Software evolution and bug finding using Coccinelle

Julia Lawall (Inria/LIP6-Whisper team, Julia.Lawall@lip6.fr)

February 20, 2019

What is the Linux kernel?

An open-source operating system, known for:

- Reliability:

```
14:51:09 up 460 days,  3:39,  1 user,  ...
```

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

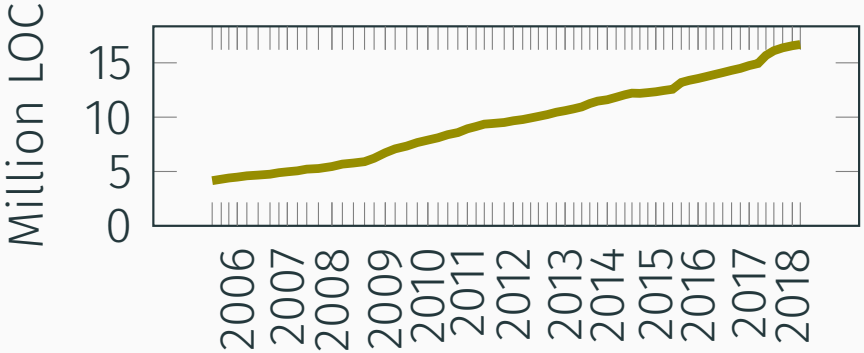
- Low cost per unit

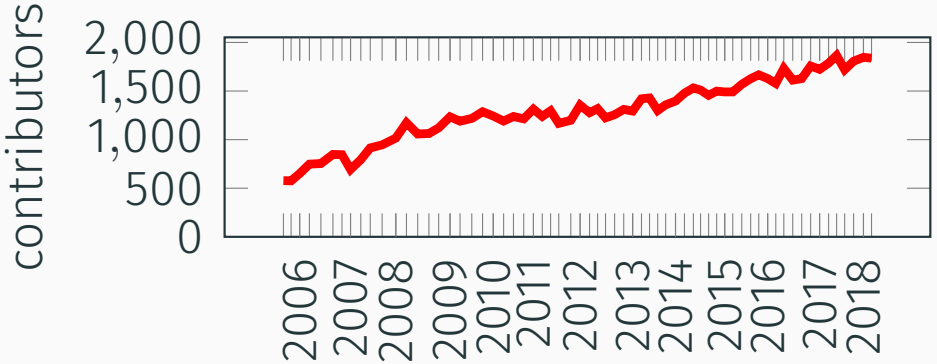
Some history

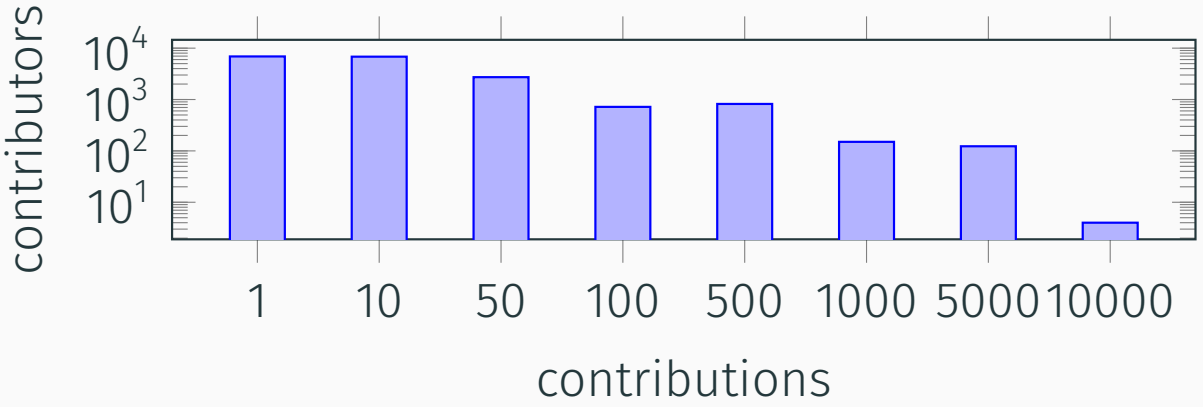
First release in 1991.

