RESEARCH CENTRE

Paris

IN PARTNERSHIP WITH:

CNRS, Sorbonne Université (UPMC)

2020 ACTIVITY REPORT

Project-Team WHISPER

Well Honed Infrastructure Software for Programming Environments and Runtimes

IN COLLABORATION WITH: Laboratoire d'informatique de Paris 6 (LIP6)

DOMAIN

Networks, Systems and Services, Distributed Computing

THEME

Distributed Systems and middleware

Contents

Project-Team WHISPER

Creation of the Team: 2014 May 15, updated into Project-Team: 2015 December 01

Keywords

Computer sciences and digital sciences

A1. – Architectures, systems and networks

A1.1.1. – Multicore, Manycore

A1.1.3. – Memory models

A1.1.13. – Virtualization

A2.1.6. – Concurrent programming

A2.1.10. – Domain-specific languages

A2.1.11. – Proof languages

A2.2.1. – Static analysis

A2.2.5. – Run-time systems

A2.2.8. – Code generation

A2.3.1. – Embedded systems

A2.3.3. – Real-time systems

A2.4. – Formal method for verification, reliability, certification

A2.4.3. – Proofs

A2.5. – Software engineering

A2.6.1. – Operating systems

A2.6.2. – Middleware

A2.6.3. – Virtual machines

Other research topics and application domains

B5. – Industry of the future

B5.2.1. – Road vehicles

B5.2.3. – Aviation

B5.2.4. – Aerospace

B6.1. – Software industry

B6.1.1. – Software engineering

B6.1.2. – Software evolution, maintenance

B6.3.3. – Network Management

B6.5. – Information systems

B6.6. – Embedded systems

1 Team members, visitors, external collaborators

Research Scientists

- Gilles Muller [Team leader, Inria, Senior Researcher, HDR]
- Pierre-Évariste Dagand [CNRS, Researcher]
- Julia Lawall [Inria, Senior Researcher]

Faculty Members

- Maria Virginia Aponte-Garcia [CNAM, Associate Professor, until Sep 2020]
- Bertil Folliot [Université Pierre et Marie Curie, Professor, HDR]

PhD Students

- Yoann Ghigoff [Orange Labs]
- Josselin Giet [École Normale Supérieure de Paris, from Sep 2020]
- Redha Gouicem [Université Pierre et Marie Curie, until Oct 2020]
- Darius Mercadier [Université Pierre et Marie Curie]
- Pierre Nigron [Inria]
- Lucas Serrano [Université Pierre et Marie Curie]

Technical Staff

- Michael Damien Carver [Université Pierre et Marie Curie, Engineer, until Mar 2020]
- Jeremie Dautheribes [Inria, Engineer, from Dec 2020]
- Corentin De Souza [Inria, Engineer]
- Rehan Malak [Inria, Engineer, from Dec 2020]
- Thierry Martinez [Inria, Engineer, from Oct 2020]
- Himadri Pandya [Inria, Engineer, from Dec 2020]

Interns and Apprentices

- Romeo La Spina [Inria, from Jun 2020 until Jul 2020]
- Adrien Le Berre [ENS Lyon, from Jun 2020 until Jul 2020]

Administrative Assistants

- Christine Anocq [Inria]
- Nelly Maloisel [Inria]

Visiting Scientist

- Beatriz Martins De Carvalho [LSD-Unicamp, until Apr 2020]

2 Overall objectives

The focus of Whisper is on how to develop (new) and improve (existing) infrastructure software. Infrastructure software (also called systems software) is the software that underlies all computing. Such software allows applications to access resources and provides essential services such as memory management, synchronization and inter-process interactions. Starting bottom-up from the hardware, examples include virtual machine hypervisors, operating systems, managed runtime environments, standard libraries, and browsers, which amount to the new operating system layer for Internet applications. For such software, efficiency and correctness are fundamental. Any overhead will impact the performance of all supported applications. Any failure will prevent the supported applications from running correctly. Since computing now pervades our society, with few paper backup solutions, correctness of software at all levels is critical. Formal methods are increasingly being applied to operating systems code in the research community [47, 53, 82]. Still, such efforts require a huge amount of manpower and a high degree of expertise which makes this work difficult to replicate in standard infrastructure-software development.

In terms of methodology, Whisper is at the interface of the domains of operating systems, software engineering and programming languages. Our approach is to combine the study of problems in the development of real-world infrastructure software with concepts in programming language design and implementation, *e.g.*, of domain-specific languages, and knowledge of low-level system behavior. A focus of our work is on providing support for legacy code, while taking the needs and competences of ordinary system developers into account.

We aim at providing solutions that can be easily learned and adopted by system developers in the short term. Such solutions can be tools, such as Coccinelle [2, 9, 10] for transforming C programs, or domain-specific languages such as Devil [6] and Bossa [8] for designing drivers and kernel schedulers. Due to the small size of the team, Whisper mainly targets operating system kernels and runtimes for programming languages. We put an emphasis on achieving measurable improvements in performance and safety in practice, and on feeding these improvements back to the infrastructure software developer community.

3 Research program

3.1 Program analysis

A fundamental goal of the research in the Whisper team is to elicit and exploit the knowledge found in existing code. To do this in a way that scales to a large code base, systematic methods are needed to infer code properties. We may build on either static [40, 41, 43] or dynamic analysis [61, 63, 68]. Static analysis consists of approximating the behavior of the source code from the source code alone, while dynamic analysis draws conclusions from observations of sample executions, typically of test cases. While dynamic analysis can be more accurate, because it has access to information about actual program behavior, obtaining adequate test cases is difficult. This difficulty is compounded for infrastructure software, where many, often obscure, cases must be handled, and external effects such as timing can have a significant impact. Thus, we expect to primarily use static analyses. Static analyses come in a range of flavors, varying in the extent to which the analysis is *sound*, *i.e.*, the extent to which the results are guaranteed to reflect possible run-time behaviors.

One form of sound static analysis is *abstract interpretation* [41]. In abstract interpretation, atomic terms are interpreted as sound abstractions of their values, and operators are interpreted as functions that soundly manipulate these abstract values. The analysis is then performed by interpreting the program in a compositional manner using these abstracted values and operators. Alternatively, *dataflow analysis* [52] iteratively infers connections between variable definitions and uses, in terms of local transition rules that describe how various kinds of program constructs may impact variable values. Schmidt has explored the relationship between abstract interpretation and dataflow analysis [76]. More recently, more general forms of symbolic execution [40] have emerged as a means of understanding complex code. In symbolic execution, concrete values are used when available, and these are complemented by constraints that are inferred from terms for which only partial information is available. Reasoning about these constraints is then used to prune infeasible paths, and obtain more precise results. A number of works apply symbolic execution to operating systems code [36, 37].

While sound approaches are guaranteed to give correct results, they typically do not scale to the very diverse code bases that are prevalent in infrastructure software. An important insight of Engler et al. [45] was that valuable information could be obtained even when sacrificing soundness, and that sacrificing soundness could make it possible to treat software at the scales of the kernels of the Linux or BSD operating systems. Indeed, for certain types of problems, on certain code bases, that may mostly follow certain coding conventions, it may mostly be safe to *e.g.*, ignore the effects of aliases, assume that variable values are unchanged by calls to unanalyzed functions, etc. Real code has to be understood by developers and thus cannot be too complicated, so such simplifying assumptions are likely to hold in practice. Nevertheless, approaches that sacrifice soundness also require the user to manually validate the results. Still, it is likely to be much more efficient for the user to perform a potentially complex manual analysis in a specific case, rather than to implement all possible required analyses and apply them everywhere in the code base. A refinement of unsound analysis is the CEGAR approach [39], in which a highly approximate analysis is complemented by a sound analysis that checks the individual reports of the approximate analysis, and then any errors in reasoning detected by the sound analysis are used to refine the approximate analysis. The CEGAR approach has been applied effectively on device driver code in tools developed at Microsoft [28]. The environment in which the driver executes, however, is still represented by possibly unsound approximations.

Going further in the direction of sacrificing soundness for scalability, the software engineering community has recently explored a number of approaches to code understanding based on techniques developed in the areas of natural language understanding, data mining, and information retrieval. These approaches view code, as well as other software-reated artifacts, such as documentation and postings on mailing lists, as bags of words structured in various ways. Statistical methods are then used to collect words or phrases that seem to be highly correlated, independently of the semantics of the program constructs that connect them. The obliviousness to program semantics can lead to many false positives (invalid conclusions) [58], but can also highlight trends that are not apparent at the low level of individual program statements. We have previously explored combining such statistical methods with more traditional static analysis in identifying faults in the usage of constants in Linux kernel code [56].

