HAL
archives-ouvertes.fr

Coccinelle: 10 Years of Automated Evolution in the Linux Kernel

Julia Lawall, Gilles Muller

▶ To cite this version:

Julia Lawall, Gilles Muller. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. 2018 USENIX Annual Technical Conference, Jul 2018, Boston, MA, United States. hal-01853271

HAL Id: hal-01853271
https://hal.inria.fr/hal-01853271

Submitted on 2 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Coccinelle: 10 Years of Automated Evolution in the Linux Kernel

Julia Lawall and Gilles Muller, ***Sorbonne University/Inria/LIP6***

https://www.usenix.org/conference/atc18/presentation/lawall

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Coccinelle: 10 Years of Automated Evolution in the Linux Kernel

Julia Lawall
Sorbonne University/Inria/LIP6

Gilles Muller
Sorbonne University/Inria/LIP6

Abstract

The Coccinelle C-program matching and transformation tool was first released in 2008 to facilitate specification and automation in the evolution of Linux kernel code. The novel contribution of Coccinelle was that it allows software developers to write code manipulation rules in terms of the code structure itself, via a generalization of the patch syntax. Over the years, Coccinelle has been extensively used in Linux kernel development, resulting in over 6000 commits to the Linux kernel, and has found its place as part of the Linux kernel development process. This paper studies the impact of Coccinelle on Linux kernel development and the features of Coccinelle that have made it possible. It provides guidance on how other research-based tools can achieve practical impact in the open-source development community.

1 Introduction

Today, everyone uses the Linux kernel, whether on a mobile phone (86% of the smartphones sold in the first quarter of 2017 were running Android [22]), in the cloud (at the end of 2016, 92% of virtual machine instances on Amazon's Elastic Compute Cloud (EC2) were running Linux), or on a supercomputer (at the end of 2017, all of the top 500 supercomputers were running Linux [63]). To support these diverse computing environments, the size of the Linux kernel has been steadily growing, reaching 16.5 million lines of code in the recently released version 4.15 (Jan. 2018). Furthermore, the existing source code of the Linux kernel is continually changing, with around 13,000 commits per release recently, to improve security, performance or maintainability, as well as to provide support for new services such as new kinds of devices, file systems, or hardware architectures.

A stumbling block in this continual revision of the Linux kernel is that ultimately some developer has to modify the source code. Developers have limited time, may not fully understand what a given change entails, and are prone to making mistakes, particularly when changes affect many code sites, pervasively, across multiple kernel subsystems. These problems are further compounded by the fact that the Linux kernel has a widely dispersed and very diverse set of developers, ranging from core maintainers, with many years of experience, to occasional contributors, to developers of out-of-tree code who do not participate in the Linux kernel developer community. Indeed, while over 1700 developers have contributed to each recent release, in each case a third or more of these developers have contributed only a single patch to that release. A potential solution is to formally specify changes and automate them. To be used in practice, such an approach must fit with the background and habits of the developers themselves.

The Coccinelle C-program matching and transformation tool was first released in 2008 to facilitate specification and automation of the evolution of Linux kernel code [44]. Coccinelle was built around the observation that Linux kernel developers already have a precise notation for describing changes with which they are very familiar, the *patch* [36]. A patch is an extract of source code in which some lines are annotated with - or + indicating that the line should be removed or added, respectively. All contributions to the Linux kernel pass in the form of patches through mailing lists where they are commented on by other developers, and thus developers are used to seeing and understanding them. Exploiting this background of kernel developers, Coccinelle is designed around a domain-specific language (DSL), SmPL (Semantic Patch Language), for expressing changes in terms of an abstracted form of patch, referred to as a *semantic patch*. Unlike a patch, which is tied to specific lines and files in the source code, a single semantic patch can update all relevant locations in the entire code base.

Today, Coccinelle has been under development for 12 years. 59 semantic patches are part of the Linux kernel source tree, and over 6000 Linux kernel commits, includ-

ing 900 from Linux kernel maintainers, use Coccinelle. Coccinelle retains as a guiding principle the notion of an abstracted patch. Nevertheless, it has grown, both in terms of expressiveness and to improve performance, based on lessons learned from the experiences of Linux developers and other users. At the same time, it has successfully integrated into the Linux kernel community.

Coccinelle has been used in previous research [30, 31, 34, 46]. This paper instead focuses on its evolution and impact. We examine its initial design (Sec. 2), and how that design has been refined in response to experience with the tool and feedback from users (Sec. 3). We then evaluate performance (Sec. 4) and the benefit of our expressivity extensions (Sec. 5), quantify the impact of Coccinelle in the Linux community (Sec. 6), and give an overview of its wider use (Sec. 7). Finally, we present some related work (Sec. 8) and conclude (Sec. 9), with lessons learned about dissemination of a research tool.

2 Initial Design of Coccinelle

Coccinelle development began in 2006. It was first made publicly available in binary in 2007 and in open source in 2008. We first review the original design decisions for Coccinelle, in terms of goals, expressivity, performance, correctness guarantees, and dissemination.

2.1 Goals

Coccinelle was initially designed to solve a specific problem, that of porting Linux device drivers from Linux 2.4, a previous stable version, to stable version Linux 2.6, which had been released shortly before the start of the project. The initial design was motivated by an earlier paper on *collateral evolutions* in the Linux kernel [45], *i.e.*, evolutions needed in API clients in response to changes in the API interface. The examples from that paper showed that to automate Linux kernel collateral evolutions it would be necessary to support transformations on scattered parts of the source code with various kinds of connections between them, including intraprocedural control-flow paths with specific properties. As a small research project could not encode the entire porting activity, these kinds of connections, derived from program-analysis concepts, would need to be expressed in a way that would be accessible to Linux driver developers, who could carry on the work. Targeting driver developers furthermore implied that Coccinelle would have to allow the user to reason about the code as it is shown to him, without simplification to an internal representation, and that it would have to treat a very large subset of C constructs, including various gcc extensions, according to the needs of arbitrary Linux kernel device driver code. Finally, the generated code would have to retain the structure of the original source code, including comments and whitespace, to ensure the code's continuing maintainability.

```
1 @ rule1 @
2 identifier fn, irq, dev_id;
3 typedef irqreturn_t;
4 @@
5 static irqreturn_t
6 fn(int irq, void *dev_id)
7 { ... }
8 @@
9 identifier rule1.fn;
10 expression E1, E2, E3;
11 @@
12  fn(E1, E2
13 -    ,E3
14     )
```

Figure 1: The first semantic patch submitted to Linux

2.2 Design decisions affecting expressivity

Coccinelle provides a transformation language SmPL (Semantic Patch Language) and an engine for applying SmPL semantic patches to C code. SmPL was conceived as a code pattern-matching language, mimicking the patch syntax. A SmPL semantic patch consists of a series of rules, analogous to patch hunks, each providing a code pattern to match or transform. Patterns are comprised of concrete syntax, "...", and metavariables. Concrete syntax matches itself, "..." matches a possibly empty sequence of arbitrary terms, *e.g.*, the list of statements between two other statements, and metavariables match arbitrary terms of a particular type. Metavariables are declared in a rule header and are used as ordinary variables in the pattern, to make the patterns close to the source code. Ideally, a Linux kernel developer should be able to copy a typical code example and add metavariable declarations, "...", and - and + annotations, to obtain a transformation rule with minimal effort.