- 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: kzalloc

$\Longrightarrow$ Collateral evolution: Merge kmalloc and memset into kzalloc

```
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));
```

Example

Evolution: A new function: kzalloc

$\Longrightarrow$ Collateral evolution: Merge kmalloc and memset into kzalloc

```
fh = kzalloc(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: kzalloc

⟹ Collateral evolution: Merge kmalloc and memset into kzalloc

```
fh = kzalloc(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.
- $\Longrightarrow$ 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;
  }
}
```

4 instances in Linux v4.20 (December, 2018)

Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
- Changes may be widely scattered across the code base.
- Changes may come in many variants.

Assessment

- Changes may involve scattered code fragments and data and control flow relationships between them.
- Changes may be widely scattered across the code base.
- Changes may come in many variants.

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).

Starting point: a patch

```
@@ -1348,9 +1348,8 @@
- fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
+ fh = kzalloc(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));
```

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.

Creating a semantic patch: kmalloc → kzalloc

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 → kzalloc

Eliminate irrelevant code

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



...



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

Creating a semantic patch: kmalloc → kzalloc

Describe transformations

```
- fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
+ fh = kzalloc(sizeof(struct zoran_fh), GFP_KERNEL);
  ...

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

Creating a semantic patch: kmalloc → kzalloc

Abstract over subterms

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

@@

- x = kmalloc(E1,E2);
+ x = kzalloc(E1,E2);
  ...

