

# Introduction to Coccinelle and its Usage in the Linux Kernel

---

Julia Lawall (Inria/LIP6)

May 24, 2018

# What is the Linux kernel?

An open-source operating system, known for:

- Reliability:

```
13:27:36 up 187 days,  1:15,  4 users,  load average: ...
```

- Flexibility

- 86% of smartphones run Android (2017)
- 92% of Amazon EC2 instances run Linux (2016)
- 100% of the top 500 supercomputers run Linux (2017)

- Low cost per unit

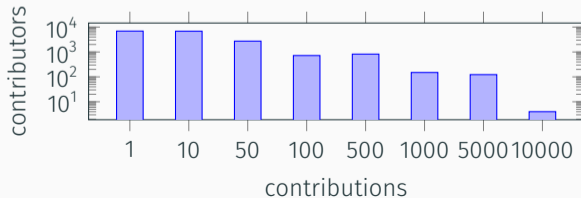
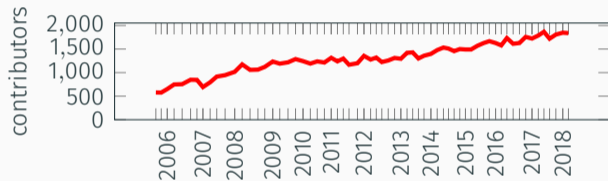
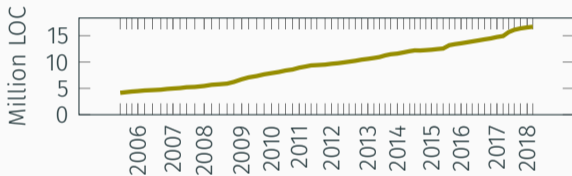


# Some history

First release in 1991.

- v1.0 in 1994: 121 KLOC, v2.0 in 1996: 500 KLOC

Recent evolution:



# Challenges

## Critical code:

- Requires both correctness and performance.

## Large code base.

## Large, diverse developer base.

## Need for automation and scalability:

- How to impose API improvements on the entire kernel?
- How to ensure that a bug found in one place is fixed everywhere?

## Example

Evolution: A new function: kcalloc

⇒ Collateral evolution: Merge kcalloc and memset into kcalloc

```
fh = kcalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (!fh) {
    dprintk(1,
            KERN_ERR
            "%s: zoran_open(): allocation of zoran_fh failed\n",
            ZR_DEVNAME(zr));
    return -ENOMEM;
}
memset(fh, 0, sizeof(struct zoran_fh));
```

## Example

Evolution: A new function: kcalloc

⇒ Collateral evolution: Merge kmalloc and memset into kcalloc

```
fh = kcalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (!fh) {
    dprintk(1,
           KERN_ERR
           "%s: zoran_open(): allocation of zoran_fh failed\n",
           ZR_DEVNAME(zr));
    return -ENOMEM;
}
```

## Example

Evolution: A new function: kcalloc

⇒ Collateral evolution: Merge kmalloc and memset into kcalloc

```
fh = kcalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (!fh) {
    dprintk(1,
           KERN_ERR
           "%s: zoran_open(): allocation of zoran_fh failed\n",
           ZR_DEVNAME(zr));
    return -ENOMEM;
}
```

Originally, hundreds of kmalloc and memset calls

## Example

Bug: Reference count mismanagement

- `for_each` iterator increments the reference count of the current element and decrements the reference count of the previous one.
- `break;` escapes, skipping the decrement.
- $\implies$  A memory leak.

```
/* Initialise all packet dmas */
for_each_child_of_node(node, child) {
    ret = dma_init(node, child);
    if (ret) {
        dev_err(&pdev->dev, "init failed with %d\n", ret);
        break;
    }
}
```

6 instances in linux-next (May 4, 2018)



Coccinelle to the rescue!

# What is Coccinelle?

- Pattern-based language for matching and transforming C code
- Under development since 2005. Open source since 2008.
- Allows code changes to be expressed using patch-like code patterns (semantic patches).

# Semantic patches

- Like patches, but independent of irrelevant details (line numbers, spacing, variable names, etc.)
- Derived from code, with abstraction.
- **Goal:** fit with the existing habits of the Linux programmer.

