Coccinelle:
10 Years of Automated Evolution in the Linux Kernel

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

September 27, 2019

Our target

- Maintenance of large, critical software.
- C code.
- Focus on the Linux kernel.


ds: 0010 es: 0010 ss: 0010
Process swapper (pid: 0, stackpage=c8245000)
Stack: 00000000 00000001 f6467c68 f7c765c8 00000000 00000001 c8106dcc f6467c68
00000001 f7c765c8 f7c3200d 00000957 c835e6a9 c8192a8 f7c765c8 00000001
c835e7ec 01c3200d 00000000 f7c765c8 c835e6a4 c819ca40 00000001 f7c3200d
Call Trace: [<c8106dcc>] end_that_request_first [kernel] 0x5a
[<c81902a8>] ide_end_request [kernel] 0x58
[<c819ca40>] ide_dma_intr [kernel] 0x78
[<c8191e45>] ide_intr [kernel] 0x45
[<c819ca40>] ide_dma_intr [kernel] 0x0
[<c8107c7a>] handle_IRQ_event [kernel] 0x3a
[<c8107df8>] do_IRQ [kernel] 0xb8
[<c8106dc8>] default_idle [kernel] 0x0
[<c8105000>] stext [kernel] 0x0
[<c8106dc8>] default_idle [kernel] 0x0
[<c8106dc8>] default_idle [kernel] 0x0
[<c8105000>] stext [kernel] 0x0
[<c8106e03>] default_idle [kernel] 0x23
[<c8106e5f>] cpu_idle [kernel] 0x2f
Code: 8b 53 fc 8b 02 85 45 f0 74 10 6a 00 52 e8 7e f8 ff ff 83 c4
<0>Kernel panic: Aiee, killing interrupt handler!
In interrupt handler - not syncing

Software lifecycle

Great idea!

Software lifecycle

Great idea!

↓

Quick and dirty implementation

Software lifecycle

Great idea!

↓

Quick and dirty implementation

↓

People like the idea and start to use the code

Software lifecycle

Great idea!

↓

Quick and dirty implementation

↓

People like the idea and start to use the code

↓

Feature requests

Software lifecycle

Great idea!

↓

Quick and dirty implementation

↓

People like the idea and start to use the code

↓ ↓

Feature requests Bugs!

Software lifecycle

Great idea!

↓

Quick and dirty implementation

↓

People like the idea and start to use the code

↓ Feature requests

↓ Bugs!

Software lifecycle

Great idea!

↓

Quick and dirty implementation

↓

People like the idea and start to use the code

↓ ↓

Feature requests Bugs!

Software lifecycle

Great idea!

↓

Quick and dirty implementation

↓

People like the idea and start to use the code

↓ ↓ ↓

Feature requests | Bugs! | Attacks!!!

Software lifecycle

Great idea!

↓

Quick and dirty implementation

↓

People like the idea and start to use the code

↓ ↓ ↓

Feature requests | Bugs! | Attacks!!!

Software lifecycle

As software matures, it is continually necessary to rethink design decisions.

- Can induce pervasive changes.

Software lifecycle

As software matures, it is continually necessary to rethink design decisions.

- Can induce pervasive changes.

Maturation of software often comes with code growth:

- Linux kernel: from 122K LOC in 1994 (v1.0) to 18M LOC in 2019

Key challenge

As software grows, how to ensure its continued maintenance?

- Updating interfaces is easy.
  Make functions and data structures:
  - More efficient
  - Easier to use correctly
  - Better adapted to their usage context

Key challenge

As software grows, how to ensure its continued maintenance?

- Updating interfaces is easy.
  Make functions and data structures:
  - More efficient
  - Easier to use correctly
  - Better adapted to their usage context
- Updating the uses of interfaces gets harder as the software grows.
  - More time consuming
  - More error prone

Key challenge

As software grows, how to ensure its continued maintenance?

- Updating interfaces is easy.
  Make functions and data structures:
  - More efficient
  - Easier to use correctly
  - Better adapted to their usage context
- Updating the uses of interfaces gets harder as the software grows.
  - More time consuming
  - More error prone

Developers may hesitate to make needed changes.