- memset(x, 0, E1);
```

Creating a semantic patch: kmalloc → kzalloc

Refinement

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

- x = kmalloc(E1,E2);
+ x = kzalloc(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

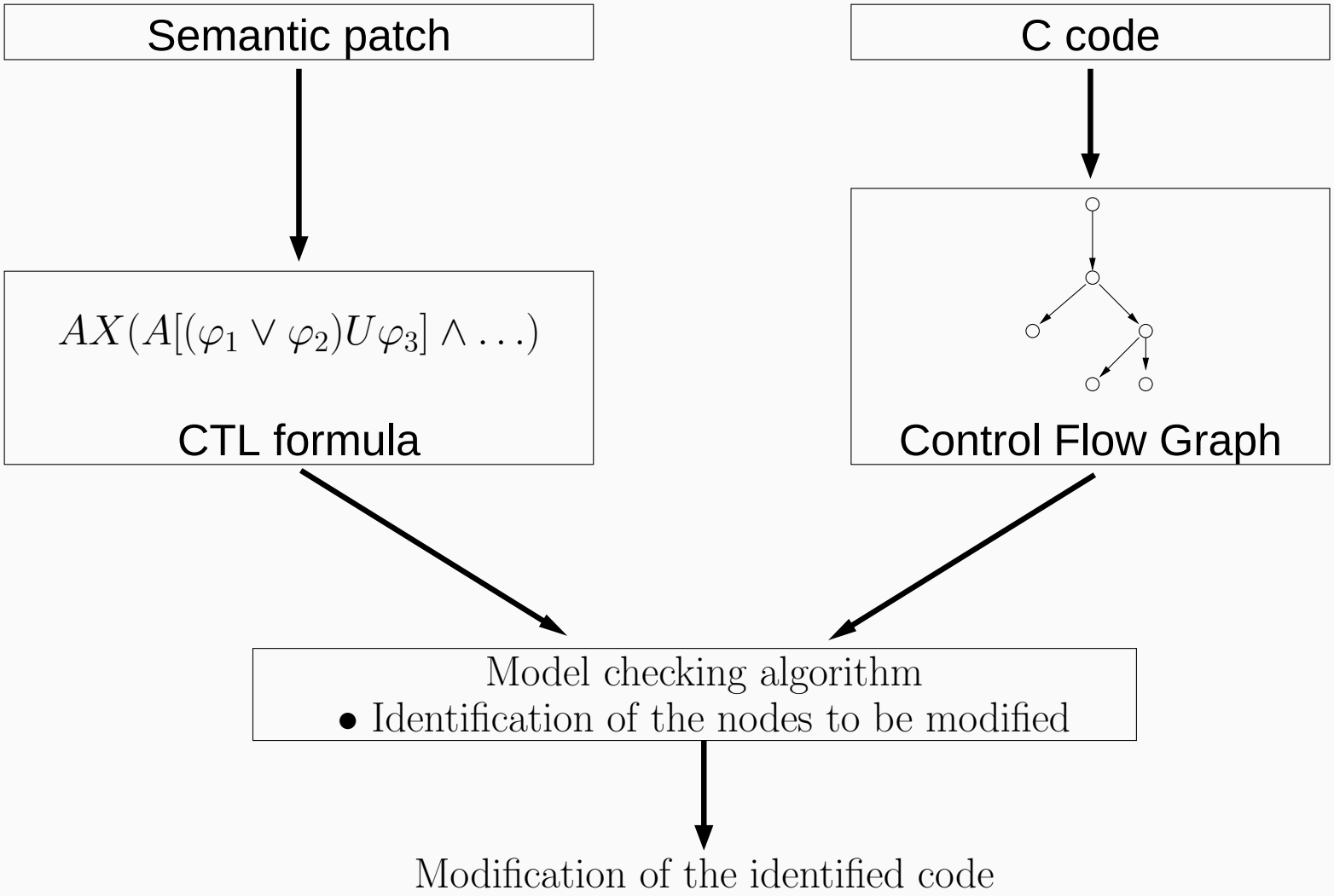
- Correctly updates 14 occurrences
  - 5 false positives, could be eliminated by more "`when`" tests
- Other opportunities:
  - `acpi_os_allocate` → `acpi_os_allocate_zeroed`
  - `dma_pool_alloc` → `dma_pool_zalloc`