## Semantic patch example

```
@@
expression x,E1,E2;
@@
- x = kmalloc(E1,E2);
+ x = kzalloc(E1,E2);
  ...
- memset(x, 0, E1);
```

## Creating a semantic patch: kmalloc → kcalloc

Start with an example

```
fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (!fh) {
    dprintk(1,
        KERN_ERR
        "%s: zoran_open(): allocation of zoran_fh failed\n",
        ZR_DEVNAME(zr));
    return -ENOMEM;
}
memset(fh, 0, sizeof(struct zoran_fh));
```

## Creating a semantic patch: kmalloc → kcalloc

### Eliminate irrelevant code

```
fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
```

```
...
```

```
memset(fh, 0, sizeof(struct zoran_fh));
```

## Creating a semantic patch: kmalloc → kcalloc

### Describe transformations

```
- fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);  
+ fh = kcalloc(sizeof(struct zoran_fh), GFP_KERNEL);  
...  
- memset(fh, 0, sizeof(struct zoran_fh));
```

## Creating a semantic patch: kmalloc → kcalloc

### Abstract over subterms

@@

```
expression x;  
expression E1,E2;
```

@@

```
- x = kmalloc(E1,E2);  
+ x = kcalloc(E1,E2);  
  ...  
  
- memset(x, 0, E1);
```



# Creating a semantic patch: kmalloc → kcalloc

## Refinement

@@

```
expression x;  
expression E1,E2,E3;  
identifier f;
```

@@

```
- x = kmalloc(E1,E2);  
+ x = kcalloc(E1,E2);  
  ... when != (<+...x...+>) = E3  
    when != f(...,x,...)  
- memset(x, 0, E1);
```

# Results

- Correctly updates 14 occurrences
  - 5 false positives, could be eliminated by more “**when**” tests

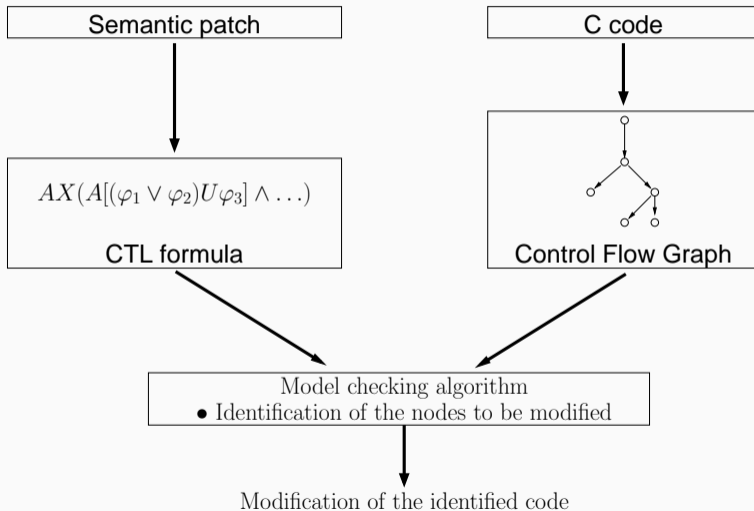
# Results

- Correctly updates 14 occurrences
  - 5 false positives, could be eliminated by more “when” tests
- Other opportunities:
  - `acpi_os_allocate` → `acpi_os_allocate_zeroed`
  - `dma_pool_alloc` → `dma_pool_zalloc`
  - `dma_alloc_coherent` → `dma_zalloc_coherent`
  - `kmem_cache_alloc` → `kmem_cache_zalloc`
  - `pci_alloc_consistent` → `pci_zalloc_consistent`
  - `vmalloc` → `vzalloc`
  - `vmalloc_node` → `vzalloc_node`

## Semantic patch example

```
@@
expression root,e;
local idexpression child;
iterator name for_each_child_of_node;
@@
    for_each_child_of_node(root, child) {
        ... when != of_node_put(child)
            when != e = child
+   of_node_put(child);
?   break;
        ...
    }
... when != child
```

## How does it work?



## Processing of C code

Goal: Support processing real Linux source code.

## Processing of C code

Goal: Support processing real Linux source code.

Dedicated C parser, keeping space and comment information.

## Processing of C code

Goal: Support processing real Linux source code.

Dedicated C parser, keeping space and comment information.

Limited, user-controlled inclusion of header files, to reduce runtime.



# Processing of C code

Goal: Support processing real Linux source code.

Dedicated C parser, keeping space and comment information.

Limited, user-controlled inclusion of header files, to reduce runtime.

No preprocessing.

- Code manipulated in terms of what the developer sees in the code base.
- Avoids the need for most header files.

# Processing of C code

Goal: Support processing real Linux source code.

Dedicated C parser, keeping space and comment information.

Limited, user-controlled inclusion of header files, to reduce runtime.

No preprocessing.

- Code manipulated in terms of what the developer sees in the code base.
- Avoids the need for most header files.

Intraprocedural CFG.

## Processing of Semantic patches

**Goal:** Allow specifying changes at all code levels.

- Concise and readable.

# Processing of Semantic patches

**Goal:** Allow specifying changes at all code levels.

- Concise and readable.

Support most of C, with few meta-level extensions

- ..., when, etc.

# Processing of Semantic patches

**Goal:** Allow specifying changes at all code levels.

- Concise and readable.

Support most of C, with few meta-level extensions

- ..., when, etc.