3.2 Domain Specific Languages

Writing low-level infrastructure code is tedious and difficult, and verifying it is even more so. To produce non-trivial programs, we could benefit from moving up the abstraction stack to enable both programming and proving as quickly as possible. Domain-specific languages (DSLs), also known as *little languages*, are a means to that end [7] [64].

Traditional approach. Using little languages to aid in software development is a tried-and-trusted technique [78] by which programmers can express high-level ideas about the system at hand and avoid writing large quantities of formulaic C boilerplate.

This approach is typified by the Devil language for hardware access [6]. An OS programmer describes the register set of a hardware device in the high-level Devil language, which is then compiled into a library providing C functions to read and write values from the device registers. In doing so, Devil frees the programmer from having to write extensive bit-manipulation macros or inline functions to map between the values the OS code deals with, and the bit-representation used by the hardware: Devil generates code to do this automatically.

However, DSLs are not restricted to being "stub" compilers from declarative specifications. The Bossa language [8] is a prime example of a DSL involving imperative code (syntactically close to C) while offering a high-level of abstraction. This design of Bossa enables the developer to implement new process scheduling policies at a level of abstraction tailored to the application domain.

Conceptually, a DSL both abstracts away low-level details and justifies the abstraction by its semantics. In principle, it reduces development time by allowing the programmer to focus on high-level abstractions. The programmer needs to write less code, in a language with syntax and type checks adapted to the problem at hand, thus reducing the likelihood of errors.

Embedding DSLs. The idea of a DSL has yet to realize its full potential in the OS community. Indeed, with the notable exception of interface definition languages for remote procedure call (RPC) stubs, most

OS code is still written in a low-level language, such as C. Where DSL code generators are used in an OS, they tend to be extremely simple in both syntax and semantics. We conjecture that the effort to implement a given DSL usually outweighs its benefit. We identify several serious obstacles to using DSLs to build a modern OS: specifying what the generated code will look like, evolving the DSL over time, debugging generated code, implementing a bug-free code generator, and testing the DSL compiler.

Filet-o-Fish (FoF) [42] addresses these issues by providing a framework in which to build correct code generators from semantic specifications. This framework is presented as a Haskell library, enabling DSL writers to *embed* their languages within Haskell. DSL compilers built using FoF are quick to write, simple, and compact, but encode rigorous semantics for the generated code. They allow formal proofs of the run-time behavior of generated code, and automated testing of the code generator based on randomized inputs, providing greater test coverage than is usually feasible in a DSL. The use of FoF results in DSL compilers that OS developers can quickly implement and evolve, and that generate provably correct code. FoF has been used to build a number of domain-specific languages used in Barrelfish, [29] an OS for heterogeneous multicore systems developed at ETH Zurich.

The development of an embedded DSL requires a few supporting abstractions in the host programming language. FoF was developed in the purely functional language Haskell, thus benefiting from the type class mechanism for overloading, a flexible parser offering convenient syntactic sugar, and purity enabling a more algebraic approach based on small, composable combinators. Object-oriented languages – such as Smalltalk [46] and its descendant Pharo [33] – or multi-paradigm languages – such as the Scala programming language [66] – also offer a wide range of mechanisms enabling the development of embedded DSLs. Perhaps suprisingly, a low-level imperative language – such as C – can also be extended so as to enable the development of embedded compilers [31].

Certifying DSLs. Whilst automated and interactive software verification tools are progressively being applied to larger and larger programs, we have not yet reached the point where large-scale, legacy software – such as the Linux kernel – could formally be proved "correct". DSLs enable a pragmatic approach, by which one could realistically strengthen a large legacy software by first narrowing down its critical component(s) and then focus our verification efforts onto these components.

Dependently-typed languages, such as Coq or Idris, offer an ideal environment for embedding DSLs [38, 34] in a unified framework enabling verification. Dependent types support the type-safe embedding of object languages and Coq's mixfix notation system enables reasonably idiomatic domain-specific concrete syntax. Coq's powerful abstraction facilities provide a flexible framework in which to not only implement and verify a range of domain-specific compilers [42], but also to combine them, and reason about their combination.

Working with many DSLs optimizes the "horizontal" compositionality of systems, and favors reuse of building blocks, by contrast with the "vertical" composition of the traditional compiler pipeline, involving a stack of comparatively large intermediate languages that are harder to reuse the higher one goes. The idea of building compilers from reusable building blocks is a common one, of course. But the interface contracts of such blocks tend to be complex, so combinations are hard to get right. We believe that being able to write and verify formal specifications for the pieces will make it possible to know when components can be combined, and should help in designing good interfaces.

Furthermore, the fact that Coq is also a system for formalizing mathematics enables one to establish a close, formal connection between embedded DSLs and non-trivial domain-specific models. The possibility of developing software in a truly "model-driven" way is an exciting one. Following this methodology, we have implemented a certified compiler from regular expressions to x86 machine code [51]. Interestingly, our development crucially relied on an existing Coq formalization, due to Braibant and Pous, [35] of the theory of Kleene algebras.

While these individual experiments seem to converge toward embedding domain-specific languages in rich type theories, further experimental validation is required. Indeed, Barrelfish is an extremely small software compared to the Linux kernel. The challenge lies in scaling this methodology up to large software systems. Doing so calls for a unified platform enabling the development of a myriad of DSLs, supporting code reuse across DSLs as well as providing support for mechanically-verified proofs.

3.3 Research direction: Tools for improving legacy infrastructure software

A cornerstone of our work on legacy infrastructure software is the Coccinelle program matching and transformation tool for C code. Coccinelle has been in continuous development since 2005. Today, Coccinelle is extensively used in the context of Linux kernel development, as well as in the development of other software, such as wine, python, kvm, and systemd. Currently, Coccinelle is a mature software project, and no research is being conducted on Coccinelle itself. Instead, we leverage Coccinelle in other research projects [30, 32, 67, 69, 74, 75, 77, 62, 57], both for code exploration, to better understand at a large scale problems in Linux development, and as an essential component in tools that require program matching and transformation. The continuing development and use of Coccinelle is also a source of visibility in the Linux kernel developer community. We submitted the first patches to the Linux kernel based on Coccinelle in 2007. Since then, over 5500 patches have been accepted into the Linux kernel based on the use of Coccinelle, including around 3000 by over 500 developers from outside our research group.

Our recent work has focused on driver porting. Specifically, we have considered the problem of porting a Linux device driver across versions, particularly backporting, in which a modern driver needs to be used by a client who, typically for reasons of stability, is not able to update their Linux kernel to the most recent version. When multiple drivers need to be backported, they typically need many common changes, suggesting that Coccinelle could be applicable. Using Coccinelle, however, requires writing backporting transformation rules. In order to more fully automate the backporting (or symmetrically forward porting) process, these rules should be generated automatically. We have carried out a preliminary study in this direction with David Lo of Singapore Management University; this work, published at ICSME 2016 [80], is limited to a port from one version to the next one, in the case where the amount of change required is limited to a single line of code. Whisper has been awarded an ANR PRCI grant to collaborate with the group of David Lo on scaling up the rule inference process and proposing a fully automatic porting solution.

3.4 Research direction: developing infrastructure software using Domain Specific Languages

We wish to pursue a *declarative* approach to developing infrastructure software. Indeed, there exists a significant gap between the high-level objectives of these systems and their implementation in low-level, imperative programming languages. To bridge that gap, we propose an approach based on domain-specific languages (DSLs). By abstracting away boilerplate code, DSLs increase the productivity of systems programmers. By providing a more declarative language, DSLs reduce the complexity of code, thus the likelihood of bugs.

Traditionally, systems are built by accretion of several, independent DSLs. For example, one might use Devil [6] to interact with devices, Bossa [8] to implement the scheduling policies. However, much effort is duplicated in implementing the back-ends of the individual DSLs. Our long term goal is to design a unified framework for developing and composing DSLs, following our work on Filet-o-Fish [42]. By providing a single conceptual framework, we hope to amortize the development cost of a myriad of DSLs through a principled approach to reusing and composing them.

Beyond the software engineering aspects, a unified platform brings us closer to the implementation of mechanically-verified DSLs. Using the Coq proof assistant as an x86 macro-assembler [51] is a step in that direction, which belongs to a larger trend of hosting DSLs in dependent type theories [34, 38, 65]. A key benefit of those approaches is to provide – by construction – a formal, mechanized semantics to the DSLs thus developed. This semantics offers a foundation on which to base further verification efforts, whilst allowing interaction with non-verified code. We advocate a methodology based on incremental, piece-wise verification. Whilst building fully-certified systems from the top-down is a worthwhile endeavor [53], we wish to explore a bottom-up approach by which one focuses first and foremost on crucial subsystems and their associated properties.

Our current work on DSLs has two complementary goals: (i) the design of a unified framework for developing and composing DSLs, following our work on Filet-o-Fish, and (ii) the design of domain-specific languages for domains where there is a critical need for code correctness, and corresponding methodologies for proving properties of the run-time behavior of the system.