Example: `init_timer` → `setup_timer`

Initializing a timer requires:

- The callback function to run when the timer expires
- The data that should be passed to that callback function

Example: `init_timer` → `setup_timer`

Initializing a timer requires:

- The callback function to run when the timer expires
- The data that should be passed to that callback function

Original initialization strategy (present in Linux v1.2.0):

```
init_timer(&ns_timer);
ns_timer.data = 0UL;
ns_timer.function = ns_poll;
```

Example: `init_timer` → `setup_timer`

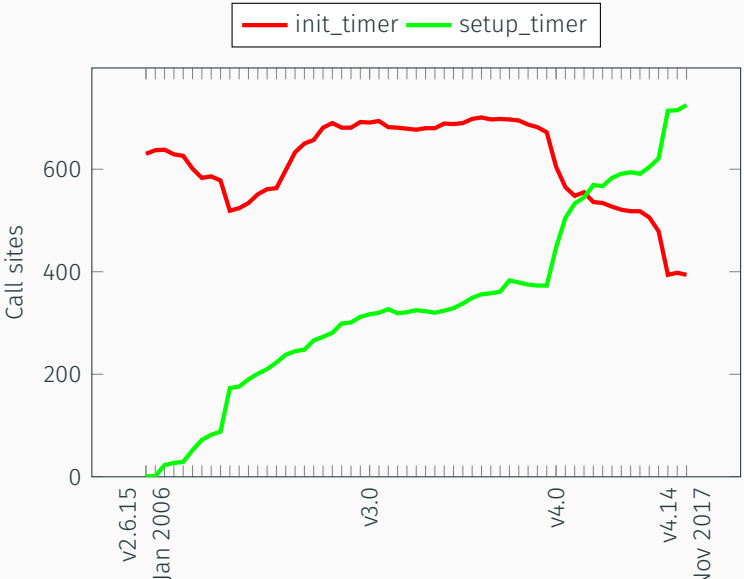
Replacement initialization strategy (introduced in Linux v2.6.15, Jan. 2006):

```
setup_timer(&ns_timer, ns_poll, 0UL);
```

Advantages:

- More concise
- More uniform
- More secure

Example: `init_timer` → `setup_timer`



Example: `kzalloc` → `devm_kzalloc`

`kzalloc`:

- Basic Linux kernel memory allocation function
- Requires explicit kfree (sometimes forgotten, causing memory leaks)

Example: `kzalloc` → `devm_kzalloc`

`kzalloc`:

- Basic Linux kernel memory allocation function
- Requires explicit kfree (sometimes forgotten, causing memory leaks)

Observation: Device drivers often allocate memory in a similar way

- Allocate in `probe`, free in `remove`.

Example: `kzalloc` → `devm_kzalloc`

`kzalloc`:

- Basic Linux kernel memory allocation function
- Requires explicit kfree (sometimes forgotten, causing memory leaks)

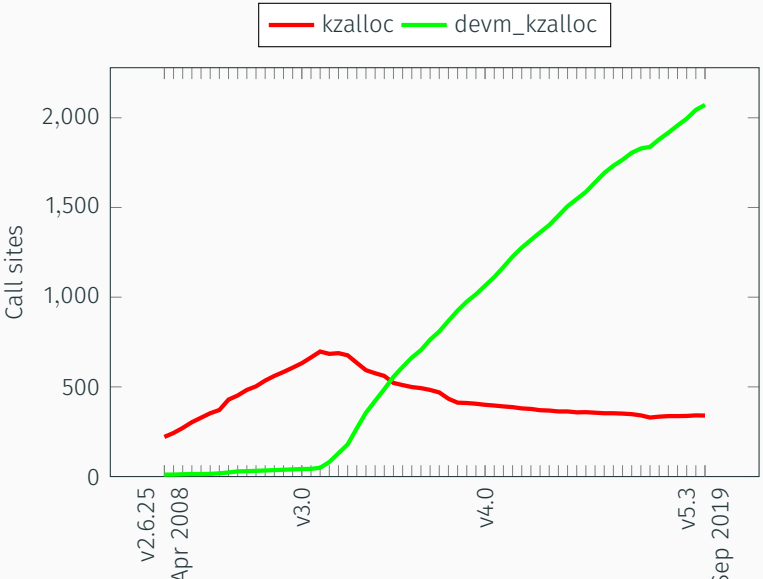
Observation: Device drivers often allocate memory in a similar way

- Allocate in `probe`, free in `remove`.

`devm_kzalloc`:

- Usable in device driver probe functions
- Free done implicitly by the device layer

Example: `kzalloc` → `devm_kzalloc`



Example: missing of_node_puts

Device node structures are reference counted:

- of_node_get to access the structure.
- of_node_put to let go of the structure.

Iterators, e.g., `for_each_child_of_node`, `put` one value and `get` another.

- Explicit `put` needed on `break`, `return`, `goto` out of the loop.
- Often forgotten.

