Verifications around the Linux kernel

Alexandre Lissy
Mandriva
alissy@mandriva.com
Laboratoire d'Informatique de l'Université de Tours
alexandre.lissy@etu.univ-tours.fr

Stéphane Laurière
Mandriva
slauriere@mandriva.com

Patrick Martineau
Laboratoire d'Informatique de l'Université de Tours
patrick.martineau@univ-tours.fr

Abstract

Ensuring software safety has always been needed, whether you are designing an on-board aircraft computer or next-gen mobile phone, even if the purpose of the verification is not the same in both cases. We propose to show the current state of the art of work around the verification of the Linux kernel, and by extension also present what has been done on other kernels. We will conclude with future needs that must be addressed, and some way of improvements that should be followed.

1 Introduction

The Linux kernel is an important piece of code, both in terms of size (it is currently evaluated at 5 million lines of code) and of role. Like any software, it needs to be tested. Testing generally occurs through users running the kernel and reporting the bugs they identify. Achieving good coverage of the code with this manual approach is however not feasible: code analysis can be used to ease verification and bring it to a new level. These techniques are known as *static analysis* and have been used for several years, not only for the Linux kernel. Linux distributions do not use exactly the vanilla kernel and may have patches added, so it is important to be able to check not only upstream code but also downstream code. Having a clear view of what has already been done in the field of Linux kernel checking is a first but important step before deciding what kind of new verifications to work on. First, a more precise description of our objectives is available in section 2, followed by a presentation of some of the tools and related papers starting with one of the main reference in section 3, then presenting the SLAM initiative in section 4. The next section 5 provides a specific look at the Linux case, presenting SATURN in section 5.1, then some work around model checking in section 5.2. Coccinelle and Undertaker related work is presented respectively in section 5.3 and 5.4 just before presenting another way to approach the problem in section 5.5. Before concluding, some work being done to entirely verify an OS kernel in section 6 is presented.

2 Objectives

Our main goal is to bring more stability from a runtime point of view to the kernel, and in fact to any software that is packaged and available in Linux distributions in general, and in the Mandriva Linux distribution in particular. The kernel is an interesting piece of code, because it evolves rapidly (so that complex quality assurance is hard) and the codebase differs between the vanilla one and the one used by distributions, which apply patches relating typically to hardware compatibility, extended features, backports, etc. This part of a Linux distribution is also a good candidate because of its aura: is it easier to find people interested in augmenting the quality of this kind of code than for other parts of other critical code, even though some parts of the kernel are less likely to bring attention. Thus the question: "can we model-check the Linux kernel?", which led to a first part of the work: what is the current status of checking techniques in Linux kernel and also in any other kernel component? What is the literature on model-checking techniques applied to the same case? What has been already tested? What are the current known

limits of model-checking? What are the current known limits of tools applying model-checking? Even "what does model checking exactly mean in the context of the Linux kernel?". So the objectives of the present article is to summarize what has been collected as part of this digging work, trying to give a view of the current state of the model checking techniques applied to Linux kernel and other kernel (Windows, seL4). It is in no way completely exhaustive but a hopefully good compilation of several "related work" section in many papers and communications that have been found.

3 Foundation: From Stanford to Coverity

The ability to verify the source code of a kernel has been studied by Engler et al., in [9] in which they present some new way to enforce system reliability by using rules in the compiler written by the programmers themselves. By "systems", they do not only target kernel, but also libraries or embedded systems. They intend to check interfaces usage, i.e. whether APIs are used correctly. First, they describe why in their opinion model-checking is not a viable option for this problem with the following arguments:

- Specifications are costly to build, hard to write
- They may not exactly abstract what they should
- Real life shows that they often miss some points and are over-simplifying