The semantic patch in Figure 1 illustrates the various features of SmPL. This semantic patch completes an evolution and associated collateral evolutions that had been initiated by a Linux developer. The evolution changed the type of a callback function, by removing its third parameter. This required additionally removing the third argument from direct calls to this function. This change is challenging because the names of the affected functions are all different, implying that `grep` may not be sufficient to find all occurrences. A common strategy is to identify code to fix by compiler errors, iterating until the kernel compiles successfully. In this case, some of the direct calls had been overlooked due to being under ifdefs or in the support for obscure architectures.

The semantic patch consists of two rules, on lines 1-7 and lines 8-14, respectively. The first rule, named `rule1`, declares three identifier metavariables `fn`, `irq` and `dev_id`, representing the name of the function to match and the names of its two parameters, respectively. The rest of the first rule is a pattern that matches a function definition, in which the parameters and return value are indicated to have specific types (lines 5-6) and the

body is allowed to be an arbitrary sequence of statements (line 7). The second rule, which has no name, declares four metavariables: the function name `fn`, whose value is explicitly inherited from the previous rule (line 9), and three expression metavariables, `E1`, `E2`, and `E3`, representing arbitrary argument expressions (line 10). The rest of this rule matches a call to the function that was identified in the first rule. In this call, the third argument is indicated to be removed (line 13). A wider variety of semantic patches is illustrated in various publications [31, 43, 50] and at the Coccinelle website [10, 11].

To apply a semantic patch to a code base, Coccinelle processes the C source code files one at a time. On each file, it applies the first rule of the semantic patch to each function or other top-level declaration, then applies the second rule to the code resulting from the first rule application, etc. Based on the needs observed in the prior collateral evolution study, the processing of a function is based on its intraprocedural control-flow graph. Thus, at the statement level, *e.g.*, line 7 of Figure 1, "..." follows intraprocedural control-flow paths, using a semantics based on a variant of CTL model checking [5]. By default, a pattern must match all control-flow paths starting from the control-flow graph node matching the beginning of the pattern, to ensure that the semantic patch describes a consistent view of the program behavior. For example, when a pattern such as `A(); ... B();` matches code including a conditional, `B();` must be reachable from `A();` via both branches of the conditional. Alternatively, a rule or an individual instance of "..." can be annotated with `exists` to indicate that only the existence of a matching path is required. By default, "..." cannot contain any code that is matched by the code pattern immediately preceding or following it, *e.g.*, to allow matching a call to a locking function and to the unlocking call closest to it, as needed due to the fine-grained locking found in the Linux kernel. Finally, a metavariable must match identical terms within a single control-flow path, but may match different terms in different control-flow paths, *e.g.*, different conditional branches [5].

2.3 Design decisions affecting performance

Coccinelle is intended to be used by a Linux developer in the course of his ordinary work, whenever a recurring transformation is needed. Accordingly, it must be usable on a standard laptop without much disruption. A number of the initial design decisions were guided by this goal.

The Linux kernel is very large, and indeed has more than tripled in size between Feb. 2007 (version 2.6.20, 5M LOC) and Jan. 2018 (version 4.15, 16.5M LOC). Processing the entire code base and achieving reasonable performance on a developer's laptop, thus requires making some tradeoffs. To reduce running time, Coccinelle focuses on regions of code that are most likely to be relevant for collateral evolutions, at the expense of the rest. A key observation is that an individual Linux kernel file typically addresses a problem at a given level of abstraction, while references to other files, via `#include` or function calls, typically move to a lower level of abstraction. Thus, the contents of header files and called functions may be less relevant for collateral evolutions.

Based on the above observation, by default, Coccinelle processes only `.c` files, includes only header files that are located in the same directory as the `.c` file or that have the same name as the `.c` file, and does not perform interprocedural analysis. Command-line options are provided to additionally process header files, independently of any files into which they may be included, and to include header files directly referenced in a `.c` file or all header files referenced recursively. The latter options, however, increase the amount of code processed, and thus the processing time. The use of these strategies is thus left up to the user, who is expected to know whether such information is relevant to the desired evolution. Finally, interprocedural analysis within a single file can be encoded up to a finite depth using a series of SmPL rules, each of which matches the definition of a function for which a function call was identified by a previous rule. More general interprocedural analysis originally required the use of external scripts to collect the names of called functions and to restart Coccinelle to process their definitions.

The only program analysis performed by Coccinelle is type inference. This analysis is best-effort, as non-inclusion of header files means that type information may be unavailable. Although the type information is incomplete, the inclusion of type information makes Coccinelle very useful for tasks such as finding where a field of a particular type of structure is referenced, without knowing the name of the variable pointing to that structure. Coccinelle performs no alias analysis or other form of dataflow analysis. Semantic patches that require control over aliases have to implement it explicitly, *e.g.*, by declaring that the code region matched by "..." cannot store the address of a given variable. This approach saves execution time, as the analysis is only performed if and to the extent that it is expected to be useful, and improves the predictability of the tool, as the semantic patch writer knows the strengths and limitations of the analysis.

2.4 Design decisions affecting correctness guarantees

Automatic program transformation has the potential to update code at a large scale reliably and efficiently, but it can also introduce pervasive bugs across a code base, if the transformation rules are incorrect or are implemented incorrectly by the transformation engine. Coc-

cinelle only checks that a rule preserves the structural well formedness of the code, *e.g.*, ensuring that a statement is replaced by a statement, an expression by an expression, etc. It does not check for semantic correctness. This enables encoding bug fixes, which are intrinsically not semantics preserving. Furthermore, it enables efficiently applying rules, without complicated, typically interprocedural, analysis to show correctness. The goal of Coccinelle is to allow the user to express his knowledge about the software and the required changes, in terms of code fragments that resemble the affected code and that can be easily checked to conform to the user's intent.

2.5 Dissemination strategy

Reaching potential users is always a challenge for research projects. An open-source development context provides a diverse audience, which increases the chance that individual users will pick up new approaches, but makes it harder to impose a new approach on the entire developer base than in a monolithic industry setting. This is particularly the case of the Linux kernel developer community, which puts few restrictions on the tools used by developers to create and manage code.

To validate the utility of Coccinelle and to encourage its use by Linux developers, the Coccinelle developers took the strategy of showing by example. The first submitted patches (*e.g.*, 632155e65944 on June 1, 2007 [61]) exploited only the Coccinelle parser [42]. Indeed, as Coccinelle does not expand macros or reduce ifdefs, its parser can find errors that are overlooked in typical compile testing. The first submitted patch generated by a semantic patch was 0da2f0f164f0 (July 5, 2007). This patch was created using the semantic patch of Figure 1, and included the semantic patch in the commit log. It updated five files in the `net`, `atm`, and `usb` directories.

