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.
Software lifecycle

Great idea!
Software lifecycle

Great idea!

Quick and dirty implementation
Great idea!
\[ \downarrow \]
Quick and dirty implementation
\[ \downarrow \]
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!
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!
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!!!

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

- Can induce pervasive changes.
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.
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):

```c
init_timer(&ns_timer);
ns_timer.data = 0UL;
ns_timer.function = ns_poll;
```
Replacement initialization strategy (introduced in Linux v2.6.15, Jan. 2006):

```c
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)

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: \texttt{kzalloc} \rightarrow \texttt{devm\_kzalloc}

\textit{kzalloc}:

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

\textbf{Observation}: Device drivers often allocate memory in a similar way

- Allocate in \texttt{probe}, free in \texttt{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`

![Graph showing call sites of `kzalloc` and `devm_kzalloc` over time](image-url)

- `kzalloc`
- `devm_kzalloc`

Call sites over time:
- April 2008
- v3.0
- v4.0
- v5.3

Versions:
- v2.6.25
- v3.0
- v4.0
- v5.3
- v5.3

Date:
- April 2008
- Sep 2019

Note: v5.3 is highlighted with a green line.
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.
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

![Graph showing jump sites](image)

- **missing**
- **present**
Coccinelle to the rescue!
What is Coccinelle?

- Pattern-based tool for matching and transforming C code
- 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).

```c
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;`
```
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:

\[\begin{align*}
\text{expression } & \text{timer, fn_arg, data_arg;} \\
\text{\textbf{init Timer}} & \text{(~timer);} \\
\text{\textbf{setup Timer}} & \text{(~timer, fn_arg, data_arg);} \\
\text{...} \\
\text{timer.data} & = \text{data_arg;} \\
\text{timer.function} & = \text{fn_arg};
\end{align*}\]
Example: Creating an `init_timer` → `setup_timer` semantic patch

Generalize a little more:

```plaintext
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:

```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` $\rightarrow$ `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 \texttt{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

![Graph showing cleanup vs. bug fix changes from 2008 to 2017. The y-axis represents the number of changes, ranging from 0 to 300. The x-axis represents the years from 2008 to 2017. The blue line represents cleanups, and the red line represents bug fixes. There is a significant spike in bug fixes in 2015 compared to other years.]
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

![Impact Chart]

- **Impact Chart**
  - Title: Impact: 0-day reports mentioning Coccinelle per year
  - Years: 2013 to 2017
  - Categories:
    - api
    - free
    - iterators
    - locks
    - null
    - tests
    - misc
  - Data points for each year:
    - Number of patches
    - Number of messages only

---

**Legend**
- api
- free
- iterators
- locks
- null
- tests
- misc

---

**Impact Analysis**
- 2013:
  - Few reports, primarily api and null
  - Messages only: minimal
- 2014:
  - Similar to 2013, with slight increase in tests
  - Messages only: negligible
- 2015:
  - Significant increase in all categories, especially api and free
  - Messages only: moderate increase
- 2016:
  - Continued growth across all categories
  - Messages only: substantial
- 2017:
  - Steady growth, especially in null and tests
  - Messages only: highest among all years

---

**Conclusion**
- Coccinelle has seen increased impact over the years, with a significant rise in reports mentioning various categories.
- Messages only have also shown a notable increase, indicating broader awareness or use.

---

**Note**
- Additional data and analysis can be found in the attached documents.
Conclusion

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

http://coccinelle.lip6.fr/
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/