RESEARCH CENTRE

Inria Paris Center at Sorbonne University

IN PARTNERSHIP WITH:

CNRS, Sorbonne Université

2022

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

Inria

Contents

Project-Team WHISPER 1

1 Team members, visitors, external collaborators 2

2 Overall objectives 2

3 Research program 3
3.1 Program analysis . . . 3
3.2 Domain Specific Languages . . . 4
3.3 Research direction: Tools for improving legacy infrastructure software . . . 4
3.4 Research direction: developing infrastructure software using Domain Specific Languages . . . 5

4 Application domains 5
4.1 Linux . . . 5
4.2 Device Drivers . . . 6

5 Social and environmental responsibility 6
5.1 Impact of research results . . . 6

6 Highlights of the year 6

7 New software and platforms 6
7.1 New software . . . 6
7.1.1 Coccinelle . . . 6
7.1.2 Prequel . . . 7

8 New results 7
8.1 OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores . . . 7
8.2 Towards User-Programmable Schedulers in the Operating System Kernel . . . 7
8.3 AndroEvolve: automated Android API update with data flow analysis and variable denormalization . . . 8

9 Bilateral contracts and grants with industry 8
9.1 Bilateral grants with industry . . . 8

10 Partnerships and cooperations 9
10.1 International initiatives . . . 9
10.1.1 Inria associate team not involved in an IIL or an international program . . . 9
10.2 International research visitors . . . 9
10.2.1 Visits of international scientists . . . 9
10.2.2 Visits to international teams . . . 10
10.3 National initiatives . . . 11
10.3.1 ANR . . . 11

11 Dissemination 11
11.1 Promoting scientific activities . . . 11
11.1.1 Scientific events: organisation . . . 11
11.1.2 Scientific events: selection . . . 11
11.1.3 Journal . . . 12
11.1.4 Invited talks . . . 12
11.1.5 Leadership within the scientific community . . . 12
11.1.6 Scientific expertise . . . 12
11.2 Teaching - Supervision - Juries . . . 12
11.2.1 Supervision . . . 12
11.2.2 Juries . . . 12
11.3 Popularization . . . 13

11.3.1 Interventions 13

12 Scientific production **13**
12.1 Major publications 13
12.2 Publications of the year 13
12.3 Cited publications 14

Project-Team WHISPER

Creation of the 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

A2.2.1. – Static analysis

A2.2.5. – Run-time systems

A2.2.8. – Code generation

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

A2.4.3. – Proofs

A2.5. – Software engineering

A2.5.4. – Software Maintenance & Evolution

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

B6.5. – Information systems

B6.6. – Embedded systems

1 Team members, visitors, external collaborators

Research Scientists

- Julia Lawall [Team leader, INRIA, Senior Researcher]
- Jean-Pierre Lozi [INRIA, Researcher, from Aug 2022]

PhD Students

- Papa Assane Fall [INRIA, from Oct 2022]
- Himadri Pandya [INRIA]

Technical Staff

- Thierry Martinez [INRIA, Engineer, 10%]

Administrative Assistants

- Christine Anocq [INRIA]
- Nelly Maloisel [INRIA]

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 [25, 30, 57]. 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 [1, 7, 8] for transforming C programs, or domain-specific languages such as Devil [4], Bossa [6] and Ipanema [3] 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 [20, 21, 22] or dynamic analysis [38, 40, 43]. 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* [21]. 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* [29] 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 [51]. More recently, more general forms of symbolic execution [20] 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 [17, 18].

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. [24] 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 [19], 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 [14]. 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) [35], 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 [33].

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 [5] [41].

Traditional approach. Using little languages to aid in software development is a tried-and-trusted technique [53] 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 [4]. 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 [6] 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.

Certifying DSLs. While 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 verification efforts onto these components.

