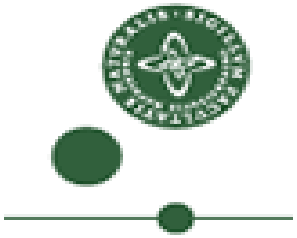
Finding Resource-Release Omission Faults in Linux

Accepted at PLOS 2011

Suman Saha[1], Julia Lawall[2], and Gilles Muller[13]
LIP6-Regal[1]/DIKU[2]/INRIA[3]



Linux OS

- Linux is used in Mobile phones, desktop computers, supercomputers and many more electronic devices.

Linux OS

- Linux is used in Mobile phones, desktop computers, supercomputers and many more electronic devices.
- Linux is a leading server OS, and runs the 10 fastest supercomputers in the world.

Linux OS

- Linux is used in Mobile phones, desktop computers, supercomputers and many more electronic devices.
- Linux is a leading server OS, and runs the 10 fastest supercomputers in the world.
- During third quarter of 2010, 25.5% of smartphones sold, used Android (modified version of Linux kernel).

Linux OS

- Linux is used in Mobile phones, desktop computers, supercomputers and many more electronic devices.
- Linux is a leading server OS, and runs the 10 fastest supercomputers in the world.
- During third quarter of 2010, 25.5% of smartphones sold, used Android (modified version of Linux kernel).
- Reliability of code used in Linux is critical.
  - Handling transient run-time errors is essential

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;
    …
}
```

arch/s390/kernel/ipl.c

- Mistakes cause deadlock and memory leaks
- Key to ensuring reliability

Issues

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

Issues

- Faults in error-handling code cause deadlocks and memory leaks
- The error-handling code is not tested often
  - Research has shown there are many faults in error-handling code

Issues

- Faults in error-handling code cause deadlocks and memory leaks
- The error-handling code is not tested often
  - Research has shown there are many faults in error-handling code
- Fixing these faults requires knowing what kind of error-handling code is required

One approach:Data-Mining based strategy

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

One approach:Data-Mining based strategy

- Data-mining is used to find protocols in source code.
  - For example, *kmalloc* and *kfree* often occur together
- Evaluate a potential protocol using two threshold values:
  - ***Min Support***: The minimum number of occurrences of a protocol
  - ***Confidence***: The number of occurrences vs the potential number of occurrences

One approach:Data-Mining based strategy

- Data-mining is used to find protocols in source code.
  - For example, *kmalloc* and *kfree* often occur together
- Evaluate a potential protocol using two threshold values:
  - ***Min Support***: The minimum number of occurrences of a protocol
  - ***Confidence***: The number of occurrences vs the potential number of occurrences
- Use statistics-based analysis to find probable protocols

One approach:Data-Mining based strategy

- Data-mining is used to find protocols in source code.
  - For example, *kmalloc* and *kfree* often occur together
- Evaluate a potential protocol using two threshold values:
  - ***Min Support***: The minimum number of occurrences of a protocol
  - ***Confidence***: The number of occurrences vs the potential number of occurrences
- Use statistics-based analysis to find probable protocols
- The identified protocols are used to find faults in source-code

Protocols with lower support or confidence

- The 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

Protocols with lower support or confidence

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

```
...
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

Our Work

- **Goal**: Detect resource-release omission faults in error-handling code
- **Approach**: Use information about error-handling code found within the same function

Our Work

- **Goal**: Detect resource-release omission faults in error-handling code
- **Approach**: Use information about error-handling code found within the same function
- The goal is to be different than data mining, not necessarily to be complete

Our Work

- **Goal**: Detect resource-release omission faults in error-handling code
- **Approach**: Use information about error-handling code found within the same function
- The goal is to be different than data mining, not necessarily to be complete
- We may have false negatives, if there is no model of the correct error handling code in the same function

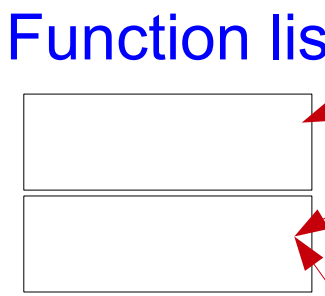
Detecting Resource-Release Omission Faults