4 Application domains

4.1 Linux

Linux is an open-source operating system that is used in settings ranging from embedded systems to supercomputers. The most recent release of the Linux kernel, v4.14, comprises over 16 million lines of code, and supports 30 different families of CPU architectures, around 50 file systems, and thousands of device drivers. Linux is also in a rapid stage of development, with new versions being released roughly every 2.5 months. Recent versions have each incorporated around 13,500 commits, from around 1500 developers. These developers have a wide range of expertise, with some providing hundreds of patches per release, while others have contributed only one. Overall, the Linux kernel is critical software, but software in which the quality of the developed source code is highly variable. These features, combined with the fact that the Linux community is open to contributions and to the use of tools, make the Linux kernel an attractive target for software researchers. Tools that result from research can be directly integrated into the development of real software, where it can have a high, visible impact.

Starting from the work of Engler et al. [44], numerous research tools have been applied to the Linux kernel, typically for finding bugs [43, 60, 70, 79] or for computing software metrics [49, 81]. In our work, we have studied generic C bugs in Linux code [10], bugs in function protocol usage [54, 55], issues related to the processing of bug reports [73] and crash dumps [48], and the problem of backporting [69, 80], illustrating the variety of issues that can be explored on this code base. Unique among research groups working in this area, we have furthermore developed numerous contacts in the Linux developer community. These contacts provide insights into the problems actually faced by developers and serve as a means of validating the practical relevance of our work.

4.2 Device Drivers

Device drivers are essential to modern computing, to provide applications with access, via the operating system, to physical devices such as keyboards, disks, networks, and cameras. Development of new computing paradigms, such as the internet of things, is hampered because device driver development is challenging and error-prone, requiring a high level of expertise in both the targeted OS and the specific device. Furthermore, implementing just one driver is often not sufficient; today's computing landscape is characterized by a number of OSes, *e.g.*, Linux, Windows, MacOS, BSD and many real time OSes, and each is found in a wide range of variants and versions. All of these factors make the development, porting, backporting, and maintenance of device drivers a critical problem for device manufacturers, industry that requires specific devices, and even for ordinary users.

The last fifteen years have seen a number of approaches directed towards easing device driver development. Réveillère, who was supervised by G. Muller, proposes Devil [6], a domain-specific language for describing the low-level interface of a device. Chipounov *et al.* propose RevNic, [37] a template-based approach for porting device drivers from one OS to another. Ryzhyk *et al.* propose Termite, [71, 72] an approach for synthesizing device driver code from a specification of an OS and a device. Currently, these approaches have been successfully applied to only a small number of toy drivers. Indeed, Kadav and Swift [50] observe that these approaches make assumptions that are not satisfied by many drivers; for example, the assumption that a driver involves little computation other than the direct interaction between the OS and the device. At the same time, a number of tools have been developed for finding bugs in driver code. These tools include SDV [28], Coverity [44], CP-Miner, [59] PR-Miner [60], and Coccinelle [9]. These approaches, however, focus on analyzing existing code, and do not provide guidelines on structuring drivers.

In summary, there is still a need for a methodology that first helps the developer understand the software architecture of drivers for commonly used operating systems, and then provides tools for the maintenance of existing drivers.

5 Social and environmental responsibility

5.1 Impact of research results

Social responsability The quarantine measures applied around the world in 2020 led to the massive utilization of video conference systems such as Zoom, Jitsi, and BigBlueButton. This was a major challenge for the robustness of these systems. The most obvious solution was the massive overprovisioning of the resources allocated to such systems.

Massive overprovisioning of video conferencing systems is not a practical solution in the long term. Very little previous research has studied the efficiency of video conferencing systems. In the context of an Inria Covid-19 project, we have carried out a study of the resource requirements of the well-known open-source video conferencing system Jitsi, considering the CPU, memory, disk, network, and micro-architecture.

We have deployed a Jitsi server in Rennes, that has been extensively used for real video conferences and for artificial stress tests. In practice, however, we have not been able to overload the server, indicating that problems are instead due to lack of bandwidth. Even with local clients, it was not possible to overload the server. We did find that Jitsi works better in Chrome than in Mozilla, because part of the protocol that it uses was developed by Google. A dedicated client would likely be even more efficient, but this goes beyond the scope of the project.

Environmental responsability The Whisper team is actively pursuing research on process scheduling for the Linux kernel, supported in part by a donation from Oracle. A current area of interest is concentrating threads on fewer cores in a multicore setting, in order to both reduce the execution time and to increase the number of cores that can enter a deep idle state. A first work in this direction was published at USENIX ATC 2020 [15].

6 New software and platforms

6.1 New software

6.1.1 Coccinelle

Keywords: Code quality, Evolution, Infrastructure software

Functional Description: Coccinelle is a tool for code search and transformation for C programs. It has been extensively used for bug finding and evolutions in Linux kernel code.

URL: http://coccinelle.lip6.fr

Authors: Gilles Muller, Julia Lawall, Nicolas Palix, Rene Rydhof Hansen, Xavier Clerc

Contact: Julia Lawall

Participants: Gilles Muller, Julia Lawall, Nicolas Palix, Rene Rydhof Hansen, Thierry Martinez

Partners: LIP6, IRILL

6.1.2 Prequel

Keywords: Code search, Git

Scientific Description: The commit history of a code base such as the Linux kernel is a gold mine of information on how evolutions should be made, how bugs should be fixed, etc. Nevertheless, the high volume of commits available and the rudimentary filtering tools provided mean that it is often necessary to wade through a lot of irrelevant information before finding example commits that can help with a specific software development problem. To address this issue, we propose Prequel (Patch Query Language), which brings the descriptive power of code matching to the problem of querying a commit history.

Functional Description: Prequel is a tool for searching for complex patterns in the commits of software managed using git.

URL: http://prequel-pql.gforge.inria.fr/

Contact: Julia Lawall

Participants: Gilles Muller, Julia Lawall

Partners: LIP6, IRILL

6.1.3 Usuba

Keywords: Cryptography, Optimizing compiler, Synchronous Language

Functional Description: Usuba is a programming language for specifying block ciphers as well as a bitslicing compiler, for producing high-throughput and secure code.

URL: https://github.com/DadaIsCrazy/usuba/

Publication: hal-01657259

Contacts: Pierre-Evariste Dagand, Darius Mercadier

6.1.4 SchedDisplay

Keywords: Linux kernel, Scheduling, Multicore

Functional Description: SchedDisplay is a visualization tool for SchedLog, a custom ring buffer collecting scheduling events in the Linux kernel. SchedDisplay allows kernel developers to analyze the behavior of the Linux scheduler while running a multicore application.

Release Contributions: First version released as part of a Demo made during the 10th PLOS workshop: https://ess.cs.uni-osnabrueck.de/workshops/plos/2019/program.php

URL: https://gitlab.inria.fr/gmuller/scheddisplay

Contact: Gilles Muller

Partner: Oracle Labs

7 New results

7.1 Software engineering for infrastructure software

Our main work in this area has been on the problem of inferring transformation rules from change examples for the purpose of fully automating large-scale evolutions in infrastructure software. This work has been carried out in collaboration with David Lo and Lingxiao Jiang of Singapore Management University, in the context of the ANR ITrans project. We have also developed approaches to find several specific kinds of bugs in the Linux kernel, and considered the more general problems of patch classification and code review, which are a well-known challenges for large-scale open-source software, of which the Linux kernel is a preeminent example.

7.1.1 Transformation rule inference

In a large software system such as the Linux kernel, there is a continual need for large-scale changes across many source files, triggered by new needs or refined design decisions. In the paper "SPINFER: Inferring Semantic Patches for the Linux Kernel", published at USENIX ATC 2020 [21], we propose to ease such changes by suggesting transformation rules to developers, inferred automatically from a collection of examples. Our approach can help automate large-scale changes as well as help understand existing large-scale changes, by highlighting the various cases that the developer who performed the changes has taken into account. We have implemented our approach as a tool, Spinfer. We evaluate Spinfer on a range of challenging large-scale changes from the Linux kernel and obtain rules that achieve 86% precision and 69% recall on average. Lucas Serrano defended his PhD thesis [27] on the design of Spinfer on November 25, 2020.