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, git, 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 [15, 16, 42, 44, 49, 50, 52, 39, 34], 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, almost 9000 patches have been accepted into the Linux kernel based on the use of Coccinelle, including thousands by over 400 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 [55], 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 (completed in 2021) 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 [4] to interact with devices, Bossa [6] 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. 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. A key benefit would be to provide – by construction – a formal, mechanized semantics to the DSLs thus developed. Such a semantics would offer a foundation on which to base further verification efforts, while allowing interaction with non-verified code. We advocate a methodology based on incremental, piece-wise verification. While building fully-certified systems from the top-down is a worthwhile endeavor [30], 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 focuses on 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, v6.1, comprises over 23 million lines of code, and supports 23 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. [23], numerous research tools have been applied to the Linux kernel, typically for finding bugs [22, 37, 45, 54] or for computing software metrics [27, 56]. In our work, we have studied generic C bugs in Linux code [8], bugs in function protocol usage [31, 32], issues related to the processing of bug reports [48] and crash dumps [26], and the problem of backporting [44, 55], 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 twenty years have seen a number of approaches directed towards easing device driver development. Réveillère, who was supervised by G. Muller, proposes Devil [4], a domain-specific language for describing the low-level interface of a device. Chipounov *et al.* propose RevNic, [18] a template-based approach for porting device drivers from one OS to another. Ryzhyk *et al.* propose Termite, [46, 47] 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 [28] 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 [14], Coverity [23], CP-Miner, [36] PR-Miner [37], and Coccinelle [7]. 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

Environmental responsability The Whisper team is actively pursuing research on process scheduling for the Linux kernel. 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, thus reducing energy consumption. Work in this direction was published at USENIX ATC 2020 and EuroSys 2022. We are continuing to work in this area as part of our collaboration with Oracle (Section 9.1).

6 Highlights of the year

Jean-Pierre Lozi joined the team as CRCN, following a position at Oracle Labs.

7 New software and platforms

7.1 New software

7.1.1 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. Extensions to support C++ and Rust are in progress. A prototype has been developed for Java.

URL: https://coccinelle.gitlabpages.inria.fr/website/

Contact: Julia Lawall

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

Partner: IRILL

7.1.2 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

8 New results

Our work this year has mainly focused on task scheduling in the Linux kernel.

8.1 OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores

Participants: Julia Lawall *(Whisper)*, Himadri Chhaya-Shailesh *(Inria)*, Jean-Pierre Lozi *(Oracle Labs)*, Baptiste Lepers *(University of Sydney)*, Willy Zwaenepoel *(University of Sydney)*, Gilles Muller *(Whisper)*.

To best support highly parallel applications, Linux's CFS scheduler tends to spread tasks across the machine on task creation and wakeup. It has been observed, however, that in a server environment, such a strategy leads to tasks being unnecessarily placed on long-idle cores that are running at lower frequencies, reducing performance, and to tasks being unnecessarily distributed across sockets, consuming more energy. In this paper, we propose to exploit the principle of core reuse, by constructing a nest of cores to be used in priority for task scheduling, thus obtaining higher frequencies and using fewer sockets. We implement the Nest scheduler in the Linux kernel. While performance and energy usage are comparable to CFS for highly parallel applications, for a range of applications using fewer tasks than cores, Nest improves performance 10%-2× and can reduce energy usage.

This work was presented at EuroSys 2022 [11]. The source code and scripts used to produce the results are available in an online artifact.

8.2 Towards User-Programmable Schedulers in the Operating System Kernel

Participants: Djob Mvondo *(Inria-Rennes, WIDE)*, Antonio Barbalace *(University of Edinburgh)*, Jean-Pierre Lozi *(Oracle Labs, now Whisper)*, Gilles Muller *(Whisper)*.

This position paper argues that it is time for OS kernel-level schedulers to be user-programmable, from at least a category of users, without any security related side-effects. We introduce our preliminary design that borrows the microkernels' design principle of dividing mechanisms from policies, and applies that to monolithic OSes. All scheduling related mechanisms are always built-in in the OS kernel, while scheduling policies are modifiable, or definable, at runtime by users' applications (with specific privileges).

This work was presented at SPMA 22 - 11th workshop on Systems for Post-Moore Architectures [13], held with EuroSys 2022.

8.3 AndroEvolve: automated Android API update with data flow analysis and variable denormalization

The Android operating system is frequently updated, with each version bringing a new set of APIs. New versions may involve API deprecation; Android apps using deprecated APIs need to be updated to ensure the apps' compatibility with old and new Android versions. Updating deprecated APIs is a time-consuming endeavor. Hence, automating the updates of Android APIs can be beneficial for developers. CocciEvolve is the state-of-the-art approach for this automation. However, it has several limitations, including its inability to resolve out-of-method variables and the low code readability of its updates due to the addition of temporary variables.

