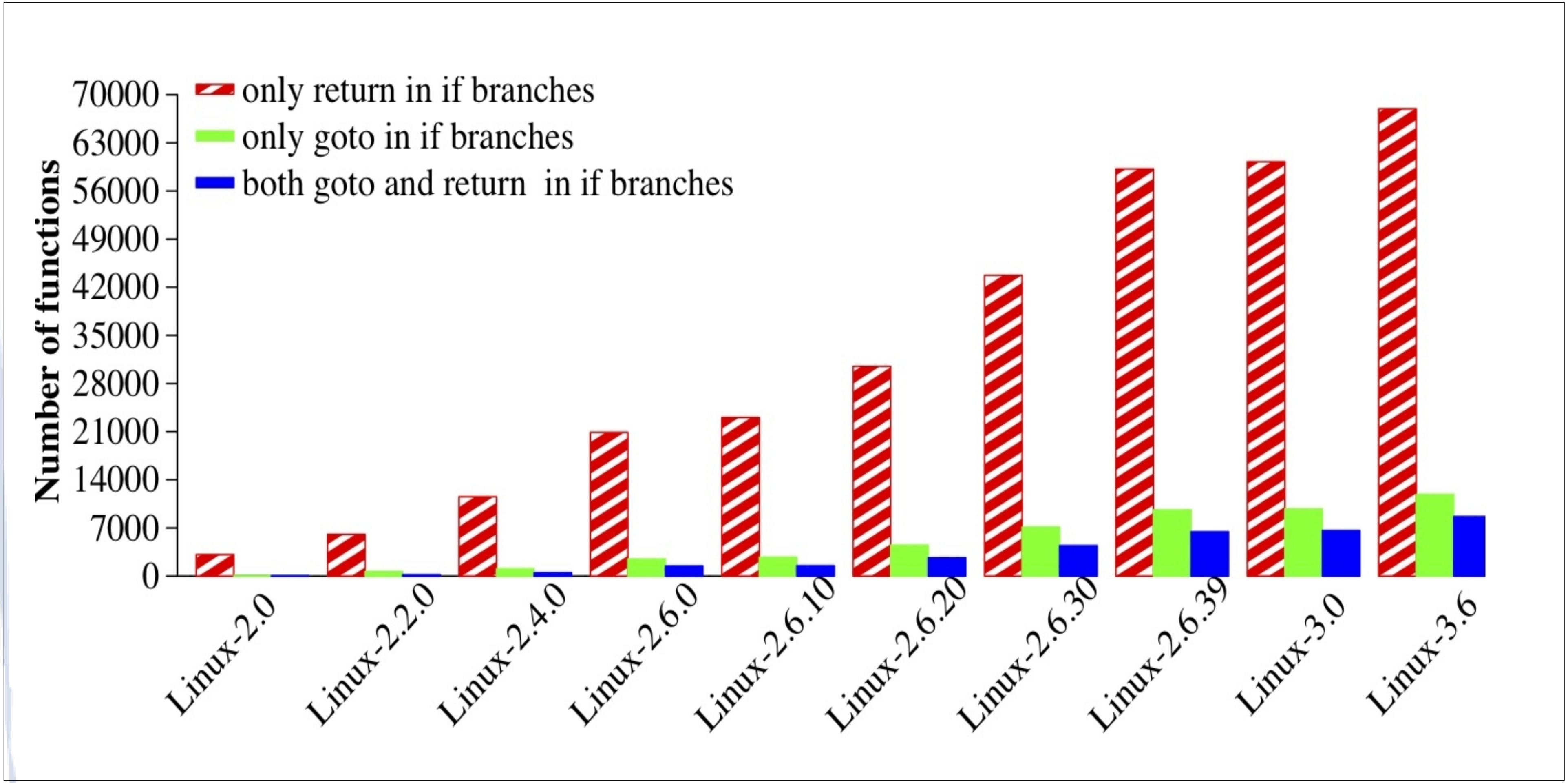
Goto-based Strategy: Benefits

Goto-based strategy

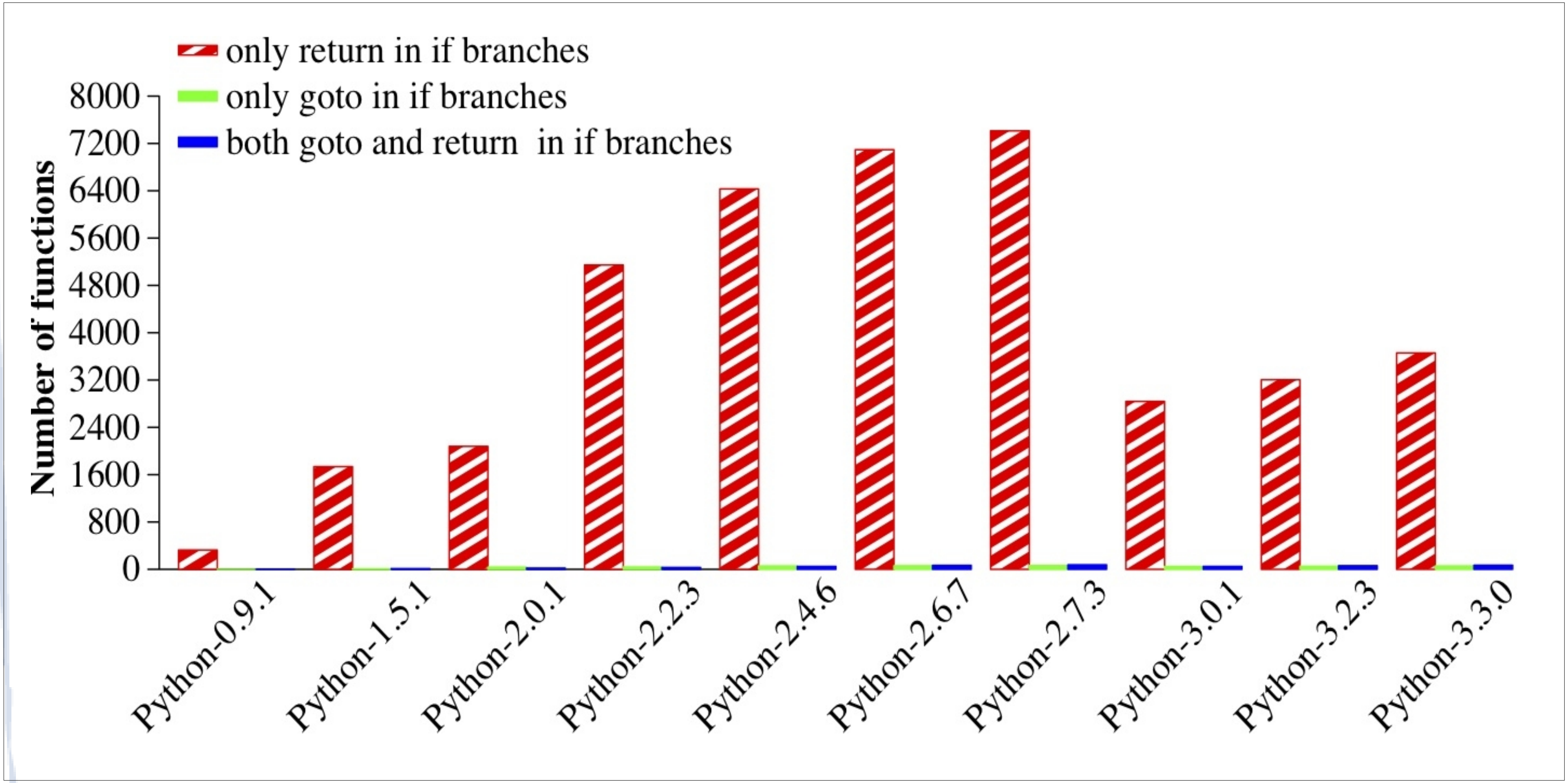
```
x = alloc();
…
n = alloc();
...
if(!y)
   goto out;
…
if(!m)
   goto out;
if(!z)
   goto out;
…
out:
   free(n);
   free(x);
   return -ENOMEM;
```