As the Android API evolves, some API methods may be deprecated, to be eventually removed. App developers must thus keep their apps up-to-date, to ensure that the apps work in both older and newer Android versions. Currently, AppEvolve is the state-of-the-art approach to automate such updates, and it has been shown to be quite effective. Still, the number of experiments reported is moderate, involving only API usage updates in 41 usage locations. In the paper "Automated Deprecated-API Usage Update for Android Apps: How Far Are We?", published in the reproducibility (RENE) track at SANER 2020 [22], we replicate the evaluation of AppEvolve and assess whether its effectiveness is generalizable. Given the set of APIs on which AppEvolve has been evaluated, we test AppEvolve on other mobile apps that use the same APIs. Our experiments show that AppEvolve fails to generate applicable updates for 81% of our dataset, even though the relevant knowledge for correct API updates is available in the examples. We first categorize the limitations of AppEvolve that lead to these failures. We then propose a mitigation strategy that solves 86% of these failures by a simple refactoring of the app code to better resemble the code in the examples. The refactoring usually involves assigning the target API method invocation and the arguments of the target API method into variables. Indeed, we have also seen such transformations in the dataset distributed with the AppEvolve replication package, as compared to the original source code from which this dataset is derived. Based on these findings, we propose some promising future directions.

The work on deprecated Android APIs was continued in the paper "Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example", published at ICPC 2020 [16]. While the state-of-the-art AppEvolve relies on having before-and after-update examples to learn from, we propose an approach named CocciEvolve that performs such updates using only a single after-update example. CocciEvolve learns edits by extracting the relevant update to a block of code from an after-update example. From preliminary experiments, we find that CocciEvolve can successfully perform 96 out of 112 updates, with a success rate of 85%.

7.1.2 Bug finding and code understanding

Atomic context is an execution state of the Linux kernel, in which kernel code monopolizes a CPU core. In this state, the Linux kernel may only perform operations that cannot sleep, as otherwise a system hang or crash may occur. We refer to this kind of concurrency bug as a sleep-in-atomic-context (SAC) bug. In practice, SAC bugs are hard to find, as they do not cause problems in all executions. In the paper "Effective Detection of Sleep-in-Atomic-Context Bugs in the Linux Kernel", published at ACM TOCS [11], we propose a practical static approach named DSAC, to effectively detect SAC bugs in the Linux kernel. DSAC uses three key techniques: (1) a summary-based analysis to identify the code that may be executed in atomic context, (2) a connection-based alias analysis to identify the set of functions referenced by a function pointer, and (3) a path-check method to filter out repeated reports and false bugs. We evaluate DSAC on Linux 4.17, and find 1159 SAC bugs. We manually check all the bugs, and find that 1068 bugs are real. We have randomly selected 300 of the real bugs and sent them to kernel developers. 220 of these bugs have been confirmed, and 51 of our patches fixing 115 bugs have been applied. This work was done in collaboration with Jia-Ju Bai of Tsinghua University.

The increasing adoption of the Linux kernel has been sustained by a large and constant maintenance effort, performed by a wide and heterogeneous base of contributors. One important problem that maintainers face in any code base is the rapid understanding of complex data structures. The Linux

kernel is written in the C language, which enables the definition of arbitrarily uninformative datatypes, via the use of casts and pointer arithmetic, of which doubly linked lists are a prominent example. In the paper "The Impact of Generic Data Structures: Decoding the Role of Lists in the Linux Kernel", published at ASE 2020 [23], we explore the advantages and disadvantages of such lists, for expressivity, for code understanding, and for code reliability. Based on our observations, we have developed a toolset that includes inference of descriptive list types and a tool for list visualization. Our tools identify more than 10,000 list fields and variables in recent Linux kernel releases and succeeds in typing 90%. We show how these tools could have been used to detect previously fixed bugs and identify 6 new ones. This work was done in collaboration with Nic Volanschi of Inria-Bordeaux.

7.1.3 Patch classification and code review

Existing work on software patches often use features specific to a single task. These works often rely on manually identified features, and human effort is required to identify these features for each task. In the paper "CC2Vec: Distributed Representations of Code Changes", published at ICSE 2020 [17], we propose CC2Vec, a neural network model that learns a representation of code changes guided by their accompanying log messages, which represent the semantic intent of the code changes. CC2Vec models the hierarchical structure of a code change with the help of the attention mechanism and uses multiple comparison functions to identify the differences between the removed and added code. To evaluate if CC2Vec can produce a distributed representation of code changes that is general and useful for multiple tasks on software patches, we use the vectors produced by CC2Vec for three tasks: log message generation, bug fixing patch identification, and just-in-time defect prediction. In all tasks, the models using CC2Vec outperform the state-of-the-art techniques. This work was done in collaboration with David Lo and Lingxiao Jiang at Singapore Management University as part of the ITrans project.

Code review is an important part of the development of any software project. Recently, many open source projects have begun practicing lightweight and tool-based code review (a.k.a modern code review) to make the process simpler and more efficient. However, those practices still require reviewers, of which there may not be sufficiently many to ensure timely decisions. In the paper "Expanding the Number of Reviewers in Open-Source Projects by Recommending Appropriate Developers", published at ICSME 2020 [14], we propose a recommender-based approach to be used by open-source projects to increase the number of reviewers from among the appropriate developers. We first motivate our approach by an exploratory study of nine projects hosted on GitHub and Gerrit. Secondly, we build the recommender system itself, which, given a code change, initially searches for relevant reviewers based on similarities between the reviewing history and the files affected by the change, and then augments this set with developers who have a similar development history as these reviewers but have little or no relevant reviewing experience. To make these recommendations, we rely on collaborative filtering, and more precisely, on matrix factorization. Our evaluation shows that all nine projects could benefit from our system by using it both to get recommendations of previous reviewers and to expand their number from among the appropriate developers. This work was done in collaboration with Reda Bendraou of LIP6.

7.2 Programming after the end of Moore's law

The end of Moore's law is a wake-up call that resonates across Computer Science at large. We are now firmly in an era of custom hardware design, as witnessed by the diversity of system-on-chip (SoC) and specialized processing units – such as graphics processing units (GPUs), tensor processing unit (TPUs) or programmable network adapters, to name but a few. This trend is justified by the existence of niche application domains (graphic processing, linear algebra, packet processing, etc.) that greatly benefit from specialized hardware. Faced with the imminent explosion of the number of niche applications and niche architectures, we are still grasping for a programming model that would accommodate this diversity.

The Usuba project is an exploratory effort in that direction. We chose a niche application domain (symmetric cryptographic algorithms), a specialized execution platform (Single Instruction Multiple Data, SIMD) processors and we set out to design a programming language faithfully describing our application domain as well as an optimizing compiler efficiently exploiting our target execution platform.

The design of software countermeasures against active and passive adversaries is a challenging problem that has been addressed by many authors in recent years. The proposed solutions adopt a theoretical

foundation (such as a leakage model) but often do not offer concrete reference implementations to validate the foundation. Contributing to the experimental dimension of this body of work, in the paper "Custom Instruction Support for Modular Defense against Side-channel and Fault Attacks", published at COSADE 2020 [18], we propose a customized processor called SKIVA that supports experiments with the design of countermeasures against a broad range of implementation attacks. Based on bitslice programming and recent advances in the literature, SKIVA offers a flexible and modular combination of countermeasures against power-based and timing-based side-channel leakage and fault injection. Multiple configurations of side-channel protection and fault protection enable the programmer to select the desired number of shares and the desired redundancy level for each slice. Recurring and security-sensitive operations are supported in hardware through custom instruction-set extensions. The new instructions support bitslicing, secret-share generation, redundant logic computation, and fault detection. We demonstrate and analyze multiple versions of AES from a side-channel analysis and a fault-injection perspective, in addition to providing a detailed performance evaluation of the protected designs. To our knowledge, this is the first validated end-to-end implementation of a modular bitslice-oriented countermeasure.

Cryptographic implementations deployed in real world devices often aim at (provable) security against the powerful class of side-channel attacks while keeping reasonable performances. Last year at Asiacrypt, a new formal verification tool named tightPROVE was put forward to exactly determine whether a masked implementation is secure in the well-deployed probing security model for any given security order t. Also recently, a compiler named Usuba was proposed to automatically generate bitsliced implementations of cryptographic primitives. The paper "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations", published at EuroCrypt 2020 [12] goes one step further in the security and performances achievements with a new automatic tool named Tornado. In a nutshell, from the high-level description of a cryptographic primitive, Tornado produces a functionally equivalent bitsliced masked implementation at any desired order proven secure in the probing model, but additionally in the so-called register probing model which much better fits the reality of software implementations. This framework is obtained by the integration of Usuba with tightPROVE+, which extends tightPROVE with the ability to verify the security of implementations in the register probing model and to fix them with inserting refresh gadgets at carefully chosen locations accordingly. We demonstrate Tornado on the lightweight cryptographic primitives selected to the second round of the NIST competition and which somehow claimed to be masking friendly. It advantageously displays performances of the resulting masked implementations for several masking orders and prove their security in the register probing model.