  - `dma_alloc_coherent` → `dma_zalloc_coherent`
  - `kmem_cache_alloc` → `kmem_cache_zalloc`
  - `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.

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.

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.

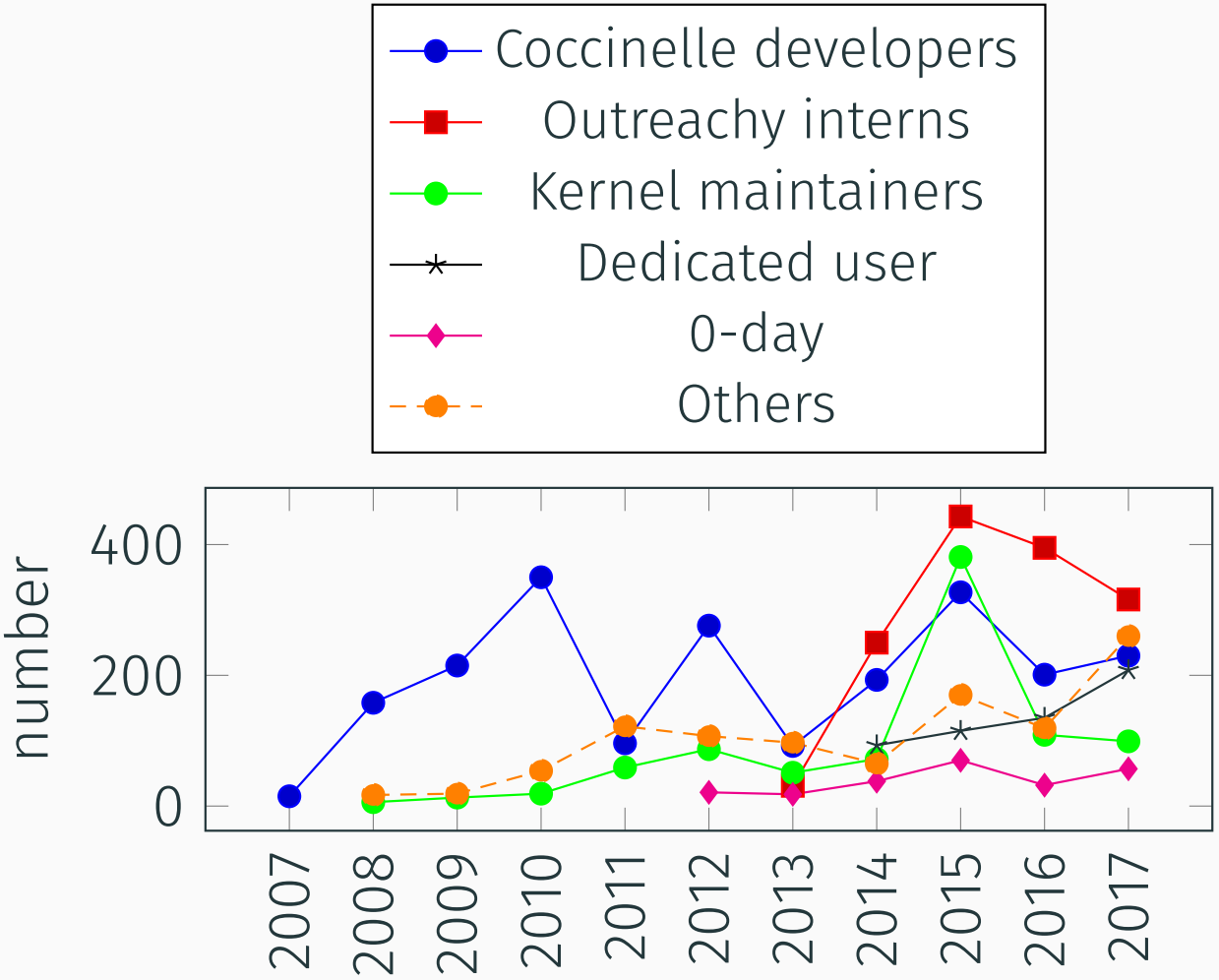
Isomorphisms, to reduce semantic patch size

- `X == NULL => !X`

Implementation via translation to CTL

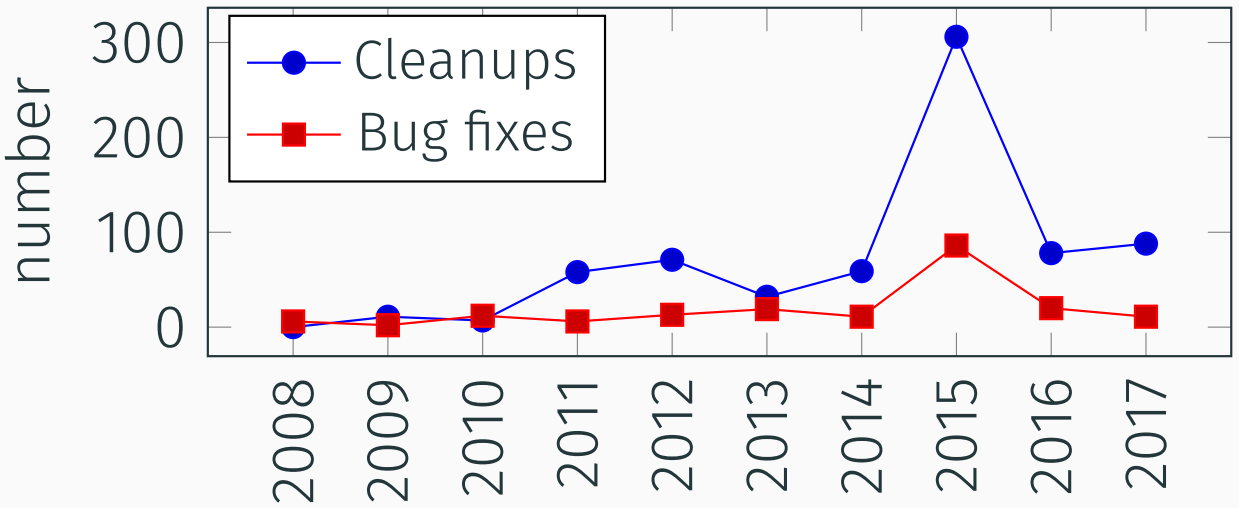
- Allows ∀ and ∃ quantification over paths.
