Automating Program Transformation with Coccinelle

Julia Lawall, Gilles Muller

Automating Program Transformation with Coccinelle

Julia Lawall[0000−0002−1684−1264] and Gilles Muller[0000−0002−0000−8569]⋆

Inria
{First.Last}@inria.fr
https://coccinelle.gitlabpages.inria.fr/website/

Abstract. Coccinelle is a program matching and transformation engine for C code. This paper introduces the use of Coccinelle through a collection of examples targeting evolutions and bug fixes in the Linux kernel.

Keywords: Linux kernel, Coccinelle, program transformation.

1 Introduction

It is the dream of every programmer to have a tool that will automatically traverse their software and make any kind of changes that the programmer wants. Early efforts include `sed` and `awk` that permit developers to write simple search-and-replace patterns involving regular expressions [11, 10]. Such tools are powerful, but regular expressions are hard to write, are error prone, have a limited view of the code, and are not aware of the programming language syntax. Tools designed according to the Visitor pattern [6], such as CIL [20], have been developed, but these require the user to become familiar with the visitor's chosen internal representation for the programming language. Must easier to use, common semantics-preserving changes, known as *refactorings*, were classified by Fowler [5], and are provided as a collection of black-box tools within integrated development environments such as Eclipse [3]. But in real software development, it is often necessary to perform changes that do not fit within a tidy collection of common refactorings. These include repetitive bug fixes, that intrinsically change the semantics of the code, and changes that respect the invariants that the developer knows, but that are difficult to automatically recover from the code base.

Coccinelle is a program matching and transformation engine for C code [15, 22]. The goal of Coccinelle is to make it easy for software developers to express code transformations and apply these transformations across a large C code base. Coccinelle's transformation specification language SmPL (Semantic Patch Language) allows transformations to be expressed using code fragments, annotated with `-` and `+`, for lines to remove and add, respectively, mirroring the familiar

⋆ Gilles Muller passed away before the writing of this paper. He initiated the Coccinelle project in 2004 and supported its development over the next 17 years.

patch syntax [18]. Such pattern-matching rules can include scripts written in Python or OCaml, for greater expressiveness. Coccinelle was originally designed for updating Linux kernel device drivers to take into account evolutions in Linux kernel internal APIs [22], and accordingly supports a very large portion of the C language. It has been used in over 9000 Linux kernel commits, and is used in other C software projects, such as `wine` [25, 27], `systemd` [26], and `git` [7].

Previous works on Coccinelle have presented the design of the tool [22], the semantics of its transformation language SmPL [1], the use of Coccinelle for finding bugs in Linux kernel code [16, 23], and a retrospective after 10 years of use, including an enumeration and assessment of the design decisions [15]. Tutorials on Coccinelle have been presented at developer conferences, some of which are available as videos [12–14]. This paper takes advantage of the written format to make a deep dive into SmPL, to describe the reasoning that goes into constructing a semantic patch: how to identify a problem for which Coccinelle can be appropriate, how to sketch a solution for such a problem using SmPL, and how to iteratively make that solution more powerful and more automatic. Our examples focus on the Linux kernel, but should be applicable to other kinds of C software.

The rest of this paper is organized as follows. Section 2 provides some background on the Linux kernel, its development challenges, and the opportunities that it raises for automatic program transformation. Section 3 presents a simple and classic example, the transformation of a call to the kernel memory allocation function `kmalloc`, followed by a zeroing call to `memset`, into a single call to the zeroing kernel memory allocation function `kzalloc`. Section 4 scales this kind of transformation up to the detection of memory leaks involving kernel `device_node` structures. Section 5 considers detection of anomalies in the use of the Linux kernel memory allocation flags, `GFP_KERNEL` and `GFP_ATOMIC`. Each of these examples emphasizes the aspect of exploration facilitated by Coccinelle – the use of Coccinelle scales naturally from simple rules with a limited scope that may have false positives, but get the job done, to more complex rules that capture a wider variety of conditions in a more accurate way. It is hoped that this work can serve as a reference for a developer who wants to use Coccinelle for the first time or who wants to explore some of its more advanced features.