1. Identify error-handling code

```
...
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




```
...
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();

```
...
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();

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

Omitted kfree(x)

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


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

1. Identify Error-Handling Code

General Idea

- Conditional branch ending with *return* or *goto*

1. Identify Error-Handling Code

General Idea

- Conditional branch ending with *return* or *goto*

- Specific return error values
  - NULL, negative constant, error pointer

1. Identify Error-Handling Code

General Idea

- Conditional branch ending with *return* or *goto*

- Specific return error values
  - NULL, negative constant, error pointer
- Return of an Identifier
  - Find reaching definition of the identifier
  - Analyze test case of the conditional branch

```
...
long r;
…
r = copy_from_user(...);
if(r) {
        kfree(newmem);
        return r;
}
...
```

2. Collect Resource-Release Operations

Operations to select

- At most one pointer-typed argument

2. Collect Resource-Release Operations

Operations to select

- At most one pointer-typed argument
- No debugging code

2. Collect Resource-Release Operations

Operations to select

- At most one pointer-typed argument
- No debugging code
- The final operation that access the argument

2. Collect Resource-Release Operations

Operations to select

- At most one pointer-typed argument
- No debugging code
- The final operation that access the argument
- Zero argument operation is automatically selected

2. Collect Resource-Release Operations

Operations to select

- At most one pointer-typed argument
- No debugging code
- The final operation that access the argument
- Zero argument operation is automatically selected

Keep in the **Function List**

3. Compare Function List with each Block

Function List

kfree(x)

ff()

Error-handling code

ff()

Candidate set: Omitted Operations

kfree(x)

Candidate set =

Function list – set of resource-releasing operation in the block

4. 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

4. 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.

4. 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

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

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

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

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

4. Analyze Omitted Releasing Operations

Four alternate ways

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

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

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

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

4. Analyze Omitted Releasing Operations

- Use a def-use chain to keep information about the variable x

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

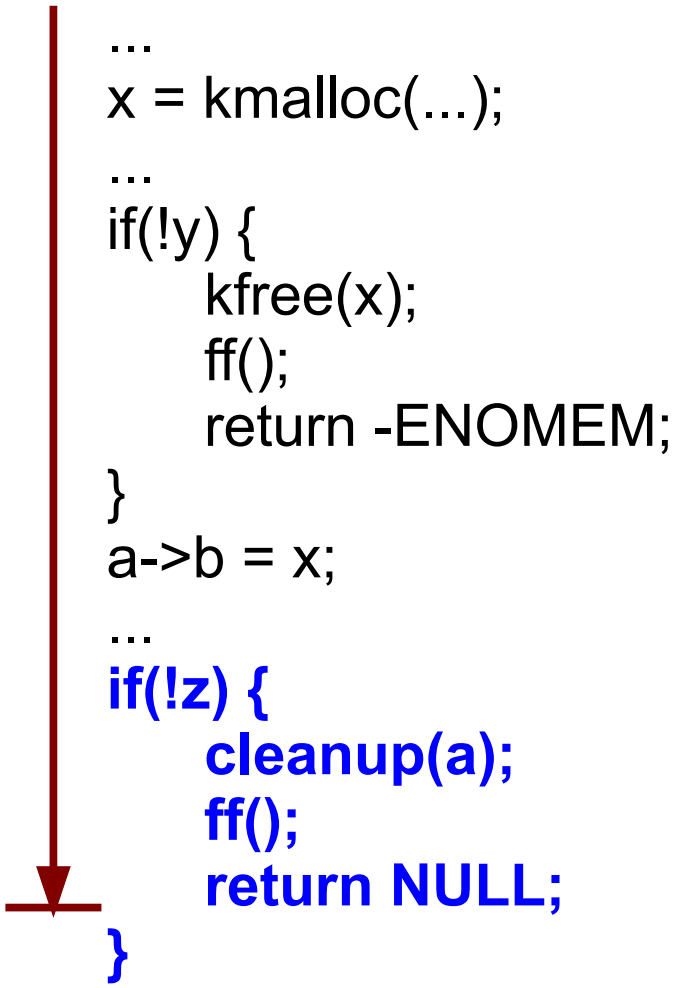
4. Analyze Omitted Releasing Operations