Transiently-powered systems featuring non-volatile memory as well as external peripherals enable the development of new low-power sensor applications. However, as programmers, we are ill-equipped to reason about systems where power failures are the norm rather than the exception. A first challenge consists in being able to capture all the volatile state of the application – external peripherals included – to ensure progress. A second, more fundamental, challenge consists in specifying how power failures may interact with peripheral operations. In the paper "Intermittent Computing with Peripherals, Formally Verified", published at LCTES 2020 [13], we propose a formal specification of intermittent computing with peripherals, an axiomatic model of interrupt-based checkpointing as well as its proof of correctness, machine-checked in the Coq proof assistant. We also illustrate our model with several systems proposed in the literature.

On November 20, 2020, Darius Mercadier defended his PhD thesis in this area [26].

7.3 Support for multicore machines

Our work in this area involves on the one hand formal proofs of scheduler properties, and on the other hand performance improvements for multicore applications. This work results from our collaboration with Oracle and with our Inria Associate Team at the University of Sydney.

Recent research and bug reports have shown that work conservation, the property that a core is idle only if no other core is overloaded, is not guaranteed by Linux's CFS or FreeBSD's ULE multicore schedulers. Indeed, multicore schedulers are challenging to specify and verify: they must operate under stringent performance requirements, while handling very large numbers of concurrent operations on threads. As a consequence, the verification of correctness properties of schedulers has not yet

been considered. In the paper "Provable Multicore Schedulers with Ipanema: Application to Work Conservation", published at EuroSys 2020 [20], we propose an approach, based on a domain-specific language and theorem provers, for developing schedulers with provable properties. We introduce the notion of concurrent work conservation (CWC), a relaxed definition of work conservation that can be achieved in a concurrent system where threads can be created, unblocked and blocked concurrently with other scheduling events. We implement several scheduling policies, inspired by CFS and ULE. We show that our schedulers obtain the same level of performance as production schedulers, while concurrent work conservation is satisfied. Virginia Aponte contributed substantially to the concurrent work conservation proof during her delegation in the Whisper team.

In modern server CPUs, individual cores can run at different frequencies, which allows for fine-grained control of the performance/energy tradeoff. Adjusting the frequency, however, incurs a high latency. This can lead to a problem of *frequency inversion*, whereby the Linux scheduler places a newly active thread on an idle core that takes dozens to hundreds of milliseconds to reach a high frequency, just before another core already running at a high frequency becomes idle. In the paper "Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance", published at USENIX ATC 2020 [15], we first illustrate the significant performance overhead of repeated frequency inversion through a case study of scheduler behavior during the compilation of the Linux kernel on an 80-core Intel® Xeon-based machine. Following this, we propose two strategies to reduce the likelihood of frequency inversion in the Linux scheduler. When benchmarked over 60 diverse applications on the Intel® Xeon, the better performing strategy, S_{move}, improves performance by more than 5% (at most 56% with no energy overhead) for 23 applications, and worsens performance by more than 5% (at most 8%) for only 3 applications. On a 4-core AMD Ryzen we obtain performance improvements up to 56%.

On October 23, 2020, Redha Gouicem defended his PhD thesis in this area [25].

8 Bilateral contracts and grants with industry

8.1 Bilateral contracts with industry

- Orange Labs, 2019-2022, 30 000 euros. The purpose of this contract is to design application-specific proxies so as to speed up network services. The PhD of Yoann Ghigoff is supported by a CIFRE fellowship as part of this contract.

- DGA-Inria, 2019-2022, 60 000 euros. The purpose of this PhD grant is to develop a high-performance, certified packet processing system. The PhD of Pierre Nigron is supported by this grant.

8.2 Bilateral grants with industry

- Oracle, 2020-2021, 100 000 dollars.

Operating system schedulers are often a performance bottleneck on multicore architectures because in order to scale, schedulers cannot make optimal decisions and instead have to rely on heuristics. Detecting that performance degradation comes from the scheduler level is extremely difficult because the issue has not been recognized until recently, and with traditional profilers, both the application and the scheduler affect the monitored metrics in the same way.

The first objective of this project is to produce a profiler that makes it possible to find out whether a bottleneck during application runtime is caused by the application itself, by suboptimal OS scheduler behavior, or by a combination of the two. It will require understanding, analyzing and classifying performance bottlenecks that are caused by schedulers, and devising ways to detect them and to provide enough information for the user to understand the root cause of the issue. Following this, the second objective of this project is to use the profiler to better understand which kinds of workloads suffer from poor scheduling, and to propose new algorithms, heuristics and/or a new scheduler design that will improve the situation. Finally, the third contribution will be a methodology that makes it possible to track scheduling bottlenecks in a specific workload using the profiler, to understand them, and to fix them either at the application or at the scheduler level. We believe that the combination of these three contributions will make it possible to fully harness the power of multicore architectures for any workload.

As part of this project, we have already identified frequency scaling and the “fork/wait” paradigm as a source of inefficiency in modern multicore machines. These first results were published in the PLOS workshop that is held together with SOSP. The diagnosis of the problem was possible thanks to the *SchedLog* and *SchedDisplay* tools that we developed as part of this project.

9 Partnerships and cooperations

9.1 International initiatives

9.1.1 Inria associate team not involved in an IIL

CSG

Title: *Concurrent Systems Group*

Duration: 2019 - 2022

Coordinator: Gilles Muller

Partners:

- University of Sydney (Australia)

Inria contact: Gilles Muller

Summary: The initial topic of this cooperation is the development of proved multicore schedulers. Over the last two years, we have explored a novel approach based on the identification of key scheduling abstractions and the realization of these abstractions as a Domain-Specific Language (DSL), Ipanema. We have introduced a concurrency model that relies on execution of scheduling events in mutual execution locally on a core, but that still permits reading the state of other cores without requiring locks.

In the three next years, we will leverage on our existing results towards the following directions: (i) Better understanding of what should the best scheduler for a given multicore application, (ii) Proving the correctness of the C code generated from the DSL policy and of the Ipanema abstract machine, (iii) Extend the Ipanema DSL to the domain of I/O request scheduling, (iv) Design of a provable complete concurrent kernel.

In 2020, we published two papers as part of this project: "Provable multicore schedulers with Ipanema: application to work conservation" at EuroSys 2020 [20] on the design of the Ipanema DSL for scheduling policies and its use in proving work conservation using Why3, and "Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance" at USENIX ATC [15] on a small modification to the Linux kernel scheduler to better exploit variable CPU frequencies. Additionally, we have started a new topic related to the documentation and verification of race conditions in the Linux kernel. Josselin Giet spent 6 months at the University of Sydney working in this area, under the supervision of Baptiste Lepers. At the end of 2020, we hired Rehan Malak as an engineer, funded by the ANR VeriAmos project, to continue the work on verifying scheduling policies, and Himadri Pandya as an engineer, funded by a gift from Oracle, to continue the work on the relationship between process scheduling and CPU frequencies.

9.1.2 Inria international partners

Informal international partners

- Julia Lawall collaborates with Sergey Starotelov of the Polzunov Altai State Technical University in Barnaul Russia on automating forward and backward portingof Linux device drivers.
- Julia Lawall collaborates with Jia-Ju Bai of Tsinghua University on issues around bug finding in Linux device drivers.

9.2 National initiatives

9.2.1 ANR

- **ITrans** - awarded in 2016, duration 2017 - 2020
- Members: LIP6 (Whisper), David Lo (Singapore Management University)
- Coordinator: Julia Lawall
- Whisper members: Julia Lawall, Gilles Muller, Lucas Serrano, Van-Anh Nguyen
- Funding: ANR PRCI, 287,820 euros.
- Objectives:

  Large, real-world software must continually change, to keep up with evolving requirements, fix bugs, and improve performance, maintainability, and security. This rate of change can pose difficulties for clients, whose code cannot always evolve at the same rate. This project will target the problems of *forward porting*, where one software component has to catch up to a code base with which it needs to interact, and *back porting*, in which it is desired to use a more modern component in a context where it is necessary to continue to use a legacy code base, focusing on the context of Linux device drivers. In this project, we will take a *history-guided source-code transformation-based* approach, which automatically traverses the history of the changes made to a software system, to find where changes in the code to be ported are required, gathers examples of the required changes, and generates change rules to incrementally back port or forward port the code. Our approach will be a success if it is able to automatically back and forward port a large number of drivers for the Linux operating system to various earlier and later versions of the Linux kernel with high accuracy while requiring minimal developer effort. This objective is not achievable by existing techniques.