2 Background

The original and primary target of Coccinelle is the Linux kernel. The Linux kernel poses a huge maintenance challenge. It amounts to over 21 million lines of code in Linux v5.16 (January 2022), accepts contributions from over 4000 developers per year, and undergoes frequent and large-scale changes, motivated by security, performance, new hardware features, etc. As part of the Linux kernel's evolution, it often occurs that some API function is found to be unsuitable, the function is redefined in some way, and then the uses of the function have to be modified across the kernel. These modifications may involve changes in the ar-

Table 1. Usage of common functions in the files of Linux 5.16, `drivers/usb/atm`. ✓ indicates that the given API function is called at least once in the given file.

		cxacru.c	speedtch.c	ueagle-atm.c	usbatm.c	xusbatm.c
atm	`usbatm_usb_probe`	✓	✓	✓	–	✓
usb	`interface_to_usbdev`	✓	✓	✓	✓	✓
specific	`usb_submit_urb`	✓	✓	✓	✓	–
	`usb_set_intfdata`	–	✓	–	✓	✓
kernel	`request_firmware`	✓	✓	✓	–	–
generic	`wait_for_completion`	✓	–	–	✓	–
	`mutex_lock`	✓	–	✓	✓	–
	`init_timer`	✓	✓	–	✓	–
	`kzalloc`	✓	✓	✓	✓	–

guments and return values, triggering the need for further changes in the usage context.

Intuitively, sustaining the high rate of development on the huge code base of the Linux kernel may seem like an impossible task. Indeed one may think of one's own small software projects, where often one decides to just live with some unsuitable code structure to avoid the need to do all of the work required to change it. Scaling this work up to 21 million lines of code, and managing to make all the changes correctly is a real challenge.

A mitigating factor is that the Linux kernel code base contains a lot of repetition [2]. For example, consider the kernel API functions (Table 1) used in the various files of the Linux v5.16 directory `drivers/atm`, containing Asynchronous Transfer Mode (ATM) network device drivers. Many of the key kernel API functions are used in many of the drivers. This commonality occurs at all levels – we see functions that are specific to ATM drivers, functions that are generic to USB drivers, and functions that are generic to the entire kernel, including `kzalloc` for memory allocation, which we use as an initial case study in Section 3. This pattern raises hope that not only may these functions be reused across the various drivers, but they may also be used in similar ways. If this is the case, then it may be possible to automate any needed changes in their usage.

Repetitive API usages raise the opportunity for using a tool to script API usage changes. That is, rather than manually collecting the relevant files (*e.g.*, with `grep`) and then tracking down the relevant usage contexts (*e.g.*, with search in an editor), it could be faster and more reliable to write a transformation rule and then leave the job of finding the relevant code and making the changes to a transformation tool. This is the role of Coccinelle, that is the focus of this paper.

3 Coccinelle in a Nutshell, Illustrated by `kzalloc`

Coccinelle offers a pattern-based language for matching and transforming C code. It has been under development since 2005 and open source since 2008. An important goal of Coccinelle is to fit with the habits of Linux kernel developers. The

Linux kernel follows an email-based development model, where developers exchange patches describing their proposed changes, and thus developers are used to creating, reading, and applying them. Accordingly, Coccinelle was designed to allow code changes to be expressed using patch-like code patterns. We refer to these as *semantic patches*, because they are like patches, but their application takes into account the program control flow, and thus part of its semantics.

A common use of Coccinelle is to reorganize a collection of one or more API functions. Accordingly, to present Coccinelle, we consider a simple example, the merging of uses of the kernel memory allocation function `kzalloc` followed by a zeroing of the allocated memory with `memset`, into a single call to the kernel zeroing memory allocation function `kzalloc`. An example of this change is shown, as a patch, in Figure 1. The change itself is simple: replace `kmalloc` by `kzalloc` and drop the now redundant call to `memset`. Still, finding the opportunities for the change is complex: The calls to `kmalloc` and `memset` are typically not contiguous – as illustrated in Figure 1, there is often at least some error-handling code in between them. Furthermore, some `kmalloc`s have no following `memset`s and some `memset`s have no preceding `kmalloc`s, so simply using `grep` to find calls to one or the other will return many irrelevant code locations. Finally, some `memset`s may serve to reinitialize a structure rather than initialize a just-allocated one. Even though calls to both `kmalloc` and `memset` are present, we do not want to create a call to `kzalloc` in these cases. Coccinelle is designed to help with these challenges.

```
@@ -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));
```

Fig. 1. An instance of the conversion of `kmalloc` and `memset` to `kzalloc`.

3.1 First steps

To develop a `kmalloc-memset` semantic patch that is widely applicable across the Linux kernel code base, we take the patch of Figure 1 as a starting point, and consider how it can be made more generic.