Isomorphisms, to reduce semantic patch size

- `X == NULL => !X`

# Processing of Semantic patches

**Goal:** Allow specifying changes at all code levels.

- Concise and readable.

Support most of C, with few meta-level extensions

- ..., when, etc.

Isomorphisms, to reduce semantic patch size

- `X == NULL => !X`

Implementation via translation to CTL

- Allows  $\forall$  and  $\exists$  quantification over paths.
- $\forall$  and  $\exists$  can be mixed in a single rule.

## First experiment (EuroSys 2008)

- Semantic patches for over 60 collateral evolutions.
- Applied to over 5800 Linux files from various versions, with a success rate of 100% on 93% of the files.

## First experiment (EuroSys 2008)

- Semantic patches for over 60 collateral evolutions.
- Applied to over 5800 Linux files from various versions, with a success rate of 100% on 93% of the files.
- Required a forgiving parser for all of C
- Required fully source-to-source transformation.



## First experiment (EuroSys 2008)

- Semantic patches for over 60 collateral evolutions.
- Applied to over 5800 Linux files from various versions, with a success rate of 100% on 93% of the files.
- Required a forgiving parser for all of C
- Required fully source-to-source transformation.

## Second experiment: the Linux kernel

- Parse errors - missing ; etc.
- kcalloc/memset: 136 files.
- 0 -> NULL for pointers, etc.

## First experiment (EuroSys 2008)

- Semantic patches for over 60 collateral evolutions.
- Applied to over 5800 Linux files from various versions, with a success rate of 100% on 93% of the files.
- Required a forgiving parser for all of C
- Required fully source-to-source transformation.

## Second experiment: the Linux kernel

- Parse errors - missing ; etc.
- kcalloc/memset: 136 files.
- 0 -> NULL for pointers, etc.
- Made releases, fixed bugs, filled in features as needed.

## First experiment (EuroSys 2008)

- Semantic patches for over 60 collateral evolutions.
- Applied to over 5800 Linux files from various versions, with a success rate of 100% on 93% of the files.
- Required a forgiving parser for all of C
- Required fully source-to-source transformation.

## Second experiment: the Linux kernel

- Parse errors - missing ; etc.
- kcalloc/memset: 136 files.
- 0 -> NULL for pointers, etc.
- Made releases, fixed bugs, filled in features as needed.

# Engagement with the Linux kernel community

Submission of over 2000 patches to the Linux kernel

Interaction with developers:

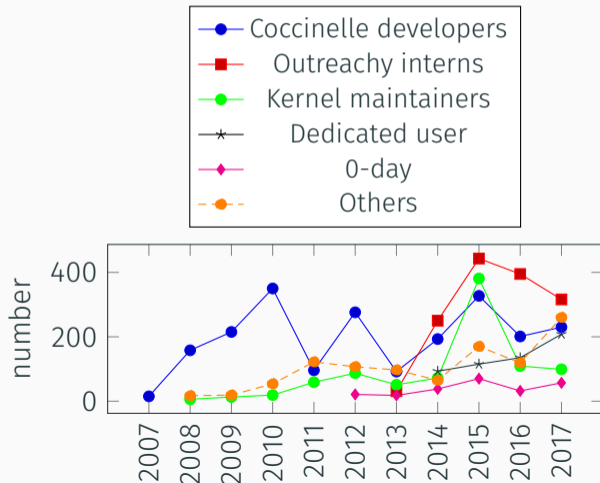
- Talks at the Kernel Summit, Linuxcon, FOSDEM, etc.
- Workshops for Linux developers and local industry.
- Quick response on mailing list (Inria engineer support).
- Hosted Luis Rodriguez, Greg Kroah Hartman (2 mo. each)
- MOU with the Linux Foundation

Supervision of interns, supported in part by the Linux Foundation.

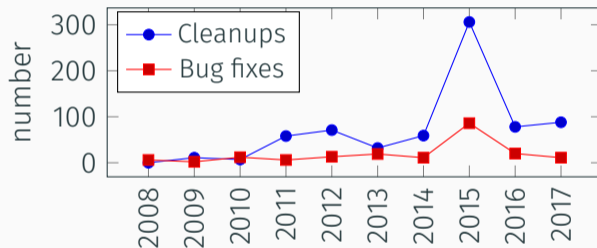
Kernel security-related project funded by the Core Infrastructure Initiative.

# Impact: Patches in the Linux kernel

Over 5500 Linux kernel commits up to Linux v4.15 (Jan 2018).