The next patch dd00cc486ab1 based on a semantic patch was submitted on July 6, 2007, changing a call to the memory allocator `kmalloc` followed by a `memset` to clear the memory into a single call to the function `kzalloc`, added in 2005. This patch affected 166 calls distributed over 146 files. The semantic patch in the log message received the comment "Cool!" from a developer,[1] but the patch ran afoul of the Linux kernel requirement that patches on different parts of the kernel be submitted separately to the relevant maintainers. Indeed, Coccinelle had shifted the burden from performing the change, which was now fully automated, to routing the individual changes to the proper maintainers, for which no automatic support was then available.

The first patches explicitly mentioning "Coccinelle" were submitted in December 2007, fixing various missing resource-release errors (76832d841643, etc.). These attempted to set a precedent for how the tool should be used, by including the URL of the tool, as well as the XML-like tags `<smpl>` and `</smpl>` around the semantic patch, to ease the tracking of the use of the tool. The first commits from outside the group of Coccinelle developers, 77bbadd5ea89 and 52fd8ca6ad41, came in July 2008, from the developer of the kernel-level memory checker `kmemcheck`. These patches corrected the type of a flag passed to various lock-related function calls. Coccinelle was released as open source in October 2008.

[1] https://lkml.org/lkml/2007/7/7/98

3 Evolutions in the Coccinelle Design

In retrospect, the design decisions presented in the previous section reflect a number of fundamental hypotheses about how to design a program transformation system that will be useful to and used by kernel developers:

Expressivity: Linux kernel developers will find it easy and convenient to describe needed code changes in terms of fragments of removed and added code.

Performance: Many interesting Linux kernel evolutions can be implemented reliably without incurring the cost of collecting and correlating information in multiple C files. Indeed, Linux kernel development relies on humans, who typically focus on one file at a time, and thus all relevant information should be directly apparent in a single C file.

Correctness: Proving correctness is not necessary because Linux kernel developers can easily incorporate their knowledge of kernel invariants into a semantic patch. Giving the developer control over the rules implies that the developer can control the rate of false positives, and can easily check for them in the results.

Dissemination: It is effective to show how a tool can be useful, rather than attempting to impose its use.

While Coccinelle still builds on these hypotheses, experience in using the tool and feedback from Linux kernel developers have provided lessons that have motivated various evolutions. We highlight those that have had the greatest impact on the use and usability of the tool.

3.1 Expressivity evolutions

Many changes, both simple and complex (*e.g.*, Figure 1), can be expressed purely in terms of code structure. Some kinds of changes, however, require more semantic information. Two evolutions that have greatly enhanced the expressivity of Coccinelle have been the introduction of *position variables* and *scripting rules*.

Position variables. A position variable is a Coccinelle metavariable that matches the position where a term occurs in a file. Position variables allow rematching the

```
@ rule1 @
expression t, f, e;
position p1, p2;
@@
init_timer@p1(&t);
... when != f = e
t.function =@p2 f;

@ rule2 @
expression rule1.t, d, e;
position rule1.p1, p3;
@@
init_timer@p1(&t);
... when != d = e
t.data =@p3 d;
@@
expression rule1.t, rule1.f,
  rule2.d;
position rule1.p1, rule1.p2,
  rule2.p3;
@@
(
- init_timer@p1(&t);
+ setup_timer(&t,f,d);
|
- t.function =@p2 f;
|
- t.data =@p3 d;
)
```

Figure 2: `init_timer` conversion

```
@r@
expression E; statement S;
position p1,p2;
@@
if@p1 (E);
  S@p2

@script:python@
p1 << r.p1; p2 << r.p2;
@@
if (p1[0].col >= p2[0].col):
  cocci.include_match(False)
@@
expression E; statement S;
position r.p1;
@@
if@p1 (E)
- ;
S
```

Figure 3: Drop spurious semicolon after if header

same code in a later rule, as well as ensuring that a match in one rule is different than a match in an earlier rule.

Figure 2 illustrates the use of position variables to convert calls to `init_timer` to `setup_timer` when the `init_timer` call is followed by initializations of the timer `data` and `function` fields. The first two rules (lines 1-15) identify instances of the same `init_timer` call with two different properties (the `when` annotations in lines 6 and 14 indicate assignments that should not occur in the matched region). The last rule (lines 16-29) transforms `init_timer` calls that satisfy both properties. This rule includes a disjunction, such that all of the relevant code fragments can be transformed at once, wherever they occur, as once a transformation takes place, all previously bound position variables are invalidated.

Scripting language interface. The scripting language interface was initially motivated by the goal of using Coccinelle for bug finding [56]. While bugs that mainly depend on the code structure, such as use after free, could be found, the pattern-matching features of Coccinelle were not sufficient to detect bugs such as buffer overflows that require reasoning about variable values.

To allow reasoning about arbitrary information, support was added in 2008 for scripting-language rules. The first language supported was Python, which was expected to be familiar to Linux developers. Coccinelle is implemented in OCaml, and OCaml scripting was added in 2010, for the convenience of the Coccinelle developers. Scripts were originally designed to filter sets of metavariable bindings established by previous rules. Figure 3 shows an example, which drops a semicolon after an `if` header if the subsequent statement is indented, suggesting that the latter statement is intended to be the `if` branch. A script rule compares the indentation of the two statements (line 11) and discards metavariable binding environments (line 12) in which the conditional is aligned with or to the right of the subsequent statement.

Ultimately, the original motivation for scripting, *i.e.*, finding bugs such as buffer overflows, was not successful. The code patterns were small and generic, and the scripts implementing the required analyses were complex. Still, scripting has been a major leap forward for the expressiveness of Coccinelle, and new scripting functionalities have been added as new needs have emerged. Early on, libraries were added for generating formatted error messages. In 2009, `initialize` and `finalize` scripts were introduced to allow defining state global to the processing of all files, to facilitate the collection of statistics. In 2010, scripts became able to create new code fragments to be stored in metavariables and inherited by subsequent rules. In 2016, to improve performance and reduce semantic patch size, it became possible to add script code to metavariable declarations, to define predicates that would discard metavariable bindings early in the matching process. Finally, scripting enables *iteration*, which allows a semantic patch to submit new "jobs" to the Coccinelle engine, in order to perform analysis across multiple files.

3.2 Performance evolutions

While avoiding including header files reduces the volume of code to process, the Linux kernel remains a large and growing code base. Furthermore, parsing the code without relevant macro definitions from header files involves using heuristics, which can increase the parsing time. Thus, further optimizations were needed.

Indexing. An early observation was that performance could be improved by not parsing files that could not be matched by the semantic patch. Indeed, many semantic patches contain keywords such as the names of API functions that must be present for the semantic patch to match and that occur only a moderate number of times in the Linux kernel. Coccinelle initially used the Unix command `grep` to find the files containing these keywords, but this was still slow, given the large code size.