The first step is to consider what parts of the patch in Figure 1 are generic to the change, and what parts are specific to a particular instance. For the `kmalloc-memset` transformation, it is necessary to have a call to `kmalloc` followed by a call to `memset`, where the second argument to `memset` should be

0. These terms will thus appear in the semantic patch exactly as they appear in Figure 1. On the other hand, some other terms in the patch of Figure 1 are important, not for their specific content, but for their relationship to other terms appearing in the affected code. This is the case for 1) the return value of `kmalloc` (*i.e.*, `fh`) and the first argument of `memset`, which must be the same expression, 2) the first argument of `kmalloc` (the size of the allocated region), that becomes the first argument of the call to `kzalloc` and should be the third (size) argument of `memset`, and 3) the second argument of `kmalloc` that becomes the second argument of `kzalloc`. These terms appear in the semantic patch as *metavariables*, *i.e.*, variables that can match against any term in the source code, but that must be matched consistently. The metavariables are declared between the initial pair of `@@`, at the place of the affected line numbers in the standard patch. The metavariables are furthermore declared with their types; all of the metavariables that are relevant to this change have type `expression`. Finally, some terms are not important to the change, such as the `if` statement between the calls to `kmalloc` and `memset`. Such terms are removed, and replaced by "`...`".[1] "`...`" matches any control-flow path from a source code term matching the pattern before the "`...`" to a source code term matching the pattern after the "`...`". Furthermore, by default, all such execution paths that do not lead to an error return must satisfy these constraints.

The resulting semantic patch is shown in Figure 2. It makes six changes in Linux v5.16, with no false positives. Figure 3 shows one change, in which the code separating the `kmalloc` and `memset` is more complex than a single `if`. All of the generated patches have been submitted to the Linux kernel. One received the feedback that a different zeroing function should be used (`kcalloc`). Four have been applied unchanged in `linux-next` as of March 25, 2022.

```
@@
expression res, size, flag;
@@
- res = kmalloc(size, flag);
+ res = kzalloc(size, flag);
  ...
- memset(res, 0, size);
```

Fig. 2. A first attempt at a `kmalloc` and `memset` to `kzalloc` semantic patch.

3.2 A refinement

While our experiment with the semantic patch in Figure 1 was completely successful on Linux v5.16, the semantic patch is not fully reliable. Figure 4 shows a false positive in `net/sunrpc/auth_gss/gss_krb5_keys.c`, in Linux v5.2. Here a `kmalloc` is indeed followed by a `memset`, according to our pattern, but the

[1] To prevent misreading, in the text, we always enclose SmPL ... in quotes

```
1 - port = kmalloc(sizeof(*port), GFP_KERNEL);
2 + port = kmalloc(sizeof(*port), GFP_KERNEL);
3   if (!port) {
4     rc = -ENOMEM;
5     goto __error;
6   }
7   rc = snd_seq_create_kernel_client(NULL, ...);
8   if (rc < 0)
9     goto __error;
10  system_client = rc;
11 - memset(port, 0, sizeof(*port));
```

Fig. 3. A successful change in `sound/core/seq/oss/seq_oss_init.c`.

`memset` is used to reinitialize the data to 0 (just before freeing the data, for security reasons), rather than to initialize the data to 0 as done by `kzalloc`.

```
1 - inblockdata = kmalloc(blocksize, gfp_mask);
2 + inblockdata = kzalloc(blocksize, gfp_mask);
3   if (inblockdata == NULL)
4           goto err_free_cipher;
5   ...
6   inblock.data = (char *) inblockdata;
7   inblock.len = blocksize;
8   ...
9   if (in_constant->len == inblock.len) {
10          memcpy(inblock.data, in_constant->data, inblock.len);
11  } else {
12          krb5_nfold(in_constant->len * 8, in_constant->data,
13                          inblock.len * 8, inblock.data);
14  }
15  ...
16 - memset(inblockdata, 0, blocksize);
17  kfree(inblockdata);
```

Fig. 4. An false positive for the `kmalloc` and `memset` semantic patch.

Indeed, by simply replacing the code between the `kmalloc` and the `memset` by "`...`", we have eliminated any constraints on the code found in the execution path between them. To limit the matches to the cases where the `memset` represents an initialization, we can add constraints on the matching of "`...`" using the keyword `when`. For inspiration, we consider how the allocated data is used in the false positive of Figure 4. The data allocated by the call to `kmalloc` on line 1 is used in the right side of an assignment on line 6, creating an alias through which it is subsequently initialized on line 10 or 12. If such an assignment appears in the region matched by "`...`", then the `memset` is performing a reinitialization and should not be removed. This constraint is written as `e = <+... res ...+>` (Figure 5, line 7), to indicate that the value returned by `kmalloc`, `res`, should not appear anywhere on the right-hand side of the assignment. Analogous to this example use, we also add constraints to ensure that the allocated data is not assigned to directly (line 8), or passed to another function (line 9), likely

with the purpose of initializing it. Finally, we forbid loops, as the `memset` may be used to reinitialize the data on each iteration (lines 10-11). Figure 5 shows the resulting more robust semantic patch. On Linux v5.16, this semantic patch makes the same changes as the original one found in Figure 2.

```
1 @@
2 expression res, size, flag, e, f;
3 statement S;
4 @@
5 - res = kmalloc(size, flag);
6 + res = kzalloc(size, flag);
7   ... when != e = <+... res ...+>
8       when != (<+... res ...+>) = e
9       when != f(...,<+... res ...+>,...)
10      when != for(...;...;...) S
11      when != while(...) S
12 - memset(res, 0, size);
```

Fig. 5. A more robust `kmalloc` and `memset` to `kzalloc` semantic patch. Lines 3 and 7-11 are new.