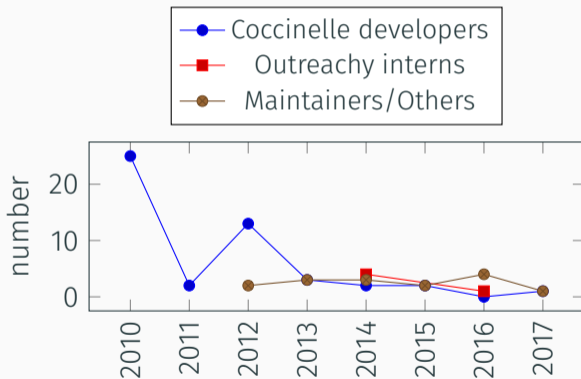


# Impact: Cleanup vs. bug fix changes among maintainer patches using Coccinelle

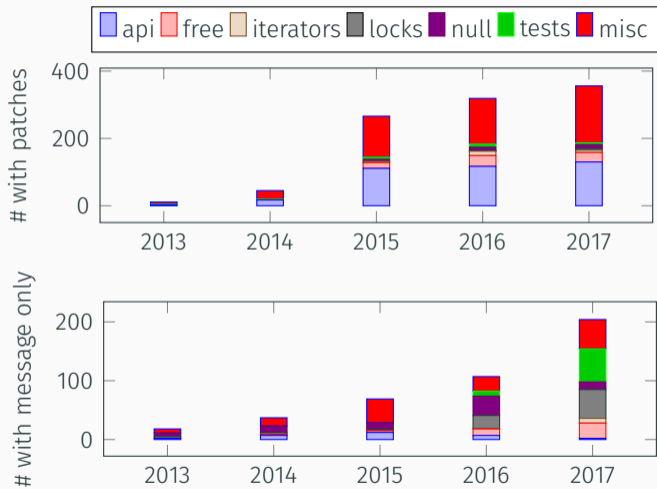


# Impact: Semantic patches in the Linux kernel


59 semantic patches in Linux v4.15 (Jan 2018).



# Impact: 0-day reports mentioning Coccinelle per year





A circular profile picture of Thomas Gleixner, a man with short brown hair, wearing a plaid shirt, smiling slightly.

**Thomas Gleixner**

Long time Linux kernel hacker with embedded background and a strong affinity to impossible missions.

Fellow

×

## Impact: Comments from users

Date: Wed, 20 May 2015 20:35:42 +0200 (CEST)

From: **Thomas Gleixner** <tglx@linutronix.de>

I think you are doing that conversion wrong. You should first change all handlers which use the irq argument to:

```
handler(unsigned __irq, *desc)
```

and add the local variable

```
unsigned irq;
```

or

```
unsigned irq = irq_desc_get_irq(desc);
```

for those.

...

**And you should do that with scripting aid. Coccinelle is the proper tool for this.**

...

It's really important to do that with scripts. It seems you try to do it via compile testing. But that will fail as you CANNOT execute all possible config combinations.

## Lessons learned

- Tools must be visible to the target community.
- Tools must be easy to install.
- Tools must be easy to use, following the habits of the target community.
- Tools must be robust.
- Support must be available to tool users.

- Used in over 5500 Linux kernel patches
  - Packaged for Debian, Ubuntu, Gentoo, FreeBSD, etc.
  - Also used by wine, systemd, qemu, riot, etc.
  - Some support for C++
- 59 semantic patches in the Linux kernel
  - Integrated with the Linux kernel 0-day build testing service

## Other activities, inspired by the results of Coccinelle

[Diagnosys \[ASE 2012\]](#): Plugging of Linux kernel safety holes

Best paper.

[Hector \[DSN 2013\]](#): Detection of missing resource release bugs.

Carter award paper.

[JMake \[DSN 2017\]](#): Feedback on compilation status in the presence of configurability.

[Prequel \[USENIX ATC 2017\]](#): Pattern-based commit query language

[ITrans \[ANR PRCI\]](#) Driver porting by inference of semantic patches from examples

# How to get involved?

## Internships:

- GSoC (for students)
- Outreachy (for women and some other underrepresented groups)
- With the Coccinelle team at LIP6

## On your own:

- Read git logs, mailing lists (lkml.org, lwn.net, kernel-janitors).
- Run tools (make coccicheck, checkpatch) on drivers/staging code.
- Look for underused API functions.
- Read all the code in a subsystem and find inconsistencies

# How to get involved?

## Internships:

- GSoC (for students)
- Outreachy (for women and some other underrepresented groups)
- With the Coccinelle team at LIP6

## On your own:

- Read git logs, mailing lists (lkml.org, lwn.net, kernel-janitors).
- Run tools (make coccicheck, checkpatch) on drivers/staging code.
- Look for underused API functions.
- Read all the code in a subsystem and find inconsistencies

**Read Documentation/SubmittingPatches!**

## Conclusion

- Targeting a specific problem of a specific community makes it possible to have an impact.
- Software development tools fit well with the distributed nature of open source development.
- Feedback from the user community motivates further research.
- Current work: Automatic inference of transformation rules to automate driver backporting and forwardporting

<http://coccinelle.lip6.fr/>