A second approach was to use `glimpse` [24] to prepare an index in advance, and then to only process the files indicated by the index. As the index is smaller than the kernel source code and is organized efficiently,

the use of `glimpse` substantially improves performance, particularly for semantic patches that involve kernel API functions. Nevertheless, `glimpse` was originally only freely available to academic users, had to be manually installed, and creating an index on each kernel update is time-consuming. In 2010, this was complemented by support for id-utils, which is part of many Linux distributions and for which index creation is much faster. In 2013, the support for users who do not have an index available was rewritten to essentially reimplement the `grep` operation in OCaml, reducing system calls and better taking into account the specific needs of Coccinelle.

Parallelism. By default Coccinelle works on each `.c` file independently, and thus is ripe for parallelism. Nevertheless, when Coccinelle was first developed, there was no convenient support for parallelism in OCaml. Instead, the Coccinelle distribution included a shell script to launch multiple Coccinelle instances in parallel, each covering a different part of the code base. Users, however, were uneasy about using Coccinelle via an external script. Furthermore, processing different files can require very different amounts of time, and the lack of load balancing in this static solution meant that many cores could end up idle. Meanwhile, the `Parmap` OCaml parallelization library [14] became available, and between 2015 and 2017 increasing support was provided for parallelism, still at the `.c` file level, within Coccinelle itself.

Supporting finer grained parallelism, at the function level, was also considered. Initial experiments, however, suggested that the cost of passing around the state built up within the matching of a given file outweighed the benefits of parallelism. In contrast, Coccinelle treats each file independently, so the amount of state that needs to be passed between processes is minimal.

3.3 Correctness guarantee evolutions

Unlike the other cases, there have been no major evolutions in the view of transformation correctness. After having created over 450 semantic patches that have led to kernel patches, the Coccinelle developers have found that the original hypothesis that giving the developer control over the rules enables them to easily check the results is mostly sufficient. The few errors, *e.g.*, [53], have come from misunderstanding of kernel invariants that would require a prohibitively complex and time consuming semantic analysis to infer and check. Kernel maintainers have indeed concluded in some cases that it was the original code that was written in an error prone way [38].

3.4 Dissemination strategy evolutions

Showing the value and capabilities of Coccinelle by the example of submitted patches generated initial interest in the tool. As the expressivity of Coccinelle evolved to permit the specification of more complex changes, it became apparent that it would also be beneficial to more directly teach developers how to use Coccinelle, and to enable Coccinelle users to interact with each other.

Four workshops were organized on the use of Coccinelle and advertised on the Coccinelle mailing list, attracting industry participants. The Coccinelle developers also presented the tool and offered tutorials in a variety of developer conferences, including those targeting open-source enthusiasts (*e.g.*, FOSDEM) and those specifically targeting Linux kernel developers (*e.g.*, Linux Plumbers). These presentations focused on the user-visible aspects of Coccinelle, such as how to write semantic patches and what results could be achieved, rather than the details of the internal design of the system, which were presented in research venues [5].

The work on Coccinelle was also picked up by the Linux Weekly News (LWN), which is the standard reference for issues around the development of the Linux kernel and other open-source software. Tutorial articles on Coccinelle appeared in 2009 [25] and 2010 [52], authored not by the Coccinelle developers, but by well-known kernel developers. LWN has also reported on various talks about Coccinelle [12, 16].

4 Performance Evaluation

Coccinelle is intended to be used by a kernel developer as part of the normal development process, on a standard professional laptop. Accordingly, Coccinelle's performance should be acceptable in this setting. We illustrate the performance on a Lenovo Thinkpad T460s with two hyperthreaded 2.30GHz cores (Intel(R) Core(TM) i5-6200U CPU), a 3M cache, and 12G RAM. Our experiments focus on the 59 semantic patches found in the Linux 4.15 kernel, using the report mode, which is supported by all these semantic patches.[2] Times are based on a single run, with a timeout of 30 seconds per file. We use id-utils indexing. Figure 4 presents the elapsed time when running the semantic patches on the Thinkpad laptop, using both cores, with hyperthreading. The semantic patches are sorted in order of increasing running time.

For the Linux kernel, there is a precise performance point of reference that is familiar to the kernel developer; the time to perform a complete compile of the Linux kernel itself. The elapsed time for full kernel compilation on the Thinkpad laptop with 4 threads (hyperthreading) with `make clean; make allyesconfig; make`, is 54 minutes. Based on the results shown in Figure 4,

[2] The semantic patches and Coccinelle version used contain some performance improvements that will appear in Linux 4.18 and Coccinelle 1.0.7, respectively.

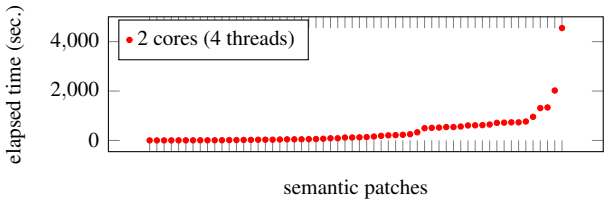

Figure 4: Elapsed time per semantic patch in the Linux 4.15 kernel on the Thinkpad laptop

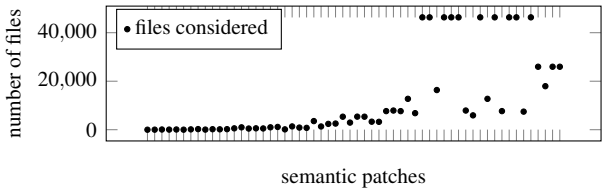

Figure 5: Files considered per Linux semantic patch.

all but one of the semantic patches complete within this time. The remaining semantic patch requires 75 minutes.

The timeout per file affects performance. In our experiment, Coccinelle timed out on 52 files, of 705,179 files considered, giving a timeout rate of 0.007%. Typically files on which timeouts occur contain complex code such as long sequences of loops and conditionals. These files can be analyzed separately, when more time is available.

Figure 5 shows the impact of indexing using id-utils on the number of files considered. The largest number of files considered is 46,336, in 10 cases, where no keywords are inferred from the semantic patch. These are associated with larger, but not the largest, execution times. At the far right of the graph, between 5000 and 26,000 files are considered, but the cost of tracing through all possible intraprocedural execution paths overwhelms the savings obtained by processing fewer files.

In terms of header files, 43 of the semantic patches specify that no include files should be considered. 11 specify to use the default (local and same-named files), and 1 specifies that all explicitly included should be taken into account. To assess the cost and benefit of including header files, we take the 44 semantic patches from the Linux kernel that complete in our test configuration in under 10 minutes and test them with options forcing the inclusion of no header files, the default, and all explicitly included header files. As compared to inclusion of no header files, the default increases the run time by up to 90% and the inclusion of all explicitly included header files increases the run time by up to 10x. The number of reports ranges from 1631 for no headers to 1691 for all headers, with most of the few differences on `.h` files.

The performance studied here is only relevant when scanning the entire kernel. When checking a single modified file, the time should rarely exceed a few seconds per semantic patch. Indexing may identify some semantic patches as irrelevant, reducing the execution time.