3.3 A second refinement

Our semantic patch requires that the allocated data size be expressed in the same way in both the call to `kmalloc` (first argument) and the call to `memset` (third argument), to ensure that the sizes are the same. However, there are two common ways of indicating data sizes in the Linux kernel: `sizeof(T)`, where `T` is the type referenced by the data pointer, and `sizeof(*x)`, where `x` is the data pointer itself. Figure 6 shows a more flexible semantic patch allowing either style or a mixture.

```
1 @@
2 expression flag, e, f;
3 statement S;
4 type T;
5 T *res;
6 @@
7   res =
8 -         kmalloc
9 +         kzalloc
10          (\(sizeof(T)\|sizeof(*res)\), flag);
11  ... when != e = <+... res ...+>
12      when != (<+... res ...+>) = e
13      when != f(...,<+... res ...+>,...)
14      when != for(...;...;...) S
15      when != while(...) S
16 - memset(res, 0, \(sizeof(T)\|sizeof(*res)\));
```

Fig. 6. A more flexible `kmalloc` and `memset` to `kzalloc` semantic patch. Lines 4-5, 7-10, and 16 are new.

This semantic patch illustrates several new features:

- `-` and `+` need not be applied to complete lines of code (lines 7-10). The matching and transformation process is independent of any whitespace in the semantic patch.
- An expression metavariable can be declared to have a specific type (line 5). This can be a C-language type, or, as illustrated here, a type metavariable.
- A *disjunction*, here written as `\(...\|...\)`, allows specifying a selection of patterns that can be allowed to match. The first match is chosen. A disjunction can also be written as `(...|...)`, where the `(`, `|`, and `)` are in column 0.

This semantic patch finds two more opportunities for `kzalloc`, as compared to the one in Figure 5, however it overlooks two opportunities as well, in which the size is not expressed as a single `sizeof` expression. For greater flexibility, we can create a single semantic patch consisting of Figure 5 followed by Figure 6, to find a larger set of transformation opportunities.

4 A Second Example: `of_node_put`

We next present a case study related to bug finding and fixing. Bug finding and fixing was not the original target of Coccinelle [22], but it can also involve searching for patterns of code and making repetitive changes accordingly, and thus Coccinelle can be useful in this case. While the previous example reorganizes a collection of API calls, this one finds the need for an API call that is missing, in a specific context. This example also illustrates how one instance of a change can be scaled up to many variants.

4.1 The problem

We consider the case of iterators over collections of `device_node` structures. These structures are managed using reference counts. Forgetting to decrement a reference count when needed prevents the structure from ever being freed, causing a memory leak. As a concrete example, we consider the use of the `for_each_child_of_node` iterator. Each iteration visits a `device_node` structure. To simplify the code, this iterator increases the reference count of the current node before executing the body of the loop, and then decreases the reference count of that node before moving on to the next iteration. Figure 7 shows a typical use of the iterator that benefits from these hidden reference count operations.

But, out of sight, out of mind. By hiding the management of the reference count in the normal case, the iterator hides the fact that explicit management of the reference count is needed in exceptional cases. Specifically, in the example of Figure 8, if there is a jump out of the loop body via the `return` (line 7), the increment of the reference count is performed, but the decrement (`of_node_put`), that is performed by the iterator at the end of a loop iteration, is not executed. The solution is to add a call to `of_node_put` (line 6).

```
1 for_each_child_of_node(parent, child)
2   pnv_php_reverse_nodes(child);
```

Fig. 7. A simple use of `for_each_child_of_node`, from `drivers/pci/hotplug/-pnv_php.c`, Linux v5.16.

```
 1  for_each_child_of_node(phandle->parent, node) {
 2    alias_id = of_alias_get_id(node, clk_name);
 3    if (alias_id >= 0 && alias_id < cmdq->gce_num) {
 4      ...
 5      if (IS_ERR(cmdq->clocks[alias_id].clk)) {
 6+       of_node_put(node);
 7        return PTR_ERR(cmdq->clocks[alias_id].clk);
 8      }
 9    }
10  }
```

Fig. 8. A use of `for_each_child_of_node` that may case a memory leak, from `drivers/mailbox/mtk-cmdq-mailbox.c`, Linux v5.16, slightly simplified for conciseness.