- **VeriAmos** - awarded in 2018, duration 2018 - 2021
- Members: Inria (Antique, Whisper), UGA (Erods)
- Coordinator: Xavier Rival
- Whisper members: Julia Lawall, Gilles Muller
- Funding: ANR, 121,739 euros.
- Objectives:

  General-purpose Operating Systems, such as Linux, are increasingly used to support high-level functionalities in the safety-critical embedded systems industry with usage in automotive, medical and cyber-physical systems. However, it is well known that general purpose OSes suffer from bugs. In the embedded systems context, bugs may have critical consequences, even affecting human life. Recently, some major advances have been done in verifying OS kernels, mostly employing interactive theorem-proving techniques. These works rely on the formalization of the programming language semantics, and of the implementation of a software component, but require significant human intervention to supply the main proof arguments. The VeriAmos project will attack this problem by building on recent advances in the design of domain-specific languages and static analyzers for systems code. We will investigate whether the restricted expressiveness and the higher level of abstraction provided by the use of a DSL will make it possible to design static analyzers that can statically and fully automatically verify important classes of semantic properties on OS code, while retaining adequate performance of the OS service. As a specific use-case, the project will target I/O scheduling components.

10 Dissemination

10.1 Promoting scientific activities

10.1.1 Scientific events: organisation

General chair, scientific chair

- Julia Lawall organized the 20th meeting of the IFIP working group 2.11 on program generation at LIP6 in February 2020.

Member of the organizing committees

- Julia Lawall is a member of the steering committee of the conference Automated Software Engineering (since 2019)
- Julia Lawall is a member of the steering committee of Microsoft's Developing Secure Systems Summit (since 2020)
- Gilles Muller is the president of the steering committee of Compas.

10.1.2 Scientific events: selection

Chair of conference program committees

- Julia Lawall was a program chair of Compas 2020.

Member of the conference program committees

- Julia Lawall was a member of the program committee of ASE 2020.
- Julia Lawall was a member of the program committee of USENIX ATC 2020.
- Julia Lawall was a member of the program committee of GPCE 2020.
- Julia Lawall was a member of the program committee of ISEC 2020.
- Julia Lawall was a member of the program committee of BENEVOL 2020.
- Julia Lawall was a member of the program committee of the Visions and Reflections track of FSE 2020.
- Julia Lawall was a member of the program committee of the NIER track of ICSE 2020.
- Julia Lawall was a member of the program committee of the doctoral symposium of ISSTA 2020.
- Gilles Muller was a member of the program committee of USENIX ATC 2020.
- Gilles Muller was a member of the program committee of VEE 2020.
- Gilles Muller was a member of the program committee of NSDI 2021.

10.1.3 Journal

- Julia Lawall reviewed papers for Science of Computer Programming, Information and Software Technology, and IEEE Transactions on Software Engineering.

Member of the editorial boards

- Julia Lawall is a member of the editorial board of Science of Computer Programming.

10.1.4 Invited talks

- Julia Lawall gave at talk at the The 62nd CREST Open Workshop - Automated Program Repair and Genetic Improvement on "SPINFER: Inferring Software Maintenance Rules for the Linux Kernel" (January 20, 2020)
- Julia Lawall gave a talk in the Inria-Rennes CIDRE team on "Coccinelle, Prequel, and Spinfer: Automating Summarization and Application of Code Evolutions in the Linux Kernel" (January 31, 2020)
- Julia Lawall gave a talk at the CEA on Ipanema and Why3 (March 3, 2020)

10.1.5 Scientific expertise

- Julia Lawall is a member of the advisory board of Software Heritage.
- Julia Lawall was a member of the jury for a Maître de conférences position at LIP/Lyon1.
- Gilles Muller was a member of the evaluation committee for the Needham award 2020.
- Gilles Muller was a member of the evaluation committee for the DSN-2020 rising star award.

10.2 Teaching - Supervision - Juries

10.2.1 Teaching

- Bertil Folliot was elected to the CS of the UFR d'Ingénierie and to the conseil de l'Institut de Formation Doctorale.
- Following the reorganization of the Licence, Bertil Folliot has been responsable in the L2 Informatique for the "complémentaire métier" DANT (Développeur d'Applications Nouvelles Technologies, sélection d'un maximum de 25 étudiants) at Sorbonne University, since 2000.
- Coordinator and designer of the module Initiation aux Sytèmes d'Exploitation in L2 DANT, Sorbonne Université (25 étudiants), since 2019.
- Follower of the internships of the students of L2 DANT at Sorbonne University (from the beginning of the internship in April to the defense at the end of August), since 2019.
- Coordinator of the module "Projets Encadrés" of L2 DANT at Sorbonne University (typical projects: guitar hero, tetris, collaborative agenda, multi-player role game, etc.), since 2019.

10.2.2 Supervision

- PhD completed : Cédric Courtaud, CIFRE Thalès, 2016-2019, defended January 28 2020 [24], Gilles Muller, Julien Sopéna (Delys).
- PhD completed : Redha Gouicem, 2016-2020, defended October 23, 2020 [25], Gilles Muller, Julien Sopéna (Delys).
- PhD completed: Darius Mercadier, 2017-2020, defended November 20, 2020 [26], Pierre-Évariste Dagand, Gilles Muller.
- PhD completed : Lucas Serrano, 2017-2020, defended November 25, 2020 [27], Julia Lawall.
- PhD in progress : Yoann Ghigoff, 2019-2022, Gilles Muller, Julien Sopéna (Delys), Kahina Lazri (Orange Labs).
- PhD in progress : Josselin Giet, 2020-2023, Xavier Rival (Antique), Gilles Muller.
- PhD in progress : Pierre Nigron, 2019-2022, Pierre-Évariste Dagand, Julia Lawall.

10.2.3 Juries

- Julia Lawall: PhD jury of Van Duc Thong Hoang (Singapore Management University, reporter), July 29, 2020.
- Gilles Muller: PhD jury of Louison Gitzinger (U. Rennes), December 8, 2020.

10.3 Popularization

10.3.1 Internal or external Inria responsibilities

- Julia Lawall was a co-chair of the Commission des Emplois Scientifiques (CES).
- Julia Lawall was a member of the jury for CRCN positions at Inria-Paris.

10.3.2 Education

- Julia Lawall mentored Jaskaran Singh and Sumera Priyadarsini as part of the Linux Foundation's Community Bridge mentorship program. The internship of Jaskaran Singh was prolonged by 3 months with support from Collabora.

10.3.3 Interventions

- Julia Lawall gave a talk at PapersWeLove London, on "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel" (January 19, 2020)
- Julia Lawall gave an invited talk at QCon London 2020, on "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel" (March 2, 2020)
- Julia Lawall gave an invited talk at the company r2c on "Coccinelle, Prequel, and Spinfer: Automating Summarization and Application of Code Evolutions in the Linux Kernel" (May 27, 2020).
- Julia Lawall gave a talk at the 2020 Linux Plumbers Conference on "Understanding Linux Lists" (August 25, 2020).
- Julia Lawall was a session chair at the 2020 Linux Plumbers Conference.
- Julia Lawall was a member of a panel on "Would Abandoning the C Language Really Help?" at the 2020 Linux Security Summit (October 30, 2020)

11 Scientific production

11.1 Major publications

[1] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet and L. Rieg. 'A formally verified compiler for Lustre'. In: *PLDI*. 2017, pp. 586–601.

[2] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall and G. Muller. 'A foundation for flow-based program matching using temporal logic and model checking'. In: *POPL*. Savannah, GA, USA: ACM, Jan. 2009, pp. 114–126.

[3] L. Burgy, L. Réveillère, J. L. Lawall and G. Muller. 'Zebu: A Language-Based Approach for Network Protocol Message Processing'. In: *IEEE Trans. Software Eng.* 37.4 (2011), pp. 575–591.

[4] P.-É. Dagand, N. Tabareau and É. Tanter. 'Partial type equivalences for verified dependent interoperability'. In: *ICFP*. 2016, pp. 298–310.

[5] D. Mercadier and P.-É. Dagand. 'Usuba: high-throughput and constant-time ciphers, by construction'. In: *PLDI*. 2019, pp. 157–173.

[6] F. Mérillon, L. Réveillère, C. Consel, R. Marlet and G. Muller. 'Devil: An IDL for hardware programming'. In: *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, California: USENIX Association, Oct. 2000, pp. 17–30.

[7] G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Mérillon and L. Réveillère. 'Towards Robust OSes for Appliances: A New Approach Based on Domain-specific Languages'. In: *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*. Kolding, Denmark, 2000, pp. 19–24.

[8] G. Muller, J. L. Lawall and H. Duchesne. 'A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation'. In: *HASE - High Assurance Systems Engineering Conference*. Heidelberg, Germany: IEEE, Oct. 2005, pp. 56–65.

[9] Y. Padioleau, J. L. Lawall, R. R. Hansen and G. Muller. 'Documenting and Automating Collateral Evolutions in Linux Device Drivers'. In: *EuroSys*. Glasgow, Scotland, Mar. 2008, pp. 247–260.

[10] N. Palix, G. Thomas, S. Saha, C. Calvès, J. L. Lawall and G. Muller. 'Faults in Linux 2.6'. In: *ACM Transactions on Computer Systems* 32.2 (June 2014), 4:1–4:40.