Example: missing of_node_puts

Device node structures are reference counted:

- of_node_get to access the structure.
- of_node_put to let go of the structure.

Iterators, e.g., `for_each_child_of_node`, `put` one value and `get` another.

- Explicit `put` needed on `break`, `return`, `goto` out of the loop.
- Often forgotten.
- How to find all of the other occurrences?

Example: missing of_node_puts

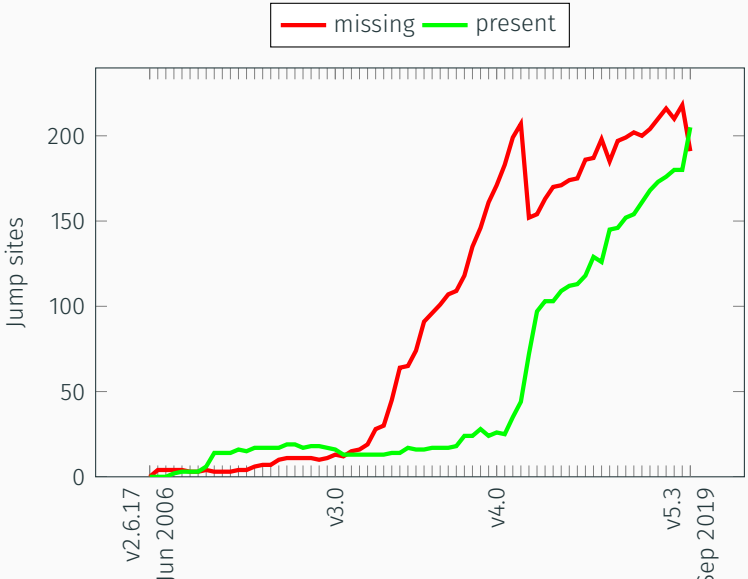
Device node structures are reference counted:

- of_node_get to access the structure.
- of_node_put to let go of the structure.

Iterators, e.g., `for_each_child_of_node`, `put` one value and `get` another.

- Explicit `put` needed on `break`, `return`, `goto` out of the loop.
- Often forgotten.
- How to find all of the other occurrences?
- How to prevent reoccurrences?

Example: missing `of_node_puts`



Coccinelle to the rescue!

What is Coccinelle?

- Pattern-based tool 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).
- Goal: fit with the existing habits of the Linux programmer.

Semantic patches

Code fragments, annotated with − and +.

- Like patches, but independent of irrelevant details (line numbers, spacing, variable names, etc.)
- Configuration independent (does not use the C preprocessor).

```
@@
expression e;
@@
- f(0,e);
+ f_zero(e);
```

Example: Creating an `init_timer` → `setup_timer` semantic patch

A patch: derived from drivers/atm/nicstar.c

```
-                init_timer(&ns_timer);
+                setup_timer(&ns_timer, ns_poll, 0UL);
                 ns_timer.expires = jiffies + NS_POLL_PERIOD;
-                ns_timer.data = 0UL;
-                ns_timer.function = ns_poll;
```

Example: Creating an `init_timer` → `setup_timer` semantic patch

Remove irrelevant code:

```
-                init_timer(&ns_timer);
+                setup_timer(&ns_timer, ns_poll, 0UL);
                 ...
-                ns_timer.data = 0UL;
-                ns_timer.function = ns_poll;
```

Example: Creating an `init_timer` → `setup_timer` semantic patch

Abstract over subterms:

```
@@
expression timer,fn_arg,data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...
-               timer.data = data_arg;
-               timer.function = fn_arg;
```

Example: Creating an `init_timer` → `setup_timer` semantic patch

Generalize a little more:

```
@@
expression timer,fn_arg,data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...
-               timer.data = data_arg;
                ...
-               timer.function = fn_arg;
```

Results

Dataset: 598 Linux kernel `init_timer` files from different versions.

- 828 calls.
- Our semantic patch updates 308 of them.

Results

Dataset: 598 Linux kernel `init_timer` files from different versions.

- 828 calls.
- Our semantic patch updates 308 of them.

Untreated example: drivers/tty/n_gsm.c:

```
init_timer(&dlci->t1);
dlci->t1.function = gsm_dlci_t1;
dlci->t1.data = (unsigned long)dlci;
```

Example: Creating an `init_timer` → `setup_timer` semantic patch

Extended semantic patch:

```
@@
expression timer,fn_arg,data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...

-               timer.data = data_arg;
                ...
-               timer.function = fn_arg;
```

Example: Creating an `init_timer` → `setup_timer` semantic patch

Extended semantic patch:

```
@@
expression timer,fn_arg,data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...
(
-               timer.data = data_arg;
                ...
-               timer.function = fn_arg;
|
-               timer.function = fn_arg;
                ...
-               timer.data = data_arg;
)
```

Covers 656/828 calls.