The issue occurs not only for jumps via `return`, but also for `goto` and `break`. The jump out of the loop body can occur anywhere within the loop body and there may be multiple such jumps. There is also a large set of relevant iterators.

4.2 The semantic patch

Figure 9 shows the semantic patch for the case of `for_each_child_of_node` and `return`. This semantic patch uses "`...`" (line 9) to trace through each possible execution path in the loop body to find those where the reference count is decremented (line 11), where the `device_node` variable may be stored in some more global way that requires the reference count to remain raised (lines 13-17), and where there is a jump out of a loop (line 20). It is on the latter that an `of_node_put` should be inserted (line 19).

The semantic patch illustrates some more features of SmPL:

- Iterators: Iterators are not part of the C language, but are rather defined by the Linux kernel as macros. While many macros can be parsed as function calls, this is not possible for iterators, because an iterator amounts to a loop header. Accordingly, SmPL provides a special notation for declaring them. `iterator name` (line 2) allows declaring the name of a specific iterator, which is then parsed similarly to a `while` loop. `iterator` (line 5) allows declaring a metavariable that can match any iterator.
- Local variables: `local idexpression` (line 3) declares a metavariable that only matches a variable declared in the current function. This feature is important in this semantic patch, to ensure that the `device_node` does not escape the loop.
- Disjunction: `( | )` in the leftmost column indicates a choice between a selection of patterns. The `?` on the last pattern indicates that the `return` is optional; as in Figure 7, some paths may not match any of the patterns.

```
1 @@
2 iterator name for_each_child_of_node;
3 local idexpression n;
4 expression e,e1;
5 iterator i1;
6 statement S;
7 @@
8  for_each_child_of_node(e,n) {
9     ...
10 (
11    of_node_put(n);
12 |
13    e1 = n
14 |
15    return n;
16 |
17    i1(...,n,...) S
18 |
19 +  of_node_put(n);
20 ?  return ...;
21 )
22    ... when any
23  }
```

Fig. 9. `for_each_child_of_node` with no `of_node_put` before a `return` out of the loop.

– When any: By default, "`...`" matches a path that does not contain a match of any pattern appearing just before or after the "`...`". `when any` allows such matches. The effect of the `when any` on the second "`...`" is that the disjunction pattern matches the first instance of the pattern along each execution path through the loop body.

4.3 Scaling up

In the previous semantic patch rule, the jump out of the loop is performed by a `return`. `goto` and `break` each introduce minor specific issues, and one can create a rule for each case. A second point of variation is the iterator name, and indeed new iterators can be introduced over time. The semantic patch in Figure 10 addresses this issue, for a small selection of iterators, using a pair of rules.

The first rule (lines 1–20), named `r` (line 1), matches the complete loop in two ways, using a conjunction (`&`), analogous to the disjunction introduced previously. The first conjunct lists the names of specific iterators to match, while the second uses metavariables to capture the name of the iterator (`i`) and the number of arguments (`len`) before the `device_node` typed index variable. Note that the position of this index variable varies depending on the iterator.

The second rule (lines 22–44) then *inherits* from rule `r` the metavariables `i` (denoted `r.i`), representing the iterator name, and `len`, representing the offset of the index variable (denoted `r.len`). These inherited metavariables can then be used freely, like any other metavariable.

When applied to a given file, the semantic patch matches the first rule across the file, and collects possible bindings of the set of metavariables. The second rule is triggered once for each unique set of bindings of the metavariables that

```
@r@
local idexpression n;
expression e;
iterator name for_each_child_of_node, for_each_available_child_of_node,
  for_each_node_with_property;
iterator i;
statement S;
expression list [len] es;
@@
(
(
 for_each_child_of_node(e,n) S
|
 for_each_available_child_of_node(e,n) S
|
 for_each_node_with_property(n,e) S
)
&
 i(es,n,...) S
)

@@
local idexpression n;
expression e1;
iterator r.i,i1;
expression list [r.len] es;
statement S;
@@
 i(es,n) {
   ...
(
   of_node_put(n);
|
   e1 = n
|
   return n;
|
   i1(...,n,...) S
|
+  of_node_put(n);
?  return ...;
)
   ... when any
 }
```

Fig. 10. `for_each_child_of_node` with no `of_node_put` before a jump out of the loop.

it inherits. Thus, the second rule will be applied to the entire file up to three times, depending on how many of the iterators mentioned in `r` are used in the file, and thus the number of bindings of rule `r`'s `i` and `len` metavariables.