11.2 Publications of the year

International journals

[11] J.-J. Bai, J. Lawall and S.-M. Hu. 'Effective Detection of Sleep-in-Atomic-Context Bugs in the Linux Kernel'. In: *ACM Transactions on Computer Systems* 36.4 (June 2020), p. 10. DOI: 10.1145/3381990. URL: https://hal.inria.fr/hal-03032244.

International peer-reviewed conferences

[12] S. Belaïd, P.-E. Dagand, D. Mercadier, M. Rivain and R. Wintersdorff. 'Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations'. In: *EUROCRYPT*. Eurocrypt 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques. Vol. 12107. Lecture Notes in Computer Science. Zagreb / Virtual, Croatia, 1st May 2020, pp. 311–341. DOI: 10.1007/978-3-030-45727-3_11. URL: https://hal.archives-ouvertes.fr/hal-02953167.

[13] G. Berthou, P.-E. Dagand, D. Demange, R. Oudin and T. Risset. 'Intermittent Computing with Peripherals, Formally Verified'. In: *LCTES '20: 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems Proceedings*. LCTES '20 - 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. London / Virtual, United Kingdom, June 2020, pp. 85–96. DOI: 10.1145/3372799.3394365. URL: https://hal.inria.fr/hal-02556878.

[14] A. Chueshev, J. Lawall, R. Bendraou and T. Ziadi. 'Expanding the Number of Reviewers in Open-Source Projects by Recommending Appropriate Developers'. In: ICSME 2020 - International Conference on Software Maintenance and Evolution. Adélaïde / Virtual, Australia, 27th Sept. 2020. URL: https://hal.inria.fr/hal-02928232.

[15] R. Gouicem, D. Carver, J.-P. Lozi, J. Sopena, B. Lepers, W. Zwaenepoel, N. Palix, J. Lawall and G. Muller. 'Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance'. In: 2020 USENIX Annual Technical Conference. Boston / Virtual, United States: https://www.usenix.org/conference/atc20/technical-sessions, 15th July 2020. URL: https://hal.inria.fr/hal-02901169.

[16] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo and L. Jiang. 'Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example'. In: ICPC 2020 - 28th IEEE/ACM International Conference on Program Comprehension - ERA track. Seoul / Virtual, South Korea: https://conf.researchr.org/home/icpc-2020, 13th July 2020. DOI: 10.1145/3387904.3389285. URL: https://hal.inria.fr/hal-02889835.

[17] T. Hoang, H. J. Kang, D. Lo and J. Lawall. 'CC2Vec: Distributed Representations of Code Changes'. In: ICSE 2020 - 42nd International Conference on Software Engineering. Seoul / Virtual, South Korea, 27th June 2020, pp. 518–529. DOI: 10.1145/3377811.3380361. URL: https://hal.inria.fr/hal-03030530.

[18] P. Kiaei, D. Mercadier, P.-E. Dagand, K. Heydemann and P. Schaumont. 'Custom Instruction Support for Modular Defense against Side-channel and Fault Attacks'. In: International Workshop on Constructive Side-Channel Analysis and Secure Design, COSADE 2020. Lecture Notes in Computer Science. Lugano, Switzerland, 5th Oct. 2020. URL: https://hal.archives-ouvertes.fr/hal-03058888.

[19] F. Laniel, D. Carver, J. Sopena, F. Wajsburt, J. Lejeune and M. Shapiro. 'MemOpLight: Leveraging application feedback to improve container memory consolidation'. In: NCA 2020 - 19th IEEE International Symposium on Network Computing and Applications. Cambridge / Virtual, United States, 24th Nov. 2020, pp. 1–10. DOI: 10.1109/NCA51143.2020.9306717. URL: https://hal.archives-ouvertes.fr/hal-03065629.

[20] B. Lepers, R. Gouicem, D. Carver, J.-P. Lozi, N. Palix, M.-V. Aponte, W. Zwaenepoel, J. Sopena, J. Lawall and G. Muller. 'Provable Multicore Schedulers with Ipanema: Application to Work Conservation'. In: Eurosys 2020 - European Conference on Computer Systems. Heraklion / Virtual, Greece, 27th Apr. 2020. DOI: 10.1145/3342195.3387544. URL: https://hal.inria.fr/hal-02554342.

[21] L. Serrano, V.-A. Nguyen, F. Thung, L. Jiang, D. Lo, J. Lawall and G. Muller. 'SPINFER: Inferring Semantic Patches for the Linux Kernel'. In: USENIX Annual Technical Conference. Boston / Virtual, United States, 15th July 2020. URL: https://hal.inria.fr/hal-02906912.

[22] F. Thung, S. A. Haryono, L. Serrano, G. Muller, J. Lawall, D. Lo and L. Jiang. 'Automated Deprecated-API Usage Update for Android Apps: How Far Are We?' In: SANER 2020 - 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER, RENE track). London, Ontario, Canada: https://saner2020.csd.uwo.ca/index, 18th Feb. 2020, pp. 602–611. DOI: 10.1109/SANER48275.2020.9054860. URL: https://hal.inria.fr/hal-02889832.

[23] N. Volanschi and J. Lawall. 'The Impact of Generic Data Structures: Decoding the Role of Lists in the Linux Kernel'. In: ASE 2020 - 35th IEEE/ACM International Conference on Automated Software Engineering. Melbourne / Virtual, Australia, 21st Sept. 2020. DOI: 10.1145/3324884.3416635. URL: https://hal.inria.fr/hal-02931554.

Doctoral dissertations and habilitation theses

[24] C. Courtaud. 'Characterization of the sensitivity to memory interferences in real-time systems embedded on multi-core platforms'. Sorbonne Universites, UPMC University of Paris 6, 28th Jan. 2020. URL: https://hal.archives-ouvertes.fr/tel-03022017.

[25] R. Gouicem. 'Thread Scheduling in Multi-core Operating Systems'. Sorbonne Université, 23rd Oct. 2020. URL: https://hal.archives-ouvertes.fr/tel-02977242.

[26] D. Mercadier. 'Usuba, Optimizing Bitslicing Compiler'. Sorbonne Université (France), 20th Nov. 2020. URL: https://tel.archives-ouvertes.fr/tel-03133456.

[27] L. Serrano. 'Automatic Inference of System Software Transformation Rules from Examples'. Sorbonne Université, 25th Nov. 2020. URL: https://hal.inria.fr/tel-03120648.

11.3 Cited publications

[28] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani and A. Ustuner. 'Thorough Static Analysis of Device Drivers'. In: *EuroSys*. 2006, pp. 73–85.

[29] A. Baumann, P. Barham, P.-É. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach and A. Singhania. 'The multikernel: A new OS architecture for scalable multicore systems'. In: *SOSP*. 2009, pp. 29–44.

[30] T. F. Bissyandé, L. Réveillère, J. L. Lawall, Y.-D. Bromberg and G. Muller. 'Implementing an Embedded Compiler using Program Transformation Rules'. In: *Software: Practice and Experience* 45.2 (Feb. 2015), pp. 177–196. URL: https://hal.archives-ouvertes.fr/hal-00844536.

[31] T. F. Bissyandé, L. Réveillère, J. L. Lawall, Y.-D. Bromberg and G. Muller. 'Implementing an embedded compiler using program transformation rules'. In: *Software: Practice and Experience* (2013).

[32] T. F. Bissyandé, L. Réveillère, J. L. Lawall and G. Muller. 'Ahead of Time Static Analysis for Automatic Generation of Debugging Interfaces to the Linux Kernel'. In: *Automated Software Engineering* (May 2014), pp. 1–39. DOI: 10.1007/s10515-014-0152-4. URL: https://hal.archives-ouvertes.fr/hal-00992283.

[33] A. P. Black, S. Ducasse, O. Nierstrasz and D. Pollet. *Pharo by Example*. Square Bracket Associates, 2010.

[34] E. Brady and K. Hammond. 'Resource-Safe Systems Programming with Embedded Domain Specific Languages'. In: *14th International Symposium on Practical Aspects of Declarative Languages (PADL)*. Vol. 7149. LNCS. Springer, 2012, pp. 242–257.

[35] T. Braibant and D. Pous. 'An Efficient Coq Tactic for Deciding Kleene Algebras'. In: *1st International Conference on Interactive Theorem Proving (ITP)*. Vol. 6172. LNCS. Springer, 2010, pp. 163–178.

[36] C. Cadar, D. Dunbar and D. R. Engler. 'KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs'. In: *OSDI*. 2008, pp. 209–224.

[37] V. Chipounov and G. Candea. 'Reverse Engineering of Binary Device Drivers with RevNIC'. In: *EuroSys*. 2010, pp. 167–180.

[38] A. Chlipala. 'The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier'. In: *ICFP*. 2013, pp. 391–402.