- 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



Basic vs Goto-based strategies in Python



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

[LCTES11]

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:

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

```
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)

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

1

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

out:
      f(a);
      return ret;
```

2

Reasons for no sharing

1. No state restoring operations

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

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

out:
      return ret;
```

Reasons for no sharing

2. Only one branch to transform

```
f() {
    ...
    x = allocate();
    …
    y = noallocate();
    if(y) {
       ...
       free(x);
       return ret;
    }
}
```

Reasons for no sharing

3. Nothing in common with other error handling code

```
...
if(y) {
  free(x);
  return ret;
}
...
if(z) {
  free(x);
  return ret;
}
...
free(x);
...
if(l) {
  action(z);
  return ret;
}
```

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

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

```
...
  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

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

→

```
...
  if (!sl->data)
    goto out1;

…
out:
    clear_bit(n,sbi-symlink_bitmap);
out1:
    unlock_kernel();
    return ret;
```

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

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

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

```
...
  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;
```

→

```
...
   if (!ent)
     goto out1;
…
return 0;
out1:
     kfree(sl->data);
out:
     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;
```

Challenge

- Identify the needed code

Omission Fault

Autofs4 code containing an 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

```
...
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 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

```
...
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

Function list

```
...
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

Function list

kfree(x);
ff();

```
...
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

Function list

kfree(x);
ff();

```
...
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

Omitted kfree(x)

```
...
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

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

```
...
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.

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

Case 3: Alternate Ways to Release

Scenario 1

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

Scenario 2

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

Scenario 3

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

Scenario 4