4.4 Impact

Figure 11 shows the number of files in each release of the Linux kernel between v4.0 (April 2015) and v5.16 (January 2022) that are missing an `of_node_put()` within a use of one of the iterators `for_each_node_by_name`, `for_each_node_by_type`, `for_each_compatible_node`, `for_each_matching_node`, `for_each_matching_node_and_match`, `for_each_child_of_node`, `for_each_available_child_of_node`, or `for_each_node_with_property`. We collected

this information using the `for_each_child.cocci` semantic patch that has been part of the Linux kernel distribution since v5.10 (December 2020).

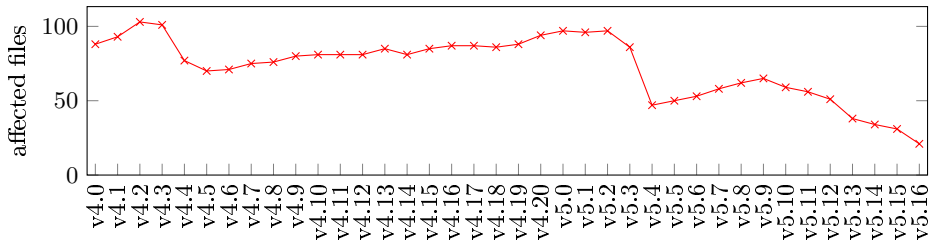

Fig. 11. Number of files missing uses of `of_node_put` as detected by the `for_each_child.cocci` semantic patch found in the Linux kernel.

Over most of the time shown (April 2015 – January 2022), the number of affected files has slowly increased, as, for example, new files have been added that do not contain the required code. The large dips from version v4.3 to version v4.4 and then from version v5.2 to version v5.4 were due in part to the use of Coccinelle to add the needed calls at a large scale. In recent years, there has been a steady decline, starting with Linux v5.10, in which a semantic patch addressing the need for `of_node_put` was added into the Linux kernel. Developers and continuous integration tools can use this semantic patch to add the missing calls even before the code is integrated into a mainline Linux kernel release, breaking the steady upward trend seen in previous releases.

5 A Third Example: Inconsistent Atomicity Flags

Our final example shows how Coccinelle can be used to collect information across a complete code base, and to report anomalies in the collected information as potential bugs. Similar reasoning has been used effectively in various prior approaches for mining API usage rules [4, 8, 17]. We how how this idea can be used in a lightweight way with Coccinelle. A challenge is that Coccinelle works on one file at a time, and within each file on one function (or other top-level declaration) at a time. We show how Coccinelle's scripting language interface, allowing the use of scripts written in OCaml or Python, makes collecting and processing information across an entire code base possible.

5.1 The problem

Our example relates to the use of the Linux kernel flags `GFP_KERNEL` and `GFP_ATOMIC` that are commonly passed to memory allocation functions to indicate whether the function may sleep or not to wait for memory to be available, respectively. Essentially, `GFP_KERNEL` should be used when no lock is held, and

GFP_ATOMIC should be used when a lock is held. The challenge is that holding a lock is an interprocedural property; taking a lock in one function means that the lock is held in the execution of all called functions, until the lock is released.

Detecting whether a caller may hold a lock is particularly difficult for function pointers, which the Linux kernel uses extensively. Figure 12 shows an example, representing an interface to a network device driver. The choice of GFP_KERNEL or GFP_ATOMIC depends on whether locks are held at the call sites of these function pointers. Such call sites are typically located in other files, and thus are not accessible to Coccinelle when processing the file that contains this interface definition and the definitions of the referenced functions. The call sites may be subject to further interprocedural locking effects that are difficult to analyze.

```
1 static struct platform_driver moxart_mac_driver = {
2         .probe  = moxart_mac_probe,
3         .remove = moxart_remove,
4         .driver = {
5                 .name           = "moxart-ethernet",
6                 .of_match_table = moxart_mac_match,
7         },
8 };
```

Fig. 12. Collection of function pointers representing an interface to the MOXA ART Ethernet (RTL8201CP) driver (drivers/net/ethernet/moxa/moxart_ether.c).

5.2 The solution

Rather than search for the function-pointer call sites and the contexts in which they occur, we instead explore what information we can infer by assuming that the function stored in a particular structure member is always called in the same way. This assumption implies that if no locking code is present in the function itself, then either GFP_KERNEL will always be used by all functions stored in a given structure member, or GFP_ATOMIC will always be used. A mixture would imply that either our hypothesis is false, and the function pointer is called in different contexts, or that the function is using an incorrect flag.

The structure of the semantic patch is roughly as follows. First, it will pass over the code base to collect the names of all functions containing a reference to GFP_KERNEL and the names of all functions containing a reference to GFP_ATOMIC. In each case, it identifies the structure member storing the function, if any. Finally, after collecting this information across the entire code base, for each structure member, it compares the number of functions in each category. If there is a large number of functions in one category and a small number of functions in the other, it is possible that inappropriate flags are being used, and the relevant code should be further investigated.