In an attempt to further improve the performance of automated Android API update, we propose an approach named AndroEvolve, that addresses the limitations of CocciEvolve through the addition of data flow analysis and variable name denormalization. Data flow analysis enables AndroEvolve to resolve the value of any variable within the file scope. Variable name denormalization replaces temporary variables that may present in the CocciEvolve update with appropriate values in the target file. We have evaluated the performance of AndroEvolve and the readability of its updates on 372 target files containing 565 deprecated API usages. Each target file represents a file from an Android application that uses a deprecated API in its code. AndroEvolve successfully updates 481 out of 565 deprecated API invocations correctly, achieving an accuracy of 85.1%. Compared to CocciEvolve, AndroEvolve produces 32.9% more instances of correct updates. Moreover, our manual and automated evaluation shows that AndroEvolve updates are more readable than CocciEvolve updates.

This work was published in the journal Empirical Software Engineering [10].

9 Bilateral contracts and grants with industry

9.1 Bilateral grants with industry

Oracle, 2022-2023, 100 000 dollar gift.

Participants: Julia Lawall, Jean-Pierre Lozi *(Oracle)*.

This donation is the third and final in a series of donations from Oracle to support research on kernel-level task schedulers in the Whisper team. This year, we will focus on designing an interface for writing custom schedulers outside of the kernel. This interface will rely on BPF, a kernel component that makes it possible to safely inject code into the kernel. BPF was initially designed for packet filtering, but it now has a lot more applications, such as tracing, and more recently, in-kernel caching. Significant research and engineering effort will be required to extend BPF with all the helpers required to write expressive scheduling policies. Custom schedulers will be written using the new BPF interface as a proof of concept, and to measure potential overheads.

10 Partnerships and cooperations

10.1 International initiatives

10.1.1 Inria associate team not involved in an IIL or an international program

CSG

Title: Proving Concurrent Multi-Core Operating Systems

Duration: 2019 -> 2022

Coordinator: Willy Zwaenepoel (willy.zwaenepoel@sydney.edu.au)

Partners:

- University of Sydney (Australie)

Inria contact: Julia Lawall

Summary: The initial topic of this cooperation was the development of proved multicore schedulers. Over the first three years, we 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 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 future, we plan to 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.

Baptiste Lepers of the University of Sydney spent one week with Whisper in March 2022, working on detecting errors in the use of memory barriers in the Linux kernel. This paper has been accepted at EuroSys 2023.

10.2 International research visitors

10.2.1 Visits of international scientists

International visits to the team

Baptiste Lepers

Status Researcher

Institution of origin: University of Sydney

Country: Australia

Dates: 1 week, March 2022

Context of the visit: Detection of bug in Linux kernel memory barrier usage

Mobility program/type of mobility: CSG associate team

Keisuke Nishimura

Status Masters student

Institution of origin: University of Tokyo

Country: Japan

Dates: May - August 2022 (3 months)

Context of the visit: Inference of bugfix backports with explanations

Mobility program/type of mobility: informal

Tathagata Roy

Status Bachelors student

Institution of origin: Indian Institute of Information Technology, Kalyani.

Country: India

Dates: Dec 7 2022 - January 6 2023

Context of the visit: Coccinelle for Rust

Mobility program/type of mobility: informal

10.2.2 Visits to international teams

Research stays abroad

Julia Lawall

Visited institution: Leibniz Supercomputing Center

Country: Germany

Dates: 1 week, July 2022

Context of the visit: Collaborate on Coccinelle for C++.

Mobility program/type of mobility: informal

Julia Lawall

Visited institution: Singapore Management University

Country: Singapore

Dates: 1 week, November 2022

Context of the visit: Explore future collaboration opportunities

Mobility program/type of mobility: informal

10.3 National initiatives

10.3.1 ANR

VeriAmos

Participants: Xavier Rival *(Antique (PI))*, Nicolas Palix *(UGA (Erods))*, Gilles Muller, Julia Lawall, Rehan Malak.

- Awarded in 2018, duration 2018 - 2022
- Members: Inria (Antique, Whisper), UGA (Erods)
- 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 is attacking this problem by building on recent advances in the design of domain-specific languages and static analyzers for systems code. We are investigating 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 targets I/O scheduling components.