5 Expressivity Extension Evaluation

The position variable and scripting extensions increase the expressivity of SmPL, but add concepts that are not found in C code and thus are not already familiar to Linux developers. We thus assess the degree to which these features are used in practice. We note, however, that our only source of information about semantic patches is from those found in the Linux kernel and from those included in commit messages. This information may be incomplete, because developers can omit or simplify semantic patches in the commit message

All of the semantic patches found in the Linux kernel use positions and scripts in order to generate output in the report mode. 20 semantic patches were contributed by developers from outside the Coccinelle team. 3325 commits up through Linux 4.15 contain semantic patches in the commit message. Of these 586 (18%) contain position variables and 165 (5%) contain scripts. 43% of the latter commits come from outside the Coccinelle team.

6 Impact on Linux

Over the past 10 years, Coccinelle has been increasingly applied to the Linux kernel, by both Coccinelle developers and Linux kernel developers. As of Linux 4.15, over 6000 commits in the Linux kernel are based on the use of Coccinelle. In this section, we give an overview of the impact of Coccinelle on the Linux kernel. Graphs by subsystem reflect commits up through the release of Linux 4.15 (Jan. 2018). Graphs by year end with 2017.

6.1 Changed lines per subsystem

Figure 6 shows the number of lines removed and added by commits using Coccinelle in various kernel subsystems. The most affected is `drivers`, with 57,882 removed lines and 77,306 added lines, followed by `arch`, `fs`, `net`, `sound`, and `include`, all of which are affected by thousands of removed or added lines. The predominance of `drivers` is not surprising, given that `drivers` makes up 67% of the Linux 4.15 kernel source code. `drivers` has also been a target for other bug finding and code reliability tools [35, 37, 48, 51, 57].

Figure 7 compares the numbers of removed and added lines to the number of code lines (non-blank, non-comment, measured using SLOCCount [54]) found in Linux 4.15. The rate of Coccinelle-motivated changed

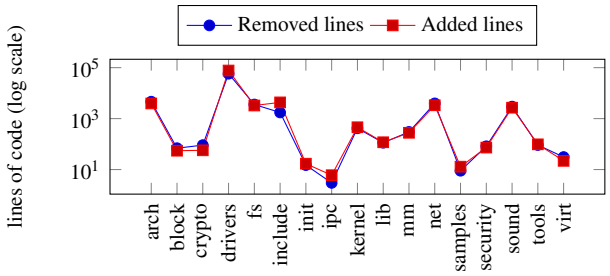

Figure 6: Number of lines removed and added by commits using Coccinelle, by subsystem

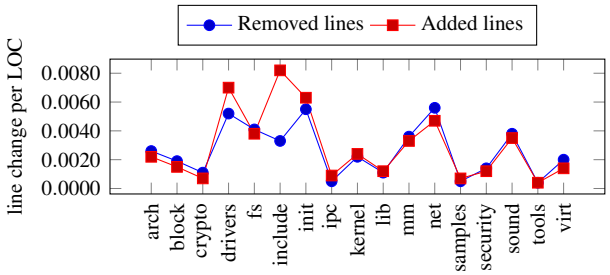

Figure 7: Lines removed and added by commits using Coccinelle per Linux 4.15 line of code, by subsystem

lines in `drivers` remains high, but the results show the applicability of Coccinelle across the kernel.

6.2 Categories of users over time

A variety of kinds of developers contribute to the Linux kernel, by submitting patches. Among those who mention Coccinelle in their commit logs, we distinguish six categories of Coccinelle users:

Coccinelle developers. These are members of the Coccinelle development team, and persons employed by the team to disseminate Coccinelle.

Outreachy interns. The Linux kernel participates in the Outreachy internship program [41] and interns may use Coccinelle in the application process or the internship.

Dedicated user. This is a single developer who uses Coccinelle in the kernel for a small collection of widely relevant simple changes.

0-day. This is an automated testing service at Intel that builds and boots the Linux kernel for multiple kernel configurations, on each commit to hundreds of git trees. The service also runs a number of static analysis tools, including Coccinelle, on the result of each commit.

Kernel maintainers. These are kernel developers who receive and commit patches, and are generally responsible for some subsystem's continued well being. We identify maintainers as developers who are named in the

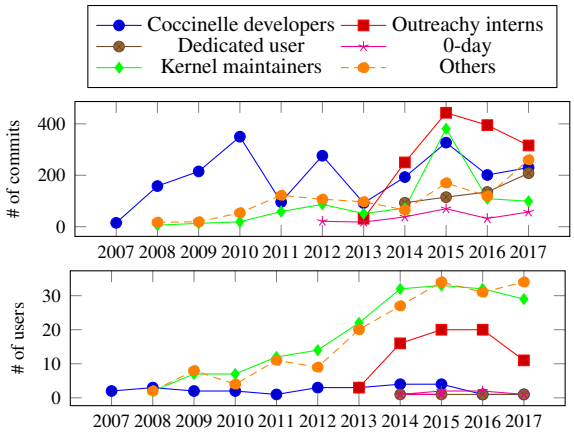

Figure 8: Number of commits using Coccinelle from various categories of Coccinelle users (top) and number of Coccinelle users in various categories having at least one commit using Coccinelle (bottom)

Linux 4.15 MAINTAINERS file (1170 developers) or who are the committer of some patch. Normally, kernel patches are transmitted by email, and only maintainers or developers the maintainers specifically designate commit to git trees that are pulled into a mainline release. Thus, being a committer is a sign of community respect.

Others. These are other Linux kernel contributors. These contributors may be frequent or occasional.

The top of Figure 8 shows the number of commits per year using Coccinelle from various kinds of Coccinelle users, while the bottom shows the number of Coccinelle users involved in each category. The first commits (2007) were from the Coccinelle development team. Use from maintainers and other kernel contributors started in the two years afterwards. The number of commits from maintainers has grown steadily, except for a major peak in 2015, when several maintainers undertook cross-tree refactoring projects using Coccinelle. The number of other kernel contributors has gone up and down, but shows an upward trend. These numbers may be underestimated, however, as some developers have revealed when asked that they used Coccinelle for repetitive changes, but did not mention it in the commit.[3]

The Linux developers who are most likely to have need for Coccinelle are those who perform large scale changes across the code base. To approximate this set of developers, we consider those who have at least one commit that touches at least 100 files, since Linux 3.0 (July 2011), *i.e.*, the period in which Coccinelle was becoming more established. There are 88 such developers, of which 67 (76%) are maintainers. All but two of the others are in the Other category. 39 (44%) of these develop-

[3] https://marc.info/?l=kernel-janitors&m=150403263119030&w=2

ers overall and 31 (46%) of these maintainers have commits using Coccinelle. Of the 88 developers, 27 (31%) have 1-5 commits using Coccinelle, 9 (10%) have 6-100 such commits, and 3 (3%) have more than 100. Among the 67 maintainers, 21 (31%) have 1-5 commits using Coccinelle, 8 (12%) have 6-100 such commits, and 2 (3%) have more than 100. These numbers suggest that Coccinelle is well known among the Linux kernel developers and maintainers who can benefit from it most.

