RESEARCH CENTRE

Inria Paris Centre

2024 ACTIVITY REPORT

Project-Team WHISPER

Well Honed Infrastructure Software for Programming Environments and Runtimes

DOMAIN

Networks, Systems and Services, Distributed Computing

THEME

Distributed Systems and middleware

Inria

Contents

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]

Post-Doctoral Fellows

- Tomas Faltin [INRIA, Post-Doctoral Fellow, from May 2024]
- Xutong Ma [INRIA, Post-Doctoral Fellow, from Sep 2024]

PhD Students

- Maxime Derri [Orange Labs, CIFRE, from Nov 2024]
- Papa Assane Fall [INRIA]
- Victor Laforet [INRIA]
- Keisuke Nishimura [INRIA, from Dec 2024]
- Himadri Pandya [INRIA]

Technical Staff

- Tomas Faltin [INRIA, Engineer, until Apr 2024]
- Victor Gambier [INRIA, Engineer, from Sep 2024]
- Keisuke Nishimura [INRIA, until Nov 2024]

Interns and Apprentices

- Clement Gachod [INRIA, Intern, until Jul 2024]
- Corinn Tiffany [UNIV BROWN, from Mar 2024 until May 2024]

Administrative Assistants

- Martial Le Henaff [INRIA, from May 2024]
- Nelly Maloisel [INRIA]

Visiting Scientist

- Tathagata Roy [IIT Delhi, until Mar 2024]

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 of infrastructure software include operating systems, virtual machine hypervisors, managed runtime environments, and standard libraries. 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. Whisper addresses these dual problems of infrastructure software performance and correctness, both in a given instance of the software and as the software evolves.

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 will put an emphasis on achieving measurable improvements in performance and correctness in practice, and on feeding these improvements back to the infrastructure-software developer community.

We focus on two main axes. The first axis, system performance, targets the performance of applications, in both bare-metal and virtual environments, based on the impact of operating-system services. This direction not only involves proposing new algorithms, but also designing tools and methodologies to help developers understand the system behavior. When addressing these areas, we will explore the use of DSLs [2, 5, 7, 8, 9], when appropriate, to enhance the usability, configurability, reliability, and robustness of the proposed approaches. The second axis, system re-engineering, targets the quality of legacy infrastructure software. Infrastructure software tends to evolve faster than many of its users are able to keep up with, leading to massive ecosystem fragmentation. Building on tools such as Coccinelle for automating widespread code changes [1, 3, 12], we will investigate how to facilitate and improve the reliability of the maintenance of such variants, for example in terms of identifying, adapting, and applying relevant bug fixes.

3 Research program

The research program of the Whisper team is designed around two main axes: system performance and system re-engineering.

3.1 System performance

Our work on system performance focuses on scheduling, virtualization, synchronization, and the interactions between them. A task scheduler decides which runnable task should be able to execute on a CPU, and, in a multicore setting, on which core that execution should take place. As the task scheduler determines how much CPU time the tasks of an application receive, as well as whether the chosen CPU has effcient access to needed hardware resources (network, accelerators, caches, etc.), it has a large impact on application performance. In a general-purpose programming setting, the task scheduler is typically located in the operating system kernel, and is expected to serve the needs of all applications. It has no knowledge of future application behavior. Such a setting raises many challenges. We are currently focusing on four main issues: scheduling and virtualization, making locks aware of scheduler preemptions, control over scheduling from the user level, and task scheduling for heterogeneous architectures.

Virtualization decomposes the system support into one or more standard operating systems, known as guest OSes, that in turn run on a minimalistic operating system, known as a hypervisor, which provides isolation between the guest OSes and provides access to the hardware. A guest OS schedules application tasks based on its view of the set of available CPUs, but these CPUs are actually just tasks that are in turn scheduled by the hypervisor on the physical CPUs of the machine. This dual level of scheduling

introduces the risk of contradictory scheduling strategies. We are investigating the impact of this dual level of scheduling on the performance of applications in a highly multicore setting.

Effcient lock algorithms are key to ensuring application scalability on multi-core architectures [20, 23], [6]. Two families of lock algorithms exist: (1) with blocking locks, tasks sleep in a wait list until the lock is free, whereas (2) with spin locks, tasks busy-wait on the value of an atomic boolean variable that records whether or not the lock is free. While blocking locks consume less CPU, they are also less reactive, as acquiring the lock requires waking up the acquiring task, which entails a context switch. Spin locks, on the other hand, are very reactive, as the lock is acquired by a busy-waiting task as soon as the lock is released. But spinlocks may perform very poorly in overloaded scenarios, because spinning tasks can preempt the lock-holding task, preventing any progress on the critical path. We would like to turn spin locks into blocking locks in an overloaded scenario, relying on BPF to safely inject code into the kernel to detect when the Linux kernel scheduler preempts the lock holder.