Example: Creating an `init_timer` → `setup_timer` semantic patch

Remaining issues

- Some code initializes the function and data before calling `init_timer`.
- Some timers have no data initialization, default to 0.
- Coccinelle sometimes times out.

Complete semantic patch

- 6 rules, 68 lines of code.
- Covers 808/828 calls.
- TODO: Some timers have no local `function` or `data` initialization.

Assessment

Understandable:

- Small, readable transformation rule fixes a large amount of code.

Reliable:

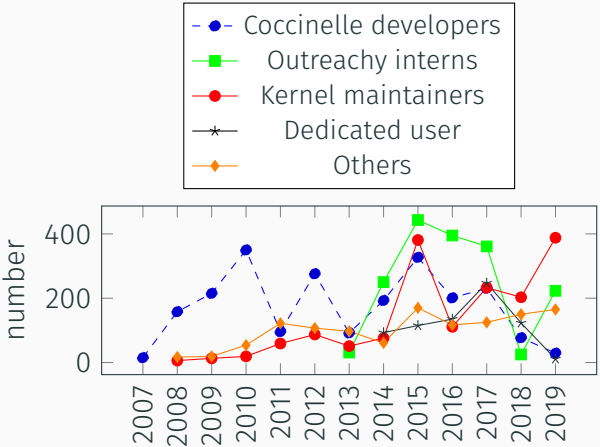
- No risk of confusion, with `bfad_init_timer`, `nes_nic_init_timer`, etc.
- Checks all files, for all architectures.

Iterative development:

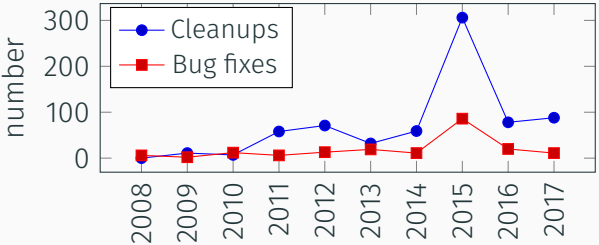
- Write rule.
- Run Coccinelle.
- Check results.
- Improve rule for omissions and incorrect transformations.

Impact: Patches in the Linux kernel

Over 7000 Linux kernel commits up to Linux v5.3 (Sep 2019).



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



Impact: Maintainer use examples

TTY. Remove an unused function argument.

- 11 affected files.

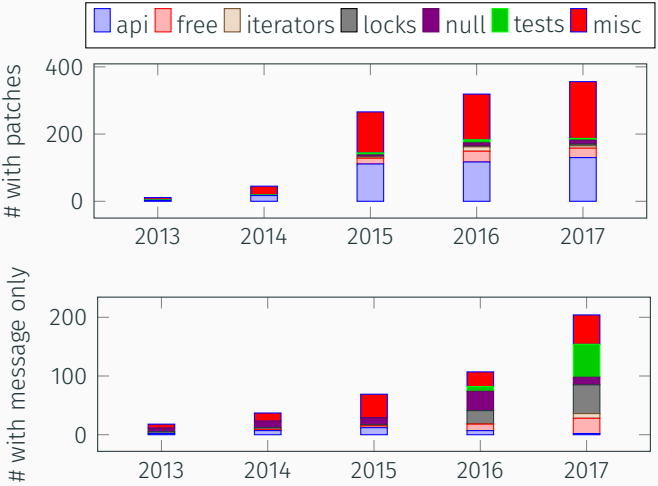
DRM. Eliminate a redundant field in a data structure.

- 54 affected files.

Interrupts. Prepare to remove the irq argument from interrupt handlers, and then remove that argument.

- 188 affected files.

Impact: 0-day reports mentioning Coccinelle per year



Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 7000 commits in the Linux kernel based on Coccinelle.

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 7000 commits in the Linux kernel based on Coccinelle.
- Future work: Automatic generation of semantic patches from examples.

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 7000 commits in the Linux kernel based on Coccinelle.
- Future work: Automatic generation of semantic patches from examples.
- Probably, everyone in this room uses some Coccinelle modified code!

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 7000 commits in the Linux kernel based on Coccinelle.
- Future work: Automatic generation of semantic patches from examples.
- Probably, everyone in this room uses some Coccinelle modified code!

http://coccinelle.lip6.fr/