So the authors introduce *Meta-Level Compilation (MC)* to allow checking programmer-written rules with an extended-version of GCC, **xg++** implementing the *metal* language, which is high-level and state-machine based. The technique is not new, and the authors mention previous work aiming at similar objectives like *ctool* and *Open C++*. The lack of data flow information within these tools is however identified by the authors as a key limitation which make them harder to use and less powerful than Meta-Level Compilation. State-machines in the *metal* language work by matching interesting features (using C++-written patterns) in the analyzed code and then causing transitions between states. Meta-Level Compilation can be used not only to find bugs, but also for optimizing code: automatically detecting if a shared variable is never written enables identification of excessive locking usage; and the reverse might be used to detect non-protected variables. Of course, those kind of rules cannot be generic and are project-specific. *Metal*-written rules are first compiled using the `mcc` compiler, before being loaded into `xg++`. Several *checkers* are proposed as usage examples: assertions side-effects, compile-time assertions, correct usage of APIs (malloc/free, userspace pointer usage in kernel code), null pointer dereference, etc.

Coverity, a widely known tool to analyze source code, which has a partnership with the Department of Homeland Security (USA) [1] and was used to check many FLOSS projects, has its roots in the Stanford Checker (which is the implementation of Engler et al. work described in [9]). This checker is not available as free software. Dan Carpenter committed *Smatch*[2] to provide the same functionality in a FLOSS compatible way. A couple of years later, the same author committed a new paper, [10], which gives a feedback on both static analysis and model checking after several experiences with both. The paper consists mainly of real case studies in both areas, not to incriminate one or the other technique but to describe and compare. They first explain how they used both approaches:

- Using classical explicit state model checkers, with two approaches for the specifications problem: one automating the extraction of slice of functionalities translated into model-checking language, another model-checking directly the C code.
- Using meta-level compilation approach, classified as static analysis, which according to them reduces the work needed to find bugs. However, this method is unsound: errors can be missed. Also, they did not model the heap, tracked most variables values nor do alias analysis. They did their best to avoid the use of annotations in the code.

Regarding model-checking, they state the assumption "model-checking will find more bugs" is false: when checking an embedded system (FLASH), this technique found fewer (four times) bugs than the static analysis method, even though the model-checking approach identified some bugs that the previous method was not able to. The main reasons for the differences are: the code needs to be executed and the environment has to

[1] http://scan.coverity.com/
[2] http://repo.or.cz/w/smatch.git

be modeled. The discussion section brings up several interesting issues.

- They thought it would be hard to find many bugs in a complex system, but it turned out false: a codebase of at least one million lines of code not showing bugs is more a tip that there is a bug in the bug finder.
- It is easier to write code on how a property must be checked than why the property was violated.
- Not being too general is easier to handle for checking
- Hard to inspect errors may be left unfixed simply because users ignore the warning and does not understand it (they give an example from commercial *PREfix* tool)
- More analysis can result in useless analysis, because it will expose complex bugs.
- Another myth they destroy is that "all bugs matter": more bugs reported does not imply more fixes committed, regardless whether it is free software, because no importance ordering is given.

From their point of view, model checking is less powerful, mainly because it needs a good model of the environment (which is difficult, requiring weeks or months of work), and it cannot analyze more efficiently such large codebase than static analysis can do. However, model-checking has several advantages over static analysis which all result in stronger correctness results.

4 Microsoft: From SLAM Project to SDV

Starting in 1999, as a result of a brainstorming inside the Software Productivity Tools group at Microsoft Programmer Productivity Research Center[3], three projects were launched aiming at improving the quality of software, especially, but not limited to, drivers. All these projects shared a unique goal: checking interface usage. Only the approach to solve the problem is different, and a brief presentation of the two that did not last as long as SLAM is provided:

Vault A new programming language with pre/post conditions on the types, thus enabling definition of rules that the program must follow; it is now used as part of MSIL byte-code for CLR virtual machine, and is available as a Visual Studio plugin: [8].

ESP is closer to SLAM, but does not take the same course for the implementation and the trade-offs of the static analysis.