[39] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. 'Counterexample-guided abstraction refinement for symbolic model checking'. In: *J. ACM* 50.5 (2003), pp. 752–794.

[40] L. A. Clarke. 'A system to generate test data and symbolically execute programs'. In: *IEEE Transactions on Software Engineering* 2.3 (1976), pp. 215–222.

[41] P. Cousot and R. Cousot. 'Abstract Interpretation: Past, Present and Future'. In: *CSL-LICS*. 2014, 2:1–2:10.

[42] P.-É. Dagand, A. Baumann and T. Roscoe. 'Filet-o-Fish: practical and dependable domain-specific languages for OS development'. In: *Programming Languages and Operating Systems (PLOS)*. 2009, pp. 51–55.

[43] I. Dillig, T. Dillig and A. Aiken. 'Sound, complete and scalable path-sensitive analysis'. In: *PLDI*. June 2008, pp. 270–280.

[44] D. R. Engler, B. Chelf, A. Chou and S. Hallem. 'Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions'. In: *OSDI*. 2000, pp. 1–16.

[45] D. R. Engler, D. Y. Chen, A. Chou and B. Chelf. 'Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code'. In: *SOSP*. 2001, pp. 57–72.

[46] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[47] L. Gu, A. Vaynberg, B. Ford, Z. Shao and D. Costanzo. 'CertiKOS: A Certified Kernel for Secure Cloud Computing'. In: *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*. 2011, 3:1–3:5.

[48] L. Guo, J. L. Lawall and G. Muller. 'Oops! Where did that code snippet come from?' In: *11th Working Conference on Mining Software Repositories, MSR*. Hyderabad, India: ACM, May 2014, pp. 52–61.

[49] A. Israeli and D. G. Feitelson. 'The Linux kernel as a case study in software evolution'. In: *Journal of Systems and Software* 83.3 (2010), pp. 485–501.

[50] A. Kadav and M. M. Swift. 'Understanding modern device drivers'. In: *ASPLOS*. 2012, pp. 87–98.

[51] A. Kennedy, N. Benton, J. B. Jensen and P.-É. Dagand. 'Coq: The World's Best Macro Assembler?' In: *PPDP*. Madrid, Spain: ACM, 2013, pp. 13–24.

[52] G. A. Kildall. 'A Unified Approach to Global Program Optimization'. In: *POPL*. 1973, pp. 194–206.

[53] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood. 'seL4: formal verification of an OS kernel'. In: *SOSP*. 2009, pp. 207–220.

[54] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart and G. Muller. 'WYSIWIB: Exploiting fine-grained program structure in a scriptable API-usage protocol-finding process'. In: *Software, Practice Experience* 43.1 (2013), pp. 67–92.

[55] J. L. Lawall, B. Laurie, R. R. Hansen, N. Palix and G. Muller. 'Finding Error Handling Bugs in OpenSSL using Coccinelle'. In: *Proceeding of the 8th European Dependable Computing Conference (EDCC)*. Valencia, Spain, Apr. 2010, pp. 191–196.

[56] J. L. Lawall and D. Lo. 'An automated approach for finding variable-constant pairing bugs'. In: *25th IEEE/ACM International Conference on Automated Software Engineering*. Antwerp, Belgium, Sept. 2010, pp. 103–112.

[57] J. L. Lawall, D. Palinski, L. Gnirke and G. Muller. 'Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers'. In: *2017 USENIX Annual Technical Conference*. Santa Clara, CA, United States, July 2017, p. 12. URL: https://hal.inria.fr/hal-01556589.

[58] C. Le Goues and W. Weimer. 'Specification Mining with Few False Positives'. In: *TACAS*. Vol. 5505. Lecture Notes in Computer Science. York, UK, Mar. 2009, pp. 292–306.

[59] Z. Li, S. Lu, S. Myagmar and Y. Zhou. 'CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code'. In: *OSDI*. 2004, pp. 289–302.

[60] Z. Li and Y. Zhou. 'PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code'. In: *Proceedings of the 10th European Software Engineering Conference*. 2005, pp. 306–315.

[61] D. Lo and S.-C. Khoo. 'SMArTIC: towards building an accurate, robust and scalable specification miner'. In: *FSE*. 2006, pp. 265–275.

[62] J.-P. Lozi, F. David, G. Thomas, J. L. Lawall and G. Muller. 'Fast and Portable Locking for Multicore Architectures'. In: *ACM Transactions on Computer Systems* (Jan. 2016). DOI: 10.1145/2845079. URL: https://hal.inria.fr/hal-01252167.

[63] S. Lu, S. Park and Y. Zhou. 'Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing'. In: *IEEE Transactions on Software Engineering* 38.4 (2012), pp. 844–860.

[64] M. Mernik, J. Heering and A. M. Sloane. 'When and How to Develop Domain-specific Languages'. In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344. URL: http://dx.doi.org/10.1145/1118890.1118892.

[65] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan and E. Gan. 'RockSalt: better, faster, stronger SFI for the x86'. In: *PLDI*. 2012, pp. 395–404.

[66] M. Odersky and T. Rompf. 'Unifying functional and object-oriented programming with Scala'. In: *Commun. ACM* 57.4 (2014), pp. 76–86.

[67] M. C. Olesen, R. R. Hansen, J. L. Lawall and N. Palix. 'Coccinelle: Tool support for automated CERT C Secure Coding Standard certification'. In: *Science of Computer Programming*. Special Issue on Selected Contributions from the Open Source Software Certification (OpenCert) Workshops 91.B (Oct. 2014), pp. 141–160. URL: https://hal.inria.fr/hal-01096185.

[68] T. Reps, T. Ball, M. Das and J. Larus. 'The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem'. In: *ESEC/FSE*. 1997, pp. 432–449.

[69] L. R. Rodriguez and J. L. Lawall. 'Increasing Automation in the Backporting of Linux Drivers Using Coccinelle'. In: *11th European Dependable Computing Conference - Dependability in Practice*. 11th European Dependable Computing Conference - Dependability in Practice. Paris, France, Nov. 2015. URL: https://hal.inria.fr/hal-01213912.

[70] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. 'Error propagation analysis for file systems'. In: *PLDI*. Dublin, Ireland: ACM, June 2009, pp. 270–280.

[71] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur and G. Heiser. 'Automatic device driver synthesis with Termite'. In: *SOSP*. 2009, pp. 73–86.

[72] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm and M. Vij. 'User-Guided Device Driver Synthesis'. In: *OSDI*. 2014, pp. 661–676.

[73] R. Saha, J. L. Lawall, S. Khurshid and D. E. Perry. 'On the Effectiveness of Information Retrieval based Bug Localization for C Programs'. In: *International Conference on Software Maintenance and Evolution (ICSME)*. Victoria, BC, Canada, Sept. 2014.

[74] R. k. Saha, J. L. Lawall, S. Khurshid and D. E. Perry. 'On the Effectiveness of Information Retrieval Based Bug Localization for C Programs'. In: *ICSME 2014 - 30th International Conference on Software Maintenance and Evolution*. IEEE. Victoria, Canada, Sept. 2014, pp. 161–170. DOI: 10.1109/ICSME.2014.38. URL: https://hal.inria.fr/hal-01086082.

[75] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall and G. Muller. 'Hector: Detecting resource-release omission faults in error-handling code for systems software'. In: *DSN 2013 - 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE/IFIP. Budapest, Hungary: IEEE Computer Society, June 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575307. URL: https://hal.inria.fr/hal-00918079.

[76] D. A. Schmidt. 'Data Flow Analysis is Model Checking of Abstract Interpretations'. In: *POPL*. 1998, pp. 38–48.

[77] P. Senna Tschudin, J. L. Lawall and G. Muller. '3L: Learning Linux Logging'. In: *BElgian-NEtherlands software eVOLution seminar (BENEVOL 2015)*. Lille, France, Dec. 2015. URL: https://hal.inria.fr/hal-01239980.

[78] M. Shapiro. 'Purpose-built languages'. In: *Commun. ACM* 52.4 (2009), pp. 36–41.

[79] R. Tartler, D. Lohmann, J. Sincero and W. Schröder-Preikschat. 'Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem'. In: *EuroSys*. 2011, pp. 47–60.

[80] F. Thung, D. X. B. Le, D. Lo and J. L. Lawall. 'Recommending Code Changes for Automatic Backporting of Linux Device Drivers'. In: *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. Raleigh, North Carolina, United States, Oct. 2016. URL: https://hal.inria.fr/hal-01355859.

[81] W. Wang and M. Godfrey. 'A Study of Cloning in the Linux SCSI Drivers'. In: *Source Code Analysis and Manipulation (SCAM)*. IEEE, 2011.

[82] J. Yang and C. Hawblitzel. 'Safe to the Last Instruction: Automated Verification of a Type-safe Operating System'. In: *PLDI*. 2010, pp. 99–110.