- Use a def-use chain to keep information about the variable x
- ***Extended*** def-use Chain (EDU-Chain): also keeps other information related to the variable.

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

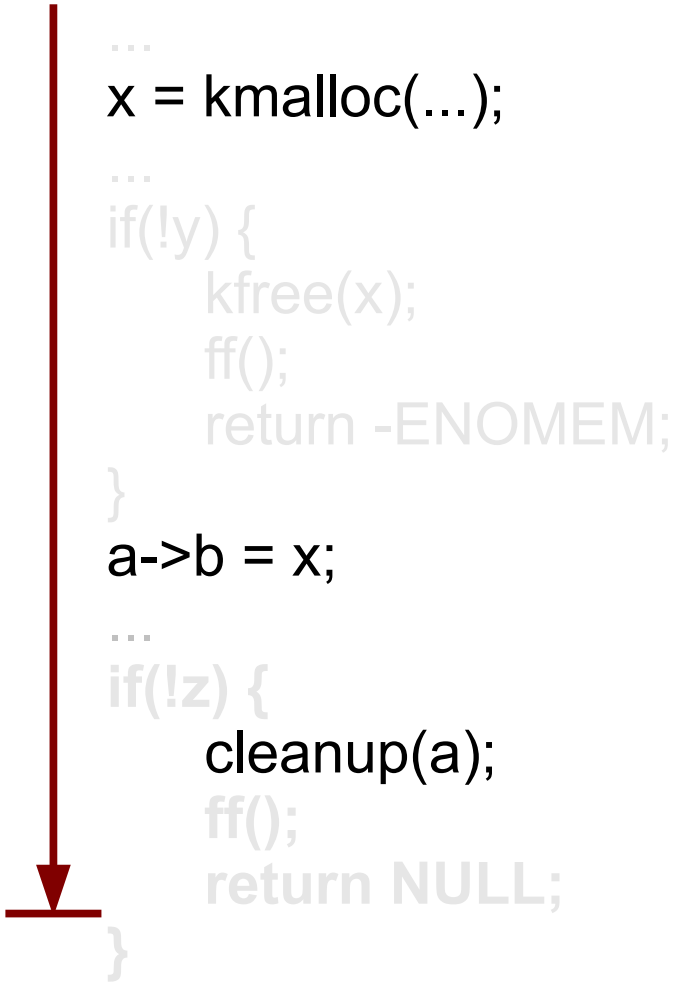
4. Analyze Omitted Releasing Operations

- Build EDU-Chain$_1$ from x to the return statement of the block that omits the operation



4. Analyze Omitted Releasing Operations

- Build EDU-Chain$_1$ from x to the return statement of the block that omits the operation



4. Analyze Omitted Releasing Operations

- Build EDU-Chain$_1$ from x to the return statement of the block that omits the operation
- Build EDU-Chain$_2$ from x to the nearest block where the omitted operation appears

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

4. Analyze Omitted Releasing Operations

- Build EDU-Chain$_1$ from x to the return statement of the block that omits the operation
- Build EDU-Chain$_2$ from x to the nearest block where the omitted operation appears

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

4. Analyze Omitted Releasing Operations

Apply Heuristics

- Do both EDU-Chains provide the same allocation points for x?

EDU-Chain$_1$

x = kmalloc(...)

a->b = x

cleanup(a)

EDU-Chain$_2$

x = kmalloc(...)

4. Analyze Omitted Releasing Operations

Apply Heuristics

- Do the both EDU-Chains provide the same allocation points for x?
- Are the reaching definitions of x same at the end of both EDU-Chains?

4. Analyze Omitted Releasing Operations

Apply Heuristics

- Do the both EDU-Chains provide the same allocation points for x?