Finally, we consider the most established kernel contributors. We collect the set of maintainers from the Linux 4.15 MAINTAINERS file and the set of developers who have committed at least one patch between Linux 3.0 and Linux 4.15. 45 have at least 10 years of experience as committers and 117 have committed at least 1000 patches. 29% and 32% of these, respectively, have created at least one patch that uses Coccinelle. These numbers reflect the knowledge of Coccinelle at the core of the Linux kernel developer community.

6.3 Changes performed using Coccinelle

Coccinelle facilitates performing changes across the kernel, that may cover code managed by multiple maintainers. Some examples are as follows:

TTY. Remove an unused function argument. One commit (429b474990cb) in 2015, affecting 11 TTY driver files. The author is not a maintainer, but has over 350 commits in the Linux kernel, since 2013.

IIO. Add missing `__devinit` and `__devexit` annotations. One commit (8e8287526844) in 2012 affecting 28 new IIO driver files. The author is an IIO maintainer.

DRM. Eliminate a redundant field in a data structure. One commit (438b74a5497c) in 2016 affecting 54 direct rendering manager (DRM) files. The author is a maintainer, but not for the affected files.

Interrupts. Prepare to remove the `irq` argument from interrupt handlers, and then remove that argument. 40 commits (*e.g.* f4acd122a738) in 2015, affecting 188 files (mostly `drivers`, `arch`). The author is a core Linux developer with over 3500 commits in the Linux kernel since the start of the Linux kernel's usage of git (2005).

More generally, Figure 9 characterizes as cleanups or bug fixes the complete set of patches that use Coccinelle from maintainers. Typical cleanups address generic C issues, such as useless double semicolons, as well as introductions of new APIs and refactorings in preparation for the introduction of new APIs. Commonly identified bugs include memory leaks, allocation of a memory region of the size of a pointer rather than the size of the referenced structure, and storing a potentially negative value in a variable of type `unsigned int` and then checking whether the value is less than zero.

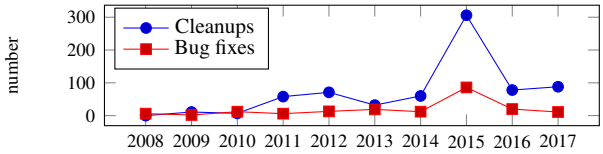

Figure 9: Cleanup vs. bug fix changes among maintainer patches using Coccinelle

As noted in Section 2.5, Coccinelle facilitates making changes that affect the entire Linux kernel source tree, and in particular subsystems managed by different maintainers. While the initial problem of knowing who maintains which part of the kernel was resolved in 2009 by the introduction of the `get_maintainer.pl` script, it is not always clear who should actually commit the changes, particularly when there are dependencies between the resulting patches. The problem of managing cross-tree changes was a proposed topic at the 2017 Linux Kernel Maintainers Summit. Preliminary discussions included proposing the use of Coccinelle for refreshing cross-tree changes when patch sets are incompletely applied [3].

6.4 Semantic patches in the Linux kernel

Since 2010, the Linux kernel has hosted a set of semantic patches in its `scripts/coccinelle` directory. Semantic patches are categorized as being related to APIs (17), resource release (7), iteration (5), locking (4), NULL values (4), test expressions (4) and others (18). The kernel Makefile contains a `coccicheck` target that runs one or all of these semantic patches on the entire kernel or some portion thereof. Kernel developers may thus easily use Coccinelle to check their work, without learning SmPL. Such use, however, is not visible in the kernel history.

As of Linux v4.15, there are 59 semantic patches in the Linux kernel. Figure 10 shows the number of commits including new semantic patches per year, as contributed by various categories of users. Semantic patches were initially contributed by the Coccinelle developers. Recently, 2-3 have been contributed each year, and a few more requested, by the wider kernel community.

6.5 0-day build testing service

Intel's 0-day testing service [26, 60] runs a number of static analyses on commits to hundreds of Linux kernel git trees, both public and private. Kernel developers may check their changes themselves on only one configuration and then rely on the 0-day service for the rest. Coccinelle-based reports generated by the 0-day build testing service come in two forms. If the semantic patch producing the report is able to propose a fix for the identified problem, then the report contains this patch. The

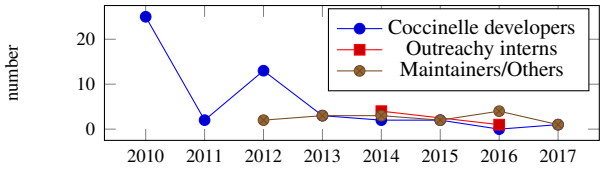

Figure 10: Number of commits adding a semantic patch to the Linux kernel source tree per year

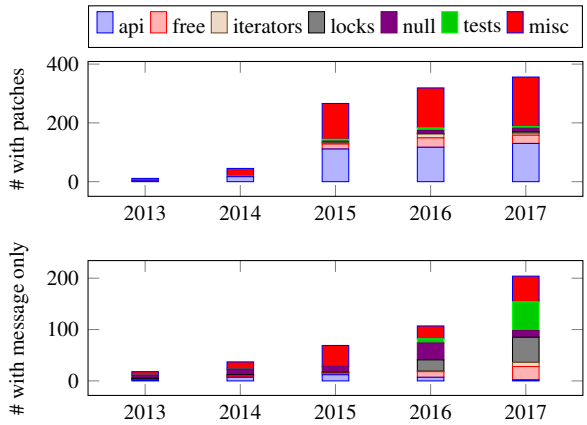

Figure 11: 0-day reports mentioning Coccinelle per year

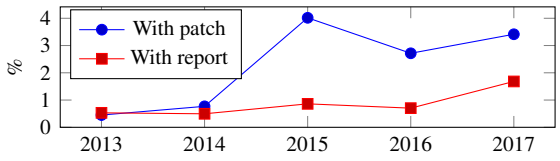

Figure 12: % of 0-day reports mentioning Coccinelle

remaining Coccinelle-based reports contain a textual error or warning message, accompanied by a code extract highlighting the relevant lines, as indicated by the semantic patch. Figure 11 shows the number of public reports that mention Coccinelle in various categories, distinguishing between those that include a patch or only a message. Figure 12 likewise shows the percentage of all public reports mentioning Coccinelle. The most common type of report including a patch removes a field initialization in drivers that is redundant with respect to the driver core (244 patches). The most common type of report including only a message detects missing unlocks (68 reports). The latter reports are manually checked by a Coccinelle developer and have few false positives. Both semantic patches involve kernel-specific features.

7 The Coccinelle Community

A measure of the long term potential impact of a project is the willingness and ability of external developers to contribute to the project's development and maintenance. Today Coccinelle amounts to over 84,000 lines of OCaml code. 25 developers have contributed to Coccinelle, with over 3000 commits over 12 years for one developer, almost 1000 in the first few years for another developer, and 200-300 commits in the last few years for several others. All of the contributors with more than 5 commits have been somehow affiliated with the core development team, as either an employee or a guest. These numbers are likely related to the fact that the implementation language of Coccinelle, OCaml, is not widely used in the target developer community, and to the interdisciplinary nature of Coccinelle, which builds on programming-language concepts but targets the systems developer community. The small number of contributions by external developers may be a source of long term fragility. Nevertheless, the fact that several developers have joined the project in recent years and each made around 200 or more commits suggests that the code is accessible to developers who did not initiate the project.