```
...
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 -ENOTTY;
}
…
out:
  free_dev_ioctl (param);
  return err;
```

Exemplar

Candidate

Autofs4 code containing a fault

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

Results

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

Table: Total number of Faults, False Positives (FP).

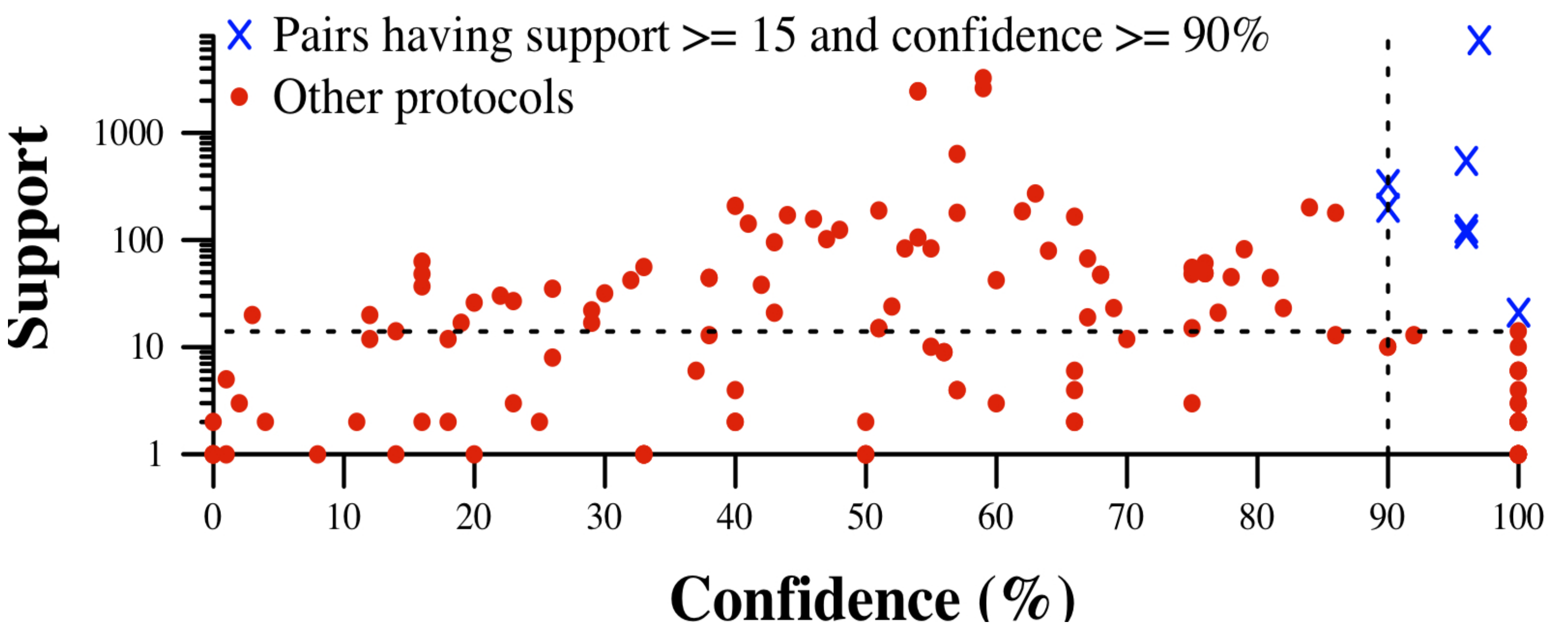
Higher Potential Impact of Detected Faults

		Lack of memory	Transient errors	No device and address	Invalid user value	Total
Read/write	Leak	2	2	6	0	**10**
	Lock	0	0	0	0	**0**
	Debug	0	0	0	2	**2**
Ioctl	Leak	12	3	16	5	**36**
	Lock	0	0	0	1	**1**
	Debug	0	0	1	2	**3**
Others	Leak	64	14	95	8	**181**
	Lock	1	1	5	0	**7**
	Debug	1	1	10	2	**14**
Static init	Leak	12	2	14	2	**30**
	Lock	0	0	0	1	**1**
	Debug	0	0	0	0	**0**
Total	Leak	**90**	**21**	**131**	**15**	**257**
	Lock	**1**	**1**	**5**	**2**	**9**
	Debug	**1**	**1**	**11**	**6**	**19**

Reasons of False Positives

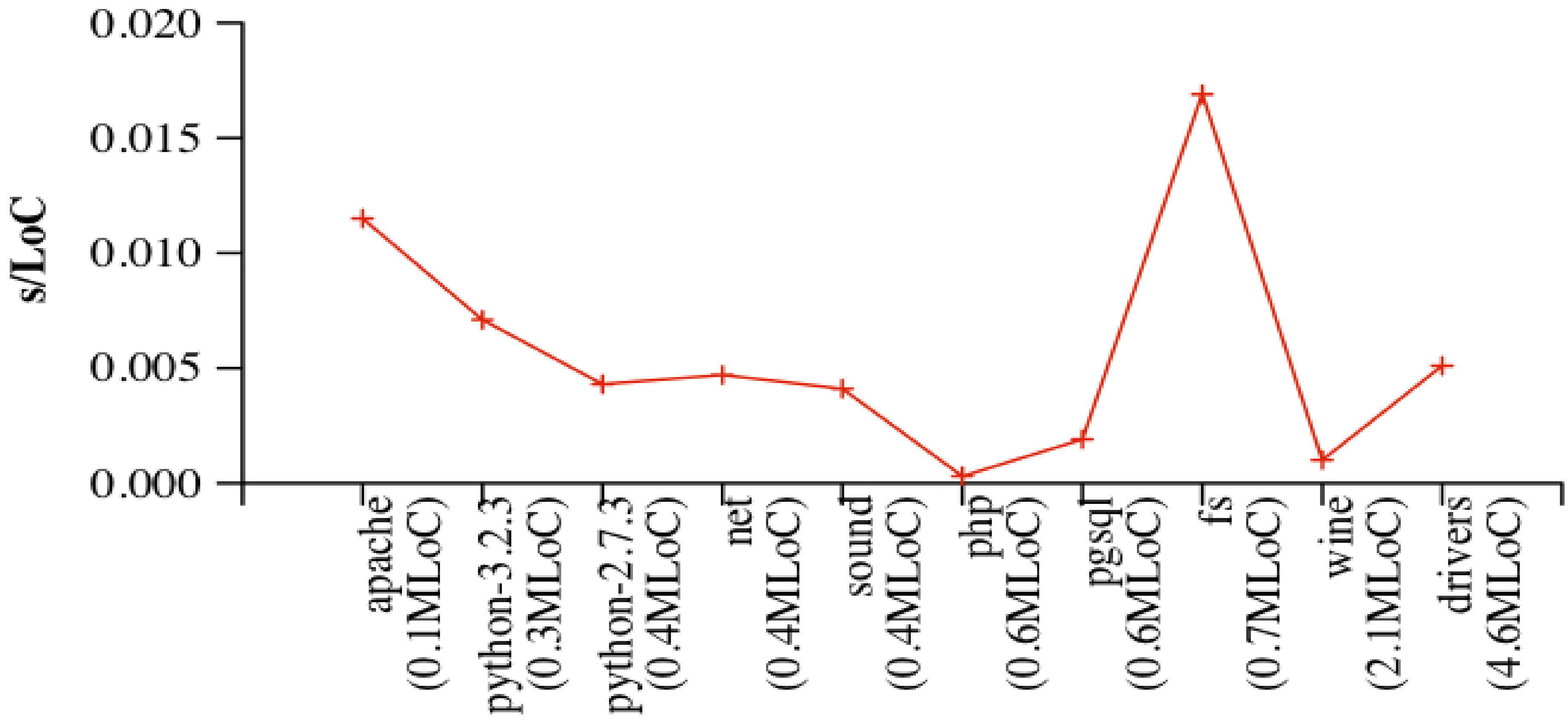
	FP	Heuristics Fail		Fail to recognize releasing operations			
		Not EHC	Not Alloc	Via Alias	Non-local Call frees	Caller frees	Other
Linux *drivers*	56	3	16	11	13	8	5
Linux *sound*	13	0	0	13	0	0	0
Linux *net*	6	0	0	0	0	1	5
Linux *fs*	25	0	7	6	1	6	5
Python (2.7)	4	0	0	3	0	0	1
Python (3.2.3)	2	0	1	0	0	0	1
Apache	2	1	0	0	0	0	1
Wine	1	0	1	0	0	0	0
PHP	3	0	3	0	0	0	0
PostgreSQL	1	0	1	0	0	0	0