The semantic patch starts as shown below, by defining some hash tables to collect information from across the code base. This rule is indicated as initial-

`ize:ocaml` (line 1), meaning that it is run before the treatment of any files, and that it contains OCaml script code. Such script code is passed directly to the OCaml interpreter, and is not processed by Coccinelle in any way.

```
1 @initialize:ocaml@
2 @@
3 let atbl = Hashtbl.create 101 (* collect functions using GFP_ATOMIC *)
4 let ktbl = Hashtbl.create 101 (* collect functions using GFP_KERNEL *)
```

Next, the semantic patch matches uses of `GFP_KERNEL` and `GFP_ATOMIC`, first identifying a use, then detecting whether the containing function is stored in a structure member, and finally, if so, storing the location of the reference in the appropriate hash table. The rules for each flag are independent, and are thus shown in parallel in Figure 13, although in the actual semantic patch, one sequence of rules comes after the other. The first rule in the `GFP_ATOMIC` case (lines 1-14 on the right of Figure 13) is more complex than the first rule in the `GFP_KERNEL` (lines 1-5 on the left of Figure 13); in the former case we have to ensure that the code is not executed when a lock is locally held, which is verified by ensuring that there is no subsequent lock release before the taking of another lock is optionally reached (lines 8-14), considering some common lock functions.

```
1 @r1@
2 identifier f;
3 position p;
4 @@
5 f@p(...,GFP_KERNEL,...)
6
7 @s1@
8 identifier i,j,fn;
9 identifier f1 :
10  script:ocaml(r1.p)
11   {f1=(List.hd p).current_element};
12 @@
13 struct i j = { .fn = f1, };
14
15 @script:ocaml@
16 p << r1.p;
17 i << s1.i;
18 fn << s1.fn;
19 @@
20 Common.hashadd ktbl (i,fn) p
```

```
1 identifier f;
2 position p;
3 @@
4 f@p(...,GFP_ATOMIC,...)
5 ... when != spin_unlock(...)
6     when != spin_unlock_irqrestore(...)
7     when != spin_unlock_bh(...)
8 (
9 spin_lock(...);
10 |
11 spin_lock_irqsave(...);
12 |
13 ?spin_lock_bh(...);
14 )
15
16 @s2@
17 identifier i,j,fn;
18 identifier f1 :
19  script:ocaml(r2.p)
20   {f1=(List.hd p).current_element};
21 @@
22 struct i j = { .fn = f1, };
23
24 @script:ocaml@
25 p << r2.p;
26 i << s2.i;
27 fn << s2.fn;
28 @@
29 Common.hashadd atbl (i,fn) p
```

Fig. 13. Collection of information about occurrences of `GFP_KERNEL` and `GFP_ATOMIC`.

The semantic patch concludes with a straightforward `finalize:ocaml` rule that iterates over one of the hash tables, and for each structure member compares the number of pointed functions using `GFP_KERNEL` or `GFP_ATOMIC`. The output

can be freely tailored to be more complete, possibly including false positives, or to only include the most likely anomalies, possibly creating false negatives. Among the results, we observe that, in Linux 5.16, 7 functions in the `probe` member of a `platform_driver` structure, as illustrated in Figure 12, use `GFP_ATOMIC`, while 2627 use `GFP_KERNEL`. Checking the 7 cases reveals that they should be converted to use `GFP_KERNEL`. Patches making these changes have been submitted to the Linux kernel, and appear in the `linux-next` version of March 10, 2022.

6 Related Work

Automated program transformation has a long history. We focus on work specifically related to Coccinelle. Lawall and Muller give an overview of the design decisions of Coccinelle, its impact, and closely related work [15]. Martone and Lawall provides a tutorial in using Coccinelle, similar to that presented here, but targeting high-performance computing [19]. Kang *et al.* [9] explore the use of Coccinelle for Java. Outside of the Coccinelle team, Nielsen *et al.* [21] propose a transformation system something like Coccinelle to meet the needs of JavaScript programs. Some Coccinelle-like features have recently been added to the Java source-code analysis and transformation tool Spoon [24].

7 Conclusion

Coccinelle has facilitated thousands of lines of changes in the Linux kernel and other software projects. By making it possible to easily write complex patterns, describing code fragments and their context, Coccinelle enables an alternate, cross cutting view of a large code base. Coccinelle has been a source of fun and pride for its developers. We hope that the reader will have a chance to try Coccinelle, and will enjoy using it too.