- Are the reaching definitions of x same at the end of both EDU-Chains?
- Is there any return statement in the EDU-Chain$_1$ that returns x?

EDU-Chain$_1$

x = kmalloc(...)

a->b = x

cleanup(a)

EDU-Chain$_2$

x = kmalloc(...)

4. Analyze Omitted Releasing Operations

Apply Heuristics

- Is the resource released through any other pointer in EDU-Chain$_1$?

EDU-Chain$_1$

x = kmalloc(...)

a->b = x

cleanup(a)

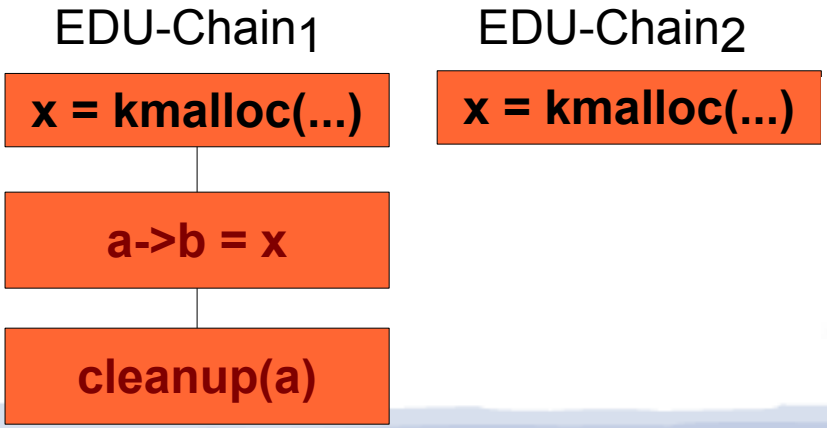
EDU-Chain$_2$

x = kmalloc(...)

4. Analyze Omitted Releasing Operations

Apply Heuristics

- Is the resource released through any other pointer in EDU-Chain$_1$?



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

Summary of Algorithm

The algorithm reports an omission fault if

- The reaching definition of the associated variable provides same allocation definition at the error-handling code.

  AND

- The error-handling code does not return the resource

  AND

- No alternate releasing operation releases the resource

Results

	Total reports	Faults	FP	TODO
Faults	126	**103**	20	3
Function	78	65	10	3

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

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.

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 can be released by two different operations and **4.2%** by three.

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 can be released by two different operations and **4.2%** by three.
- **5.2%** of them can be released via another pointer.

The Benefit of Context Sensitivity