- ∀ and ∃ can be mixed in a single rule.

Impact: Patches in the Linux kernel



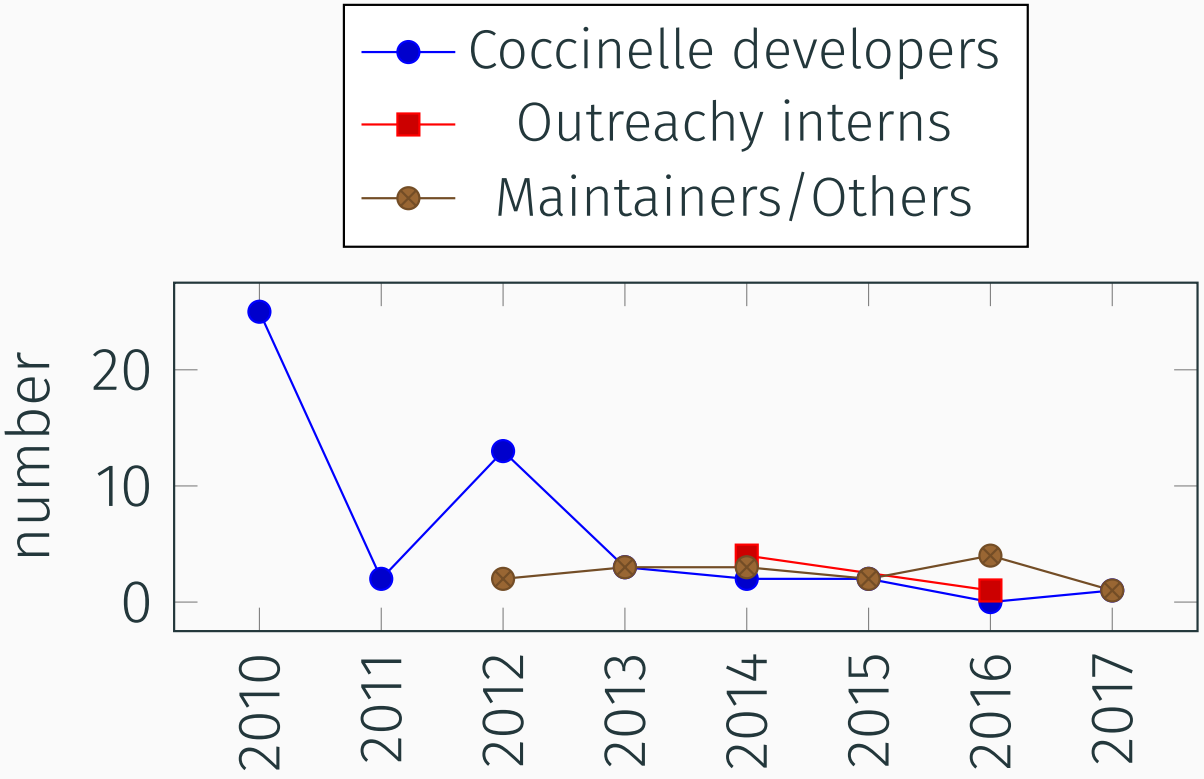
Almost 6900 as of Feb. 2019

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

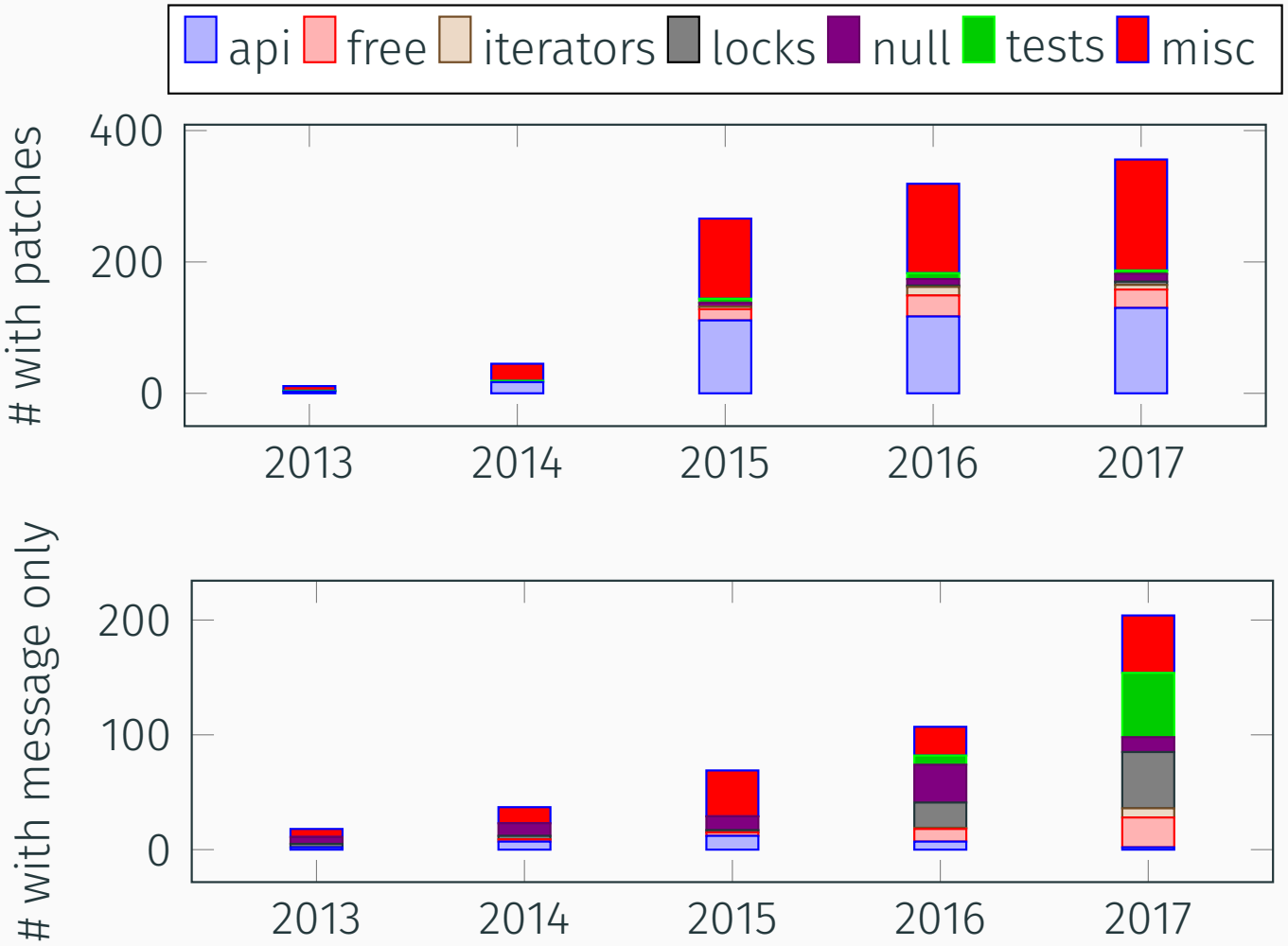


Impact: Semantic patches in the Linux kernel

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



Impact: 0-day reports mentioning Coccinelle per year



Current status

- Used in almost 6900 Linux kernel patches
  - Also used by wine, systemd, qemu, riot, zephyr, etc.
  - Packaged for Debian, Ubuntu, Gentoo, FreeBSD, etc.
  - Some support for C++
- Around 60 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

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

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
- Probably, everyone in this room uses some Coccinelle modified code!

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
- Probably, everyone in this room uses some Coccinelle modified code!

http://coccinelle.lip6.fr/