Total	113	4 (4%)	29 (26%)	33 (29%)	14 (12%)	15 (13%)	18 (16%)

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 *16^{th} International Conference on Architectural Support for Programming Languages and Operating Systems* (**ASPLOS**)*,* 2011, CA, USA.
- **Suman Saha**, Julia Lawall, and Gilles Muller "An Approach to Improving the Structure of Error-Handling Code in the Linux Kernel" in the *ACM SIGPLAN/SIGBED Conference on Language, Compilers, Tools and Theory for Embedded Systems* (**LCTES**)*,* 2011, Chicago, USA.
- **Suman Saha,** Julia Lawall, and Gilles Muller "Finding Resource-Release Omission Faults in Linux" *in the 6^{th} Workshop on Programming Languages and Operating Systems (**PLOS**), Portugal, October 2011.* **Also appeared** in *SIGOPS Operating System Review (**OSR**),* vol. 45, pp. 5-9 (2011).
- **Suman Saha,** Jean-Pierre Lozi, Gaèl Thomas, Julia Lawall, and Gilles Muller "Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software" *in the 43^{rd} Annual IEEE/IFIP International Conference on Dependable Systems and Networks (**DSN**), Budapest, June 2013.*

Conclusion

- The goal of the work is to improve the quality of the error-handling code in systems software written C language
- The work used local information that is found within the same function
- The first is an empirical studies 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