Availability: Coccinelle is available from many Linux distributions, and from the Coccinelle website: `https://coccinelle.gitlabpages.inria.fr/website/`

Acknowledgments: Yoann Padioleau and René Rydhof Hansen were postdocs working on Coccinelle in its earliest days, and contributed greatly to the design and implementation. Nicolas Palix has also maintained parts of Coccinelle over the years. Recent interns who contributed greatly to the code base include Jaskaran Singh and Keisuke Nishimura. The initial work on Coccinelle was funded in part by the French ANR and the Danish FTP. Recently, Inria has supported the continued maintenance of Coccinelle, with the help of Sébastien Hinderer and then Thierry Martinez. We are also deeply grateful for the feedback and support from the Linux kernel developer community. Keisuke Nishimura and Michele Martone also gave helpful feedback on drafts of this paper. We thank the organizers of NFM22 for the invitation to present this work.

References

1. Brunel, J., Doligez, D., Hansen, R.R., Lawall, J., Muller, G.: A foundation for flow-based program matching using temporal logic and model checking. In: POPL. pp. 114–126 (Jan 2009)
2. Casazza, G., Villano, U., Merlo, E., Antoniol, G., DiPenta, M.: Identifying clones in the Linux kernel. In: Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation (2001)
3. Eclipse (2022), https://www.eclipse.org/ide/
4. Engler, D.R., Chen, D.Y., Chou, A.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Marzullo, K., Satyanarayanan, M. (eds.) SOSP. pp. 57–72. ACM (2001)
5. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2002)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
7. Git (Sep 2021), https://github.com/git/git/tree/master/contrib/coccinelle
8. Goues, C.L., Weimer, W.: Specification mining with few false positives. In: Kowalewski, S., Philippou, A. (eds.) TACAS. Lecture Notes in Computer Science, vol. 5505, pp. 292–306. Springer (2009)
9. Kang, H.J., Thung, F., Lawall, J., Muller, G., Jiang, L., Lo, D.: Semantic patches for Java program transformation (experience report). In: ECOOP. LIPIcs, vol. 134, pp. 22:1–22:27 (2019)
10. Kernighan, B.: UNIX: A History and a Memoir. Kindle Direct Publishing (2019)
11. Kernighan, B.W., Pike, R.: The UNIX Programming Environment. Prentice Hall (1984)
12. Lawall, J.: An introduction to Coccinelle bug finding and code evolution for the Linux kernel. Suse Labs (2014), https://www.youtube.com/watch?v=buZrNd6XkEw
13. Lawall, J.: Keynote: Inside the mind of a coccinelle programmer. Linux Security Summit (2016), https://www.youtube.com/watch?v=xA5FBvuCvMs
14. Lawall, J.: Coccinelle: 10 years of automated evolution in the Linux kernel. Linaro Connect (2019), https://www.youtube.com/watch?v=LOsluYTzdMg
15. Lawall, J., Muller, G.: Coccinelle: 10 years of automated evolution in the Linux kernel. In: USENIX ATC. pp. 601–614 (2018)
16. Lawall, J.L., Brunel, J., Palix, N., Hansen, R.R., Stuart, H., Muller, G.: WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process. Software: Practice and Experience **43**(1), 67–92 (Jan 2013)
17. Li, Z., Zhou, Y.: PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In: ESEC-FSE (2005)
18. MacKenzie, D., Eggert, P., Stallman, R.: Comparing and Merging Files With Gnu Diff and Patch. Network Theory Ltd (Jan 2003), Unified Format section, http://www.gnu.org/software/diffutils/manual/ html_node/Unified-Format.html
19. Martone, M., Lawall, J.: Refactoring for performance with semantic patching: Case study with recipes. In: High Performance Computing - ISC High Performance Digital 2021 International Workshops. Lecture Notes in Computer Science, vol. 12761, pp. 226–232 (2021)
20. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Compiler Construction. Lecture Notes in Computer Science, vol. 2304, pp. 213–228 (2002)

21. Nielsen, B.B., Torp, M.T., Møller, A.: Semantic patches for adaptation of JavaScript programs to evolving libraries. In: ICSE. pp. 74–85. IEEE (2021)
22. Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in Linux device drivers. In: EuroSys 2008. pp. 247–260. ACM, Glasgow, Scotland (Mar 2008)
23. Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., Muller, G.: Faults in Linux 2.6. ACM Transactions on Computer Systems **32**(2), 4:1–4:40 (Jun 2014)
24. Spoon (Mar 2022), https://github.com/INRIA/spoon
25. Stefaniuc, M.: Coccinelle scripts for Wine (Sep 2021), https://github.com/mstefani/coccinelle-wine
26. Systemd (Feb 2022), https://github.com/systemd/systemd/tree/main/coccinelle
27. WineHQ: Static analysis (Feb 2016), https://wiki.winehq.org/Static_Analysis