Regarding our topic, only SLAM is interesting, first because this project has been successful, and also because of the lack of information available on ESP project. The first major contribution to the problem of checking interface usage is the ability to abstract C program as Boolean programs[6, 5]. The Boolean program is created from the C programs by taking the Control Flow Graph, and pruning variables with Boolean variables. Boolean programs are interesting as reachability and termination is decidable. Building those is done using *Counter-Example Guided Abstraction Refinement (CEGAR)* [5, 7, 2]. Along with their Boolean Program model, the author contributes a model-checking algorithm[5] to check this model.

The authors introduce an interface description language SLIC[7] and present the CEGAR process articulated over three tools: `C2BP` which transforms C programs to boolean programs, `Bebop` allows model-checking of boolean programs and `Newton` which checks path feasibility between C program and boolean program. *Bebop* makes uses of Context-Free Language Reachability results [32]. As of 2002-2003, the SLAM project was working well and could be used as a basis for "Static Driver Verifier" [3]: the goal is to have an easy-to-use SLAM, to be distributed for use by driver developers. Joint work with Drivers Teams led to a lot of new checking rules being written not by SLAM developers. In [4], the authors show that over the 470 rules, only 60 were written by checking "experts": this validates the widespread possibility for developers of interfaces to provide checking rules. In the same paper, they also note that the use of SDV and rules complexity has been decisive in the new driver model design for latest Windows releases: too complex rules meant complex checking, and thus it has been decided to re-design the interface to ease checks. Work to improve SLAM into SLAM2 has been done as stated in [2]: less false-positives (from 19.7% to 0.4% in the worst case reported), less timeouts (CEGAR loop unable to finish, from 6% to 3.2% in the worst case reported).

5 Linux SDV?

The Linux kernel has also been a basis for verification work, because it has widespread diffusion and is easily fixable. An overview of the work that has been done and the projects evolving around this topic is proposed. Engler et al. [10] already started some important work toward checking the Linux kernel, which they continue to do with their Coverity tool.

5.1 Saturn

In [31], the authors describe a generic framework aimed at detecting errors in large codebase. Source code is translated into boolean constraints that can be used to check properties thanks to SAT-solving tools. The advantages of the method as claimed by the authors are the following:

Precision no abstraction is being done

Flexibility boolean constraints does not put any requirements on the language used

Compacity boolean formulas can be simplified when doing intra-procedural analysis. For inter-procedural, a summary of the function is compiled and used to process the verification. It is required as the authors aims at large code base and SAT is NP-complete

SATURN processing can be parallelized, the authors claiming that an 80-processor cluster reduces the processing from 23 hours to 50 minutes, when analyzing the Linux kernel. They also contribute experimental results of the running against the previous codebase: more errors found than in previous literature's work and fewer false positives. Another interesting contribution is the autoslicing, allowing to limit the codebase to relevant parts regarding the property analyzed. They contribute two case studies:

- Checking finite state properties, also known as temporal safety properties (as in [5, 6, 7]), with an example on verifying locks
- Checking for memory leaks (not only on the Linux kernel, but also on samba, openssl, postfix, openssh and binutils)

5.2 Model Checking Concurrent Linux Device Drivers

The authors present[30] an approach to model-check shared memory programs, thus enabling the automated verification of Linux device drivers. Their tool, `DDVerify`, uses predicate abstraction and their technique introduces concurrent software verification with predicate abstraction. They justify their approach of targeting shared memory because the resulting bugs are very hard to discover and understand. They also contribute a concurrent model of the needed parts of the Linux kernel API. `DDVerify` generates an annotated version of the source code to be checked with a SAT-solver (`SatAbs`), using assertions given by the user. Dealing with concurrency implies being able to deal with threads: it has implications on the state space resulting of the model. To ensure finite space (to be able to use the `SMV checker`), their tool uses a static and bounded number of threads. Further work is needed to allow the use of infinite dynamic number of threads and the `Boppo` checker. To automate the process, they implement a *CEGAR* loop as described in [5, 7, 2]. However, they conclude with two important facts:

- In their approach, the performance of the model-checker is critical: most of the execution time is spent on verifying the abstracted program. They used three tools, `Cadence SMV`, `Boppo` and `Bp`
- Model-Checkers performances for Concurrent Device Drivers needs to be improved significantly to be able to cope with real world drivers

They also claim to have a better (more accurate) model for the operating system than in SLAM or BLAST projects. They provide results of their benchmarks in two cases: sequential and concurrent, using a small benchmark driver, machwzd (less than 500 lines of code, using locks, IO and timers), and running on Intel Xeon 3GHz, 4GB RAM. The `bp` model-checker executes in about one second, while `Boppo` and `SMV` are around 30 seconds, in the sequential case. When going concurrent, only `SMV` is available, and runs around 400 seconds on average, with peaks at 1800 seconds. Despite poor performances, their tool found one new bug related to memory regions usage (`request_region()` function).

5.3 Coccinelle

Coccinelle is a French word for a kind of beetle that eats bugs. The goal of the Coccinelle project is to kill bugs before they are introduced. It started with [14] in which the authors describe and analyze recent *collateral evolutions* and contribute requirements of a helper tool: those evolutions are API changes that impact potentially a huge number of files in the whole kernel. They give three examples:

- Elimination of the `check_region()` function, which started in Linux 2.4.2 (released on February 2001) and that was still not completed by Linux 2.6.10 (released on December 2004).
- Addition of an extra argument to the `usb_submit_urb()` function, started in Linux 2.5.4 (released on February 2002) and that was still not completed by Linux 2.6.10.
- New function for copying data from userspace to kernel space `video_usercopy()`. Worse than previous examples, this new method was introduced in a driver for Linux 2.6.3 but the old one remained in place, and it resulted in the bug never being fixed, and the **new** method being **removed** as of Linux 2.6.8.

Thanks to this study highlighting the defects of API changes in the whole kernel, the authors contribute *semantic patches* (classical patches with augmented information to describe the context, and being able to go further in the manipulation of the code), which have to comply with three requirements:

- Identify the code to modify
- Description of the new code
- Existing context impact

A more detailed description of the collateral evolution problem is also available in [18, 20]. This research report also contributes an interesting comparative study of the evolution of APIs in the Linux kernel, from releases 2.2 to 2.6, showing the huge increase of APIs available inside the kernel. Focusing on the Semantic Patches notion, the authors presents a detailed overview of their contribution in [19]. They give a motivating example with the changes that were done to the SCSI stack. They also justify why they chose to base their Semantic Patches approach on classical patches: it is human readable, which is important for the acceptation of the tool, yet it allows to be much more general.

In [15], the authors contribute design information on the heart of the Coccinelle tool: basically, they are using model-checking under the hood. Source code is parsed into a model, and semantic patches are translated to a CTL formula which can be used in a model-checking algorithm against the previous model. However, Coccinelle is able to **change** the source code after successful matching of CTL formula. They contribute experimental results of runs of their tool, showing fast processing. Much of the modifications have been automatically applied successfully, and for the remaining ones, changes in the Semantic Patches were needed but proved to be successful too. In [16, 21] the authors present a more kernel developers-oriented explanation of the Semantic Patches approach, with many collateral evolution examples addressed by Coccinelle.

In [17], the notion of "mega collateral evolution" is introduced, as a very huge evolution, touching everywhere in the Linux sources, and they propose an analysis of 23 out of 35 identified: number of files changed, impact (lines of code), number of maintainers involved, duration of the changes, misses and errors. And they compare the same collateral evolution being processes by hand-written Semantic Patches applied by Coccinelle, using criterion such as Sematic Patch size (lines of code), average execution time, misses and percent ok. The same analysis is introduced for "error-prone evolutions," i.e. evolutions that resulted in errors (mistakes, conflicts). In both case, the semantic patches approach is much more reliable, even if not perfect.