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

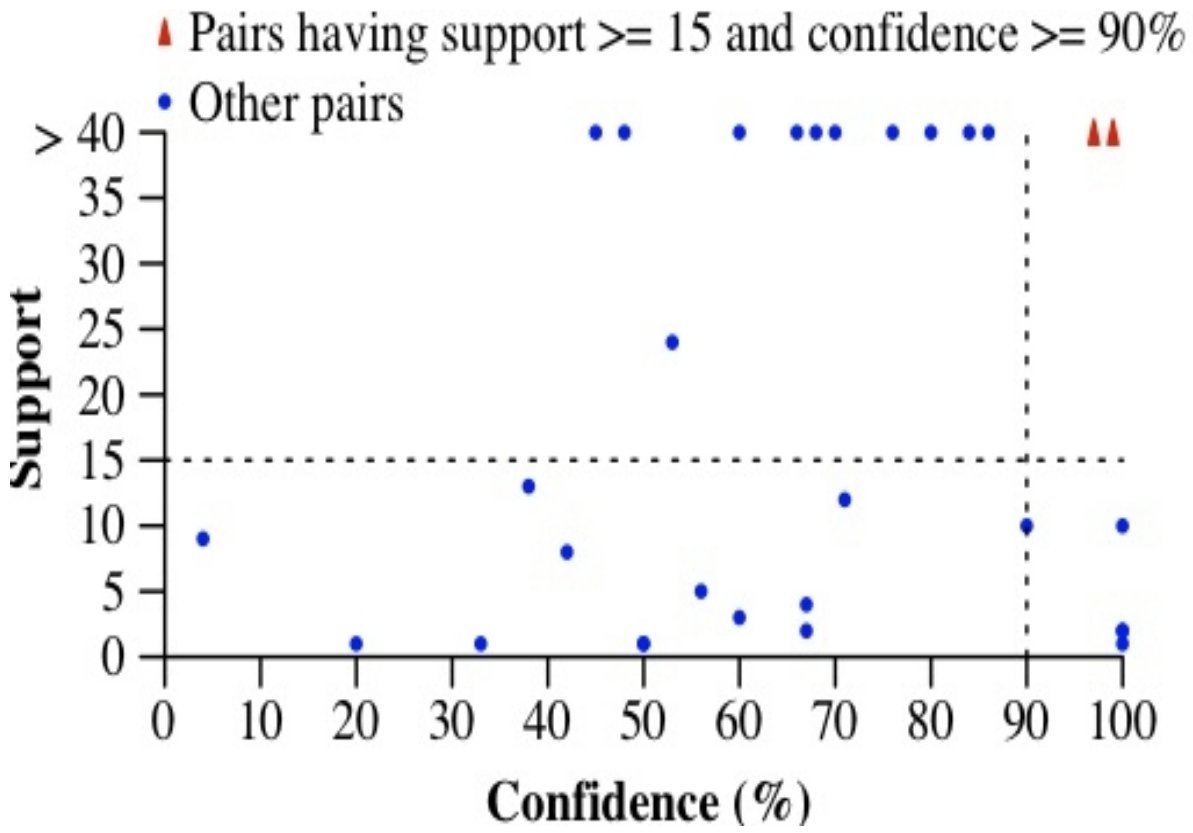
- **22.4%** of them can be released by two different operations and **4.2%** by three.
- **5.2%** of them can be released via another pointer.
- **15.7%** of them can be released by intervening functions

The Benefit of Context Sensitivity

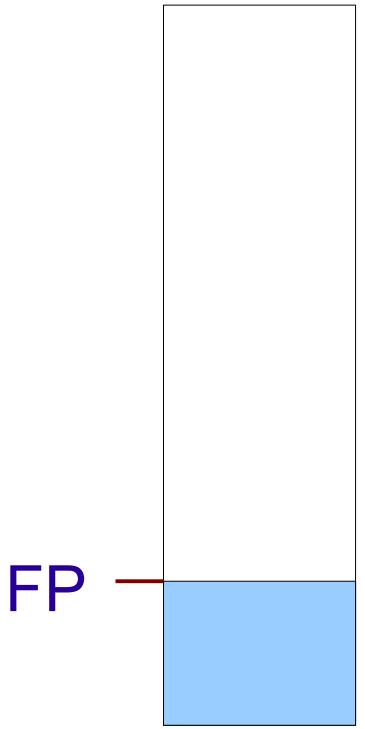
The tool found **331** resource allocations for which at least one releasing operation **seems** to be omitted.

- **22.4%** of them can be released by two different operations and **4.2%** by three.
- **5.2%** of them can be released via another pointer.
- **15.7%** of them can be released by intervening functions
- **14.8%** of them can be released by a call to another function that is defined in the same file.

Comparison with Data-Mining Strategy

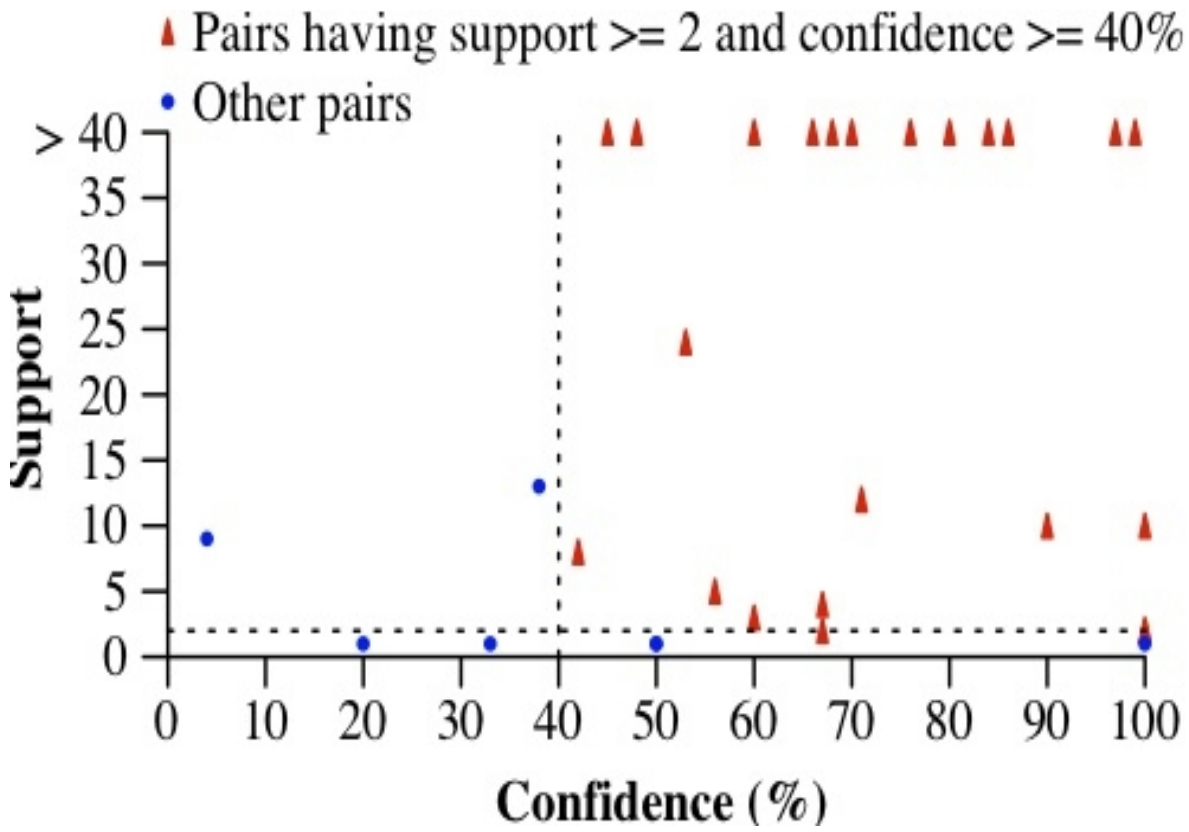


More free in threshold values more FP

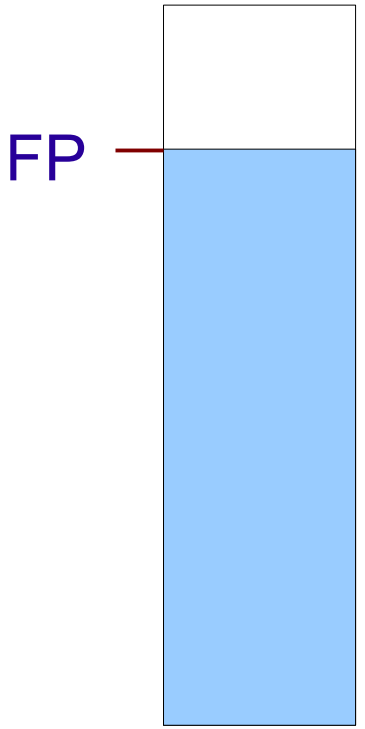


103 faults are associated with **30** protocols

Comparison with Data-Mining Strategy



More free in thresholds values more FP



103 faults are associated with **30** protocols

Conclusion

- We have focused on context-sensitivity constraints on the choice of resource-releasing operations and used this as a guideline for finding faults.

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.

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.

Future work

This approach only detects the omission of resource-releasing operations, but does not fix these faults. In future work, we will consider this issue.