Coccinelle is packaged with a number of Linux distributions, such as Ubuntu [62], Debian [15], Fedora [18], Gentoo [23], and Archlinux [1]. It is also available for FreeBSD [21] and NetBSD [40]. The full commit history is available at Github [8]. Although Coccinelle is developed using OCaml, there has been an effort to limit the amount of dependence on the traditional OCaml culture and infrastructure. Some needed OCaml libraries are bundled with the Coccinelle distribution, in case they are not available on the local machine. Once Coccinelle is installed, it is fully usable, via the C-like SmPL language and Python scripting, without knowing OCaml.

Although Coccinelle is mainly used on the Linux kernel, it is also used on other software projects. RIOT [49], qemu [47] and systemd [58] include semantic patches in their source code distributions. Patches mentioning Coccinelle are also found in the commit histories of systems software projects such as cpython (d1302c01544 and 228b12edcce) [13], wine (f6ced24999f etc.) [64], and even one in Firefox (ab4e3a0d4213) [19]. The latter used Coccinelle's rudimentary support for C++.

8 Related Work

Academic software development tools. Other academic software development tools that have had an impact on industry practice include CIL [39], LLVM [29], and Metal [17]. CIL provides basic parsing and visitor infrastructure for processing C code, and is used for rapid prototyping as well as being at the base of mature tools such as Frama-C [20]. LLVM is a compiler infrastructure that originally targeted providing good support for static, link-time, and run-time optimization, and has evolved into a common alternative to gcc, due to its cus-

tomizability, speed and space efficiency, and permissive license [7]. Neither has specifically targeted the Linux kernel and its particular coding style. Indeed, LLVM still does not fully support Linux kernel code, despite a long refactoring effort [33]. Both tools have furthermore focused on building a user community rather than taking on the challenge of integrating into an existing one.

Metal is an automata-based tool for scanning large systems source code bases for faults such as use after free and inconsistent locking. It was never made publicly available. Instead, it was the foundation of the highly successful static analysis tool Coverity [2]. Coverity has been used on the Linux kernel more or less over the years, depending on the degree to which its results have been made freely available. Nevertheless, the freely available results address generic C issues, rather than Linux specific properties.

Development tool impact analysis. Koyuncu *et al.* [28] compare properties of Linux kernel patches that are entirely manually generated, manually generated in response to a tool report, and tool generated according to a manually written transformation rule. The patches in the third category are primarily generated by Coccinelle. They find that manually generated patches are accepted more quickly than tool-supported patches, but that the acceptance rate of the latter is increasing. In contrast, we study what kinds of Linux kernel developers use Coccinelle, for what purpose, and what features of Coccinelle have led to its acceptance.

Other tools used on the Linux kernel. Checkpatch is a regular-expression based style checker, whose use is required by the Linux kernel patch submission checklist [32]. It does not have a global view of the code, so it cannot detect inconsistencies that involve multiple code fragments, such as a variable declaration and its use.

Sparse [4] was an effort by Linus Torvalds to develop an open source static checker for the Linux kernel, in response to Metal. Sparse processes developer-provided annotations, enabling it to, *e.g.*, detect endianness issues. Smatch [55] grew out of sparse as a more flexible bug finding tool. Like Coccinelle, Smatch is scriptable, but rules are expressed at the abstract-syntax tree level rather than at the source code level. Thus, the user needs to know about internal representations. Smatch also does only bug finding, not transformation. On the other hand, Smatch tracks variable values, while Coccinelle reasons only in terms of code structure. Thus, smatch can find bugs such as off-by-one errors that are difficult to find using Coccinelle. Thousands of commits in the Linux kernel are derived from the use of smatch, but the creation of new rules is mostly limited to the tool author.

Another academic effort on improving Linux kernel code is the Linux Driver Verification project [27]. It centers around developing infrastructure and rule sets making it possible to apply verification tools such as CPAchecker and BLAST to the Linux kernel. A few hundred commits in the Linux kernel are based on its results. The Undertaker project [6, 59], in contrast to the other tools, finds bug in the use of configuration variables.

9 Lessons Learned

In this paper, we have reviewed the evolution of the program transformation tool Coccinelle and its impact on the Linux kernel. The experience of Coccinelle can help guide other projects that want to have an impact on an open source systems developer community.

First, visibility is necessary. The Coccinelle developers taught by example, by using Coccinelle to make a sustained contribution to the Linux kernel. At the same time, they organized workshops on the use of Coccinelle and presented Coccinelle in a variety of developer conferences, both focusing on the user-visible aspects of Coccinelle, to make the tool accessible to developers.

Second, the tool must be easy to install and freely available. Coccinelle is implemented in OCaml, but targets C developers. There has been a concerted effort to minimize reliance on OCaml infrastructure. The cost is a complex build system, but it reduces the chance that users will immediately abandon Coccinelle because it is difficult to install. Likewise, Coccinelle is freely available (GPLv2), with no registration requirement.

Third, the tool must be easy to use and robust, with support to quickly address problems encountered by users. While many research prototypes are only robust enough to complete an evaluation for a paper submission, users will try it on all kinds of code, and use it in unanticipated ways. A strength of Coccinelle is its lax C parser, motivated by the need to parse code without reliance on header files. This has the side effect of allowing it to adapt to C variants used by different projects. On the other hand, many users have mentioned that the documentation, consisting mainly of a BNF, some examples and some tutorial presentations, is hard to understand. Nevertheless, Coccinelle has an active mailing list [9] on which user problems are quickly addressed. Fixes are made available quickly via Github [8].

Finally, in a research setting, there is a constant temptation to make a tool do more, until the resulting complexity causes the tool to collapse under its own weight. While new features have been added to Coccinelle, the tool has remained within the scope of pattern matching-based transformation of C code. This focus has allowed it to grow and achieve practical success in this area.

Availability. `http://coccinelle.lip6.fr`. This work was supported in part by ANR-NRF ITrans.

References

[1] Archlinux. https://aur.archlinux.org/packages/coccinelle.

[2] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., GROS, C.-H., KAMSKY, A., MCPEAK, S., AND ENGLER, D. R. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (2010), 66–75.

[3] BOTTOMLEY, J. Maintainer's summit agenda planning. https://lists.linuxfoundation.org/pipermail/ksummit-discuss/2017-October/004803.html.

[4] BROWN, N. Sparse: a look under the hood, June 2016. https://lwn.net/Articles/689907.

[5] BRUNEL, J., DOLIGEZ, D., HANSEN, R. R., LAWALL, J. L., AND MULLER, G. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL* (Savannah, GA, USA, 2009), pp. 114–126.

[6] Checkkconfigsymbols. https://github.com/torvalds/linux/blob/master/scripts/checkkconfigsymbols.py.

[7] Clang vs other open source compilers. http://clang.llvm.org/comparison.html.

[8] Coccinelle github repository. https://github.com/coccinelle/coccinelle.

[9] Coccinelle mailing list. https://systeme.lip6.fr/pipermail/cocci.