The major contribution at this point is real-life proof of the effectiveness of Coccinelle. With [28], the authors make another use of their code matcher and transformer, targeting at finding bugs using semantic patches. The contribution not only finds bugs, but also, with examples, shows how to automatically workaround them when it is possible. In [13] and in a more detailed version of this work available in [12], three case studies are presented: consistency of errors checking (with functions returning `NULL` or `ERR_PTR`), detection of allocation and deallocation protocols, and bad interaction with freeing memory. The authors also contribute a "protocol finding", which allows to automatically discover the

correct usage of an API.

Another use of the bug finding aspect, but not limited to the Linux kernel source code is given in [25] where several open source software projects are analyzed with Coccinelle. More recent contributions in [1] aim at helping the creation of semantic patches: the idea is to let the developer fix one driver as an example, and then infer a semantic patch that will be applicable on all the other drivers. The main contribution at this level is the `SPFIND` algorithm, that is the heart of the tool.

Another use of the Coccinelle tool and its companion is described in [22, 23], where Herodotos is introduced. This contribution aims at following bugs found in software over several versions, and the authors are able to show life cycle of classes of bugs. An interesting example is the *notand* class: misuse of boolean and bit operators. As of post 2008 kernel, a large number of these defects were dropped: this has been a target of the authors since Linux version 2.6.24, showing that their patches have improved the situation.

5.4 Undertaker

With Undertaker, the authors aim at another kind of verification around the Linux kernel: configurability. In [27], the authors introduce the LIFE (*LInux Feature Explorer*), and provide a much more detailed description of their work in [29]. In the first paper, they contribute a first important element: variability model and variability implementation. The model consists of the *Kconfig* part of the Linux kernel, i.e. where configurations options are defined and can be enabled or disabled. The implementation concerns the usage of those options inside the sourcecode. Contributions of the LIFE are mainly on three parts: **source2rsf**, a tool used to extract compilation information; **kconfigextractor**, a tool to generate a boolean formula-based model of the configuration options defined by Kconfig; and **undertaker**, a tool which analyzes the `RSF` files previously extracted. Once everything is setup, satisfiability between variability model and variability implementation is checked, and thus consistency between both. Later, in [29], the authors give more details about logic behind all steps of the analysis. Also, they explain why the current `grep`-based tool that checks for configurability defects is not enough: it is obvious that the `checkkconfigsymbols.sh` script is not being used by kernel maintainers; running this tool exposes issues which have been not fixed for several months.

The given example deals with CPU Hotplug capabilities: a typo between Kconfig and usage in source code lead to hotplug missing, for more than six months. Speculative reasons why maintainers do not use the script are: too many false positives, huge processing time, false negatives. To cope with the huge number of configurable functionalities in the Linux kernel (8000 in the first paper, 10000 in the second), the authors contribute a model-slicing algorithm for Kconfig, otherwise the boolean formulas become too complex and cannot be treated. Detected defects can either be dead code or undead code. Another contribution in this paper is the study of introduced/fixed defects among several kernel releases: they studied versions between 2.6.30 (rc1 and stable) and 2.6.36 (rc1 and stable). A huge peak of "fixed defects" can be seen on the 2.6.36-rc1 release, which the authors explain by the merge of the patches they sent to maintainers to fix the issues they found.