11 Dissemination

11.1 Promoting scientific activities

11.1.1 Scientific events: organisation

Member of the organizing committees

- Julia Lawall: ESEC-FSE 2022: workshop chair, with Jun Sun
- Julia Lawall: EuroSys 2022: Publications chair

11.1.2 Scientific events: selection

Member of the conference program committees

- Julia Lawall: GPCE 2022
- Julia Lawall: BENEVOL 2022
- Julia Lawall: Scheme workshop 2022
- Julia Lawall: EuroSys doctoral workshop 2022
- Julia Lawall: ICSME 2022
- Julia Lawall: ICPC 2022
- Julia Lawall: ASE 2022

- Julia Lawall: PEPM 2022
- Julia Lawall: DSN 2022
- Julia Lawall: EuroSys 2022
- Julia Lawall: ICSE NIER 2022

11.1.3 Journal

Member of the editorial boards

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

Reviewer - reviewing activities

- Julia Lawall: ACM TOSEM, IEEE TSE, Science of Computer Programming, Empirical Software Engineering.

11.1.4 Invited talks

- Julia Lawall, NASA Formal Methods 2022, The Coccinelle C-program matching and transformation tool. May 26, 2022. An accompanying paper was published in the conference proceedings [12].
- Julia Lawall was invited to the the Dagstuhl Seminar 22341 Power and Energy-aware Computing on Heterogeneous Systems (PEACHES) where she presented "OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores" [11].

11.1.5 Leadership within the scientific community

- Julia Lawall: Secretary of IFIP TC2.
- Julia Lawall: Member of the steering committee of IFIP France.
- Julia Lawall: President of the committee for the systems researcher prizes of the ASF, with Xavier Lagrange.
- Julia Lawall: Member of the advisory board of Software Heritage.

11.1.6 Scientific expertise

- Julia Lawall: Evaluator for an Associate Professor position at the University of Copenhagen.
- Julia Lawall: Evaluator of a proposal for the ANR (Astrid program).
- Julia Lawall: President of the jury for CRCN in Rennes.

11.2 Teaching - Supervision - Juries

11.2.1 Supervision

- Julia Lawall and Jean-Pierre Lozi supervise the PhD of Himadri Chhaya-Shailesh, and contribute to the supervision of Papa Assane Fall (with Alain Tchana in Grenoble) and Cesaire Honore (with David Bromberg and Djob Mvondo in Rennes).

11.2.2 Juries

- Julia Lawall was a member of the jury for the PhD defense of Olivier Nicole ENS/CEA.
- Julia Lawall was a member of the jury for the PhD defense of Necip Fazil Yildiran at the University of Central Florida.

11.3 Popularization

11.3.1 Interventions

- Julia Lawall presented the Nest scheduler [11] at the 2022 Linux Plumbers conference.

12 Scientific production

12.1 Major publications

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

12.2 Publications of the year

International journals

[10] S. Haryono, F. Thung, D. Lo, L. Jiang, J. Lawall, H. J. Kang, L. Serrano and G. Muller. 'AndroEvolve: automated Android API update with data flow analysis and variable denormalization'. In: *Empirical Software Engineering* 27.3 (May 2022), p. 73. DOI: 10.1007/s10664-021-10096-0. URL: https://hal.inria.fr/hal-03921758.

International peer-reviewed conferences

[11] J. Lawall, H. Chhaya-Shailesh, J.-P. Lozi, B. Lepers, W. Zwaenepoel and G. Muller. 'OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores'. In: EuroSys 2022 - Seventeenth European Conference on Computer Systems. Rennes, France, 5th Apr. 2022. DOI: 10.1145/3492321.3519585. URL: https://hal.inria.fr/hal-03612592.

[12] J. Lawall and G. Muller. 'Automating Program Transformation with Coccinelle'. In: 2022 NASA Formal Methods - 14th International Symposium. Pasadena, CA, USA, United States, 24th May 2022. URL: https://hal.inria.fr/hal-03791022.

Conferences without proceedings

[13] D. Mvondo, A. Barbalace, J.-P. Lozi and G. Muller. 'Towards User-Programmable Schedulers in the Operating System Kernel'. In: SPMA 22 - 11th workshop on Systems for Post-Moore Architectures. Rennes, France, 5th Apr. 2022, pp. 1–4. URL: https://hal.inria.fr/hal-03750209.

12.3 Cited publications

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

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

[54] 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.

[55] 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.

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

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