[10] Coccinelle website. http://coccinelle.lip6.fr/.

[11] Coccinellery. https://github.com/coccinelle/coccinellery.

[12] CORBET, J. KS2010: Lightning talks, Nov. 2010. https://lwn.net/Articles/412750.

[13] CPython. https://github.com/python/cpython.git.

[14] DANELUTTO, M., AND COSMO, R. D. A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. In *International Conference on Computational Science* (Omaha, NE, USA, June 2012), pp. 1837–1846.

[15] Debian. https://packages.debian.org/search?keywords=coccinelle.

[16] EDGE, J. Inside the mind of a Coccinelle programmer, Aug. 2016. https://lwn.net/Articles/698724.

[17] ENGLER, D. R., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI* (2000), pp. 1–16.

[18] Fedora. https://apps.fedoraproject.org/packages/coccinelle.

[19] Firefox. https://github.com/mozilla/gecko-dev.git.

[20] Frama-C. https://frama-c.com.

[21] FreeBSD. http://www.freshports.org/devel/coccinelle.

[22] Gartner says worldwide sales of smartphones grew 9 percent in first quarter of 2017. https://www.gartner.com/newsroom/id/3725117.

[23] Gentoo. https://packages.gentoo.org/packages/dev-util/coccinelle.

[24] Glimpse. http://webglimpse.net.

[25] HENSON, V. Semantic patching with Coccinelle, Jan. 2009. https://lwn.net/Articles/315686.

[26] KERRISK, M. Ks2012: Kernel build/boot testing, Sept. 2012. https://lwn.net/Articles/514278.

[27] KHOROSHILOV, A., MANDRYKIN, M., MUTILIN, V., NOVIKOV, E., PETRENKO, A., AND ZAKHAROV, I. Configurable toolset for static verification of operating systems kernel modules. *Programming and Computer Software 41*, 1 (2015), 49–64.

[28] KOYUNCU, A., BISSYANDÉ, T. F., KIM, D., KLEIN, J., MONPERRUS, M., AND TRAON, Y. L. Impact of tool support in patch construction. In *ISSTA* (2017).

[29] LATTNER, C., AND ADVE, V. S. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO* (2004), pp. 75–88.

[30] LAWALL, J., PALINSKI, D., GNIRKE, L., AND MULLER, G. Fast and precise retrieval of forward and back porting information for Linux device drivers. In *USENIX Annual Technical Conference* (2017), pp. 15–26.

[31] LAWALL, J. L., BRUNEL, J., PALIX, N., HANSEN, R. R., STUART, H., AND MULLER, G. WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process. *Softw., Pract. Exper. 1*, 43 (2013), 67–92.

[32] Linux kernel patch submission checklist. https://www.kernel.org/doc/html/v4.15/process/submit-checklist.html.

[33] LLVM. http://llvm.linuxfoundation.org/index.php/Main_Page.

[34] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J. L., AND MULLER, G. Fast and portable locking for multicore architectures. *ACM Trans. Comput. Syst. 4*, 33 (2016), 13:1–13:62.

[35] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A soundy analysis for Linux kernel drivers. In *USENIX Security* (Vancouver, BC, Canada, 2017).

[36] MACKENZIE, D., EGGERT, P., AND 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.

[37] MÉRILLON, F., RÉVEILLÈRE, L., CONSEL, C., MARLET, R., AND MULLER, G. Devil: An IDL for hardware programming. In *OSDI* (San Diego, CA, USA, 2000), USENIX Association.

[38] MOLNAR, I. Revert "make 'bt_sfi_data' const". https://lkml.org/lkml/2017/12/28/137.

[39] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, 11th International Conference* (Grenoble, France, Apr. 2002), LNCS 2304, pp. 213–228.

[40] NetBSD. ftp://ftp.netbsd.org/pub/pkgsrc/current/pkgsrc/devel/coccinelle/README.html.

[41] Outreachy. https://www.outreachy.org.

[42] PADIOLEAU, Y. Parsing C/C++ code without pre-processing. In *CC* (York, UK, Mar. 2009), pp. 109–125.

[43] PADIOLEAU, Y., LAWALL, J., HANSEN, R. R., AND MULLER, G. Semantic patches for collateral evolutions in device drivers. In *Linux Symposium* (Ottawa, Canada, June 2007).

[44] PADIOLEAU, Y., LAWALL, J. L., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys* (2008), pp. 247–260.

[45] PADIOLEAU, Y., LAWALL, J. L., AND MULLER, G. Understanding collateral evolution in Linux device drivers. In *EuroSys* (2006), pp. 59–71.

[46] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., MULLER, G., AND LAWALL, J. Faults in Linux 2.6. *ACM Trans. Comput. Syst. 32*, 2 (2014), 4:1–4:40.

[47] Qemu. `https://github.com/qemu/qemu.git`.

[48] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. Symdrive: Testing drivers without devices. In *OSDI* (Hollywood, CA, 2012), USENIX, pp. 279–292.

[49] Riot. `https://github.com/RIOT-OS/RIOT.git`.

[50] RODRIGUEZ, L. R., AND LAWALL, J. Increasing automation in the backporting of Linux drivers using Coccinelle. In *EDCC* (2015), pp. 132–143.

[51] RYZHYK, L., KEYS, J., MIRLA, B., RAGHUNATH, A., VIJ, M., AND HEISER, G. Improved device driver reliability through hardware verification reuse. In *ASPLOS* (2011), pp. 133–144.

[52] SANG, W. Evolutionary development of a semantic patch using Coccinelle, Mar. 2010. `https://lwn.net/Articles/380835`.

[53] SHEVCHENKO, A. Revert "make 'bt_sfi_data' const". `https://lkml.org/lkml/2017/12/28/85`.

[54] SLOCCount. `https://www.dwheeler.com/sloccount`.

[55] Smatch. `https://blogs.oracle.com/linuxkernel/smatch-static-analysis-tool-overview,-by-dan-carpenter`.

[56] STUART, H., HANSEN, R. R., LAWALL, J. L., ANDERSEN, J., PADIOLEAU, Y., AND MULLER, G. Towards easing the diagnosis of bugs in OS code. In *4th Workshop on Programming Languages and Operating Systems* (Stevenson, Washington, Oct. 2007).

[57] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. *ACM Trans. Comput. Syst. 24*, 4 (Nov. 2006), 333–360.

[58] Systemd. `https://github.com/systemd/systemd`.

[59] TARTLER, R., SINCERO, J., DIETRICH, C., SCHRÖDER-PREIKSCHAT, W., AND LOHMANN, D. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer 14*, 5 (2012), 531–551.

[60] The kbuild-all archives. `https://lists.01.org/pipermail/kbuild-all`.

[61] TORVALDS, L. Linux kernel source tree. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/`.

[62] Ubuntu. `https://packages.ubuntu.com/zesty/coccinelle`.

[63] VAUGHAN-NICHOLS, S. J. Linux totally dominates supercomputers. `http://www.zdnet.com/article/linux-totally-dominates-supercomputers`.

[64] Wine. `https://github.com/wine-mirror/wine.git`.