5.5 Integrated Static Analysis for Linux Device Driver

In this paper [24], the authors aim at porting the verifications techniques developed by Microsoft for SDV (*Static Driver Verifier*) [2] to the Linux kernel project. They come up with several contributions. First, they propose an extension of the SLIC language [7] from Microsoft: SLICx. This is both needed because of the lack of available SLIC tool and to fit their needs better:

- Reducing the set of possible types for state fields
- No more parallel assignments
- Allowing any C statement and expressions

For the same reason, lacking source code, they use the CBMC model-checker. A second contribution is a model for Linux, implementing life cycle for a device driver, i.e. `module_init()` and `module_exit()`. As another contribution, they give an example of checking with the RCU API. In their tool, the authors are able to get further than previous tools such as `CBMC` or `SDV`: absence of memory leaks, simulating preemption, deadlocks, race conditions. Testing occurs by annotating drivers. They also propose some experimental results over their solution. A first result is that they argue

about *modular analysis* being faulty when dealing with functions calls specifications. It is the classical issue of modeling the whole environment in model-checking. The verification process given is not automated, many steps being manual.

6 Verification of a kernel: seL4

Work has been done to verify a (almost) full kernel: seL4[11]. It is a secure version of the L4 micro-kernel. They present pretty robust results with a formal proof of a very large part of the micro-kernel, only the boot code is not yet proven. Regarding our goal, this must be looked scarcely: the codebase is "just" 8700 lines of C and 600 lines of assembly language, and both are verified. However, the authors states that performance impact is null or low enough to be discarded. The proof is machine-assisted and machine-checked, but it requires human intervention: implementation is strictly proven against the specifications.

One contribution is a rapid kernel design and implementation, that helps designers balance between formal methods and high performance, through a Haskell prototyping phase that helps to prove the final C implementation. Proof is contributed thanks to the Isabelle/HOL theorem prover, which requires human interaction. The specification part is important: abstract specifications (describing functionalities of the system), and executable specifications (describing how the system works from a low-level point of view). The C implementation is included in the verification toolchain, meaning the authors had to describe C semantics in a model: in fact, they modeled a subset of the C language (C99: types, memory model, structure padding, unsafe pointer handling, pointer arithmetic); limitations towards the full C99 language are: no use of & operator, avoiding as much as possible references (to comply with absence of ordering in expressions evaluation), functions pointers are not allowed, nor `goto` or `switch`.

The authors consider, in their goal to prove security properties, that both the compiler (GCC) and the hardware (ARMv6-based) can be trusted; however GCC is not trusted for predictable bit field compilation and optimization. The authors also contribute a model for the machine, which is needed because assembly code has to be handled and proved. Codebase size and proof size are interesting figures: in Haskell/C, the number of lines of code (LOC) is 8700, once translated in Isabelle's language, it gives 15,000 LOC. And then the proof is 5500 LOC long. The authors also contributes a project-management view of their work, and one interesting point is that their approach is not that bad: in fact, they claim it is better than EAL6 Certification and that it provides stronger guarantee.

7 Conclusion

In this paper we presented an overview of the state of the art in the field of checking the Linux kernel and beyond. We presented work that has been initiated by Engler et al. on verifying Linux, OpenBSD and the FLASH embedded system, and a comparative study of static analysis versus model checking. We presented the work that has been done and published by Microsoft around the automated verification of the usage of APIs by device drivers, which led to a powerful static analyzer tool being available in the latest Driver Development Kit. We presented work done on the Linux kernel that shares some aspects with the Microsoft approach, and also several projects (Coccinelle, Undertaker) that are quite mature. We also presented a fully proven micro-kernel analysis showing some methodology for complete correctness, but which would not scale to the size of the Linux kernel (which is on the order of 500 times bigger). The "verification" topic toward Linux kernel is not a new field, but we can observe that all the approaches in the literature are static analysis. Critical analysis made by Engler et al. [10] on Static Analysis versus Model-Checking gives some enlightenment on the reasons.

However, an interesting way to handle the model-checking approach would be to slice the Linux kernel source code into several sub-parts that can be checked independently: thus, the state explosion issue related to model-checking could be avoided. Autoslicing as described in [29] could be a good starting point. A first step might be accomplished with the Atomic API available in the kernel: it is concise, critical for some parts of the kernel, it mixes C and assembly language, and it has strict usage rules. Plus, it lives inside the `arch` directory, which according to [22] is now the more error-prone directory, before the `drivers`. Generally speaking, this last paper shows that the effort should be emphasized on `arch`. Following the Engler's Meta-Level Compilation idea expressed in [9], it would also be interesting to work inside GCC thanks to the recent availability of plugins: the MELT plugin[26] allows easier

manipulation of the GCC internals especially for pattern matching operations.

References

[1] Jesper Andersen and Julia L. Lawall. Generic patch inference. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 337–346, L'Aquila, Italy, 2008.

[2] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In *International Conference on Computer Aided Verification*, 2010.

[3] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods*, pages 1–20. Springer, 2004.

[4] T. Ball, V. Levin, and S.K. Rajamani. A decade of software model checking with slam. 2010.

[5] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, February 2000.

[6] Thomas Ball and Sriram K. Rajamani. Checking temporal properties of software with boolean programs. In *In Proceedings of the Workshop on Advances in Verification*, 2000.

[7] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[8] Robert DeLine and Manuel FÃd'hndrich. The fugue protocol checker: Is your software baroque? Technical report, 2003.

[9] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. pages 1–16, 2000.

[10] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 405–427. Springer Berlin / Heidelberg, 2004.

[11] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[12] Julia L. Lawall, Julien Brunel, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. Technical Report 08/1/INFO, Ecole des Mines de Nantes, Nantes, France, 2008.

[13] Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 43–52, Estoril, Portugal, 2009.

[14] Julia L. Lawall, Gilles Muller, and Richard Urunuela. Tarantula: Killing driver bugs before they hatch. In *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 13–18, Chicago, IL, 2005.

[15] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *PLOS 2006: Linguistic Support for Modern Operating Systems*, San Jose, CA, 2006.

[16] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Towards documenting and automating collateral evolutions in linux device drivers. Research Report 6090, INRIA, 01 2007.

[17] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, 2008.

[18] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers (long version). Research report 5769, INRIA, Rennes, France, 2005.

[19] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. In *International ERCIM Workshop on Software Evolution (2006)*, Lille, France, 2006.

[20] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, Leuven, Belgium, 2006.

[21] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Semantic patches, documenting and automating collateral evolutions in Linux device drivers. In *Ottawa Linux Symposium (OLS 2007)*, Ottawa, Canada, 2007.

[22] Nicolas Palix, Julia Lawall, and Gilles Muller. Herodotos: A Tool to Expose Bugs' Lives. Research Report RR-6984, INRIA, 2009.

[23] Nicolas Palix, Julia Lawall, and Gilles Muller. Tracking code patterns over multiple software versions with herodotos. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 169–180, New York, NY, USA, 2010. ACM.

[24] H. Post and W. Küchlin. Integrated static analysis for linux device driver verification. In *Integrated Formal Methods*, pages 518–537. Springer, 2007.

[25] Sune Rievers. Finding bugs in open source software using coccinelle, jan 2010.

[26] Basile Sarynkévitch. Extending the gcc compiler with melt to suit your needs. In *RMLL 2010*, 2010.

[27] Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the Linux 8000 Feature Nightmare. In ACM SIGOPS, editor, *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*, 2010.

[28] Henrik Stuart, René Rydhof Hansen, Julia L. Lawall, Jesper Andersen, Yoann Padioleau, and Gilles Muller. Towards easing the diagnosis of bugs in OS code. In *4th Workshop on Programming Languages and Operating Systems*, pages 1–5, Stevenson, Washington, 2007.

[29] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In European Chapter of ACM SIGOPS, editor, *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, 2011.

[30] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 501–504. ACM, 2007.

[31] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.

[32] Hao Yuan and Patrick Eugster. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 175–189, Berlin, Heidelberg, 2009. Springer-Verlag.