Recently, several frameworks has been developed (notably, ghOSt from Google [25] and sched-ext from Meta [18]) that allow the user level to inject new schedulers into the Linux kernel. We would like to investigate how such a framework can be used to improve performance, using scheduling-decision-making processes that are not feasible in the kernel. For example, machine learning could be beneficial for inferring task properties and guiding scheduling accordingly, but is not practical in the kernel, due to the latency involved, memory constraints, and the lack of kernel support for floating point numbers. Our goal would be to provide a proof-of-concept, as it is not our focus to develop new machine-learning algorithms. We will also consider whether it can be useful to export other kinds of operating system services to user level, such as memory management. Finally, we will consider whether it is possible to design a generic kernel interface for exporting to the user level performance-critical operating-system services. This work is in collaboration with Alain Tchana and Renaud Lachaize of the Erods team at LIG, as part of the Inria Défi OS

The heuristics for scheduling employed by the Linux kernel have evolved over time, primarily in a homogeneous setting, where all cores have similar properties. Today, heterogeneous computing is becoming more prevalent, combining powerful processors for compute-intensive tasks with slower, more energy-efficient, processors for lighter weight tasks. Such an architecture is found in the big.LITTLE processors from Arm, typically used in mobile environments, and is now moving to desktop computing, with the Alder Lake architecture from Intel. We will assess the degree to which the Linux kernel task scheduler is able to effectively support such architectures and propose improvements, in terms of both meeting application performance requirements and saving energy. This work is in collaboration with David Bromberg and Djob Mvondo of the WIDE team at Inria-Rennes, as part of the Inria Défi OS.

3.2 System re-engineering

In the area of system re-engineering, we are interested in improving the quality of infrastructure software as well as in automating the transformations required to make it possible to use such software in new settings. Our work builds on our previous work on Coccinelle [3], a program transformation system for C code that has been extensively used on the Linux kernel and other C infrastructure software.

We are extending Coccinelle to C++ and to Rust. The extension to C++ reuses the existing Coccinelle implementation, simply adding the C++-specific constructs. The goal of this work, in collaboration with Michele Martone of the Leibniz Supercomputing Center, is to consider how Coccinelle can be used to adapt existing HPC software for use on GPUs, by introducing the use of calls to relevant libraries. The extension to Rust, on the other hand, is a complete re-implementation of Coccinelle, to benefit from tools being developed by the Rust community for parsing and pretty printing (Rust Analyzer and rustfmt) Rust code. The reuse of these tools has made it possible to quickly develop a preliminary implementation for the full Rust language. In the future, we may consider how to generalize this process to other languages, addressing the challenge that generic parsing tools may not present information in a way that is suitable for program transformation, for example in the choice of non-terminals.

We are considering how Coccinelle can be used to facilitate long-term software maintenance. Many software projects are obliged to maintain multiple variants, *e.g.*, to meet the demands of users who rely on the properties of old variants. An essential part of software maintenance is thus to apply new bug fixes to these old variants. Intervening changes in the software, however, may imply that fixes can not be applied to older versions directly. We are looking into how Coccinelle can be used to collect and exploit

information about recent code changes, to adapt new bug fixes to older source code.

Today, there are numerous tools for scanning infrastructure software for common types of vulnerabilities, such as NULL pointer dereferences and buffer overflows. Avoiding such issues, however, is not sufficient to ensure that the code behaves correctly; the code must also implement the intended algorithm. Checking algorithmic properties, however, is much more challenging, because each service implements its own algorithm. Formal verification tools such as Frama-C [19] can make it possible to express and verify such properties, but remain challenging to use, particularly for developers without experience in formal verification. The challenge is compounded by the fact that the code may evolve frequently, making any proofs quickly out of date. We are investigating how to design tools that can facilitate proofs about Linux kernel code, both in how to extract relevant parts of the code base, to reduce the amount of code that has to be considered by the proving process, and in how proofs can be evolved in response to changes in the code base, analogous to the adaptation of bug fixes over multiple versions.

4 Application domains

4.1 Linux

Linux is an open-source operating system that is used in settings ranging from embedded systems to supercomputers. Linux kernel v6.1, released in December 2022, 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. [22], numerous research tools have been applied to the Linux kernel, typically for finding bugs [21, 28, 31, 35] or for computing software metrics [26, 41]. In our work, we have studied generic C bugs in Linux code [11], bugs in function protocol usage [29, 30], issues related to the processing of bug reports [39] and crash dumps [24], and the problem of backporting [36, 40], 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 *et al.* propose Devil [34], a domain-specific language for describing the low-level interface of a device. Chipounov *et al.* propose RevNic, [17] a template-based approach for porting device drivers from one OS to another. Ryzhyk *et al.* propose Termite, [37, 38] 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 [27] 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 [16], Coverity [22], CP-Miner, [32] PR-Miner [33], and Coccinelle [10]. 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.

4.3 Multicore computing

Today, multicore computing is fundamental across a wide variety of industries. Applications such as machines learning, scientific computing, image processing, etc., run large numbers of threads on highly multicore processors. Such applications must furthermore contend with a variety of hardware architectures, with different CPU and memory topologies, CPU capacities, access to external resources such as storage, networking and accellerators, etc. Virtualization facilitates hardware sharing, but complicates access to and reasoning about hardware resources. Our work on system performance targets helping multicore applications run more efficiently, and helping developers understand their application performance, to enable them to find appropriate solutions.

5 Social and environmental responsibility

5.1 Footprint of research activities

Most of our resource-intensive research activities are carried out on Grid 5000, which is a shared national infrastructure, saving the cost of building, transporting, and continuously running a variety of dedicated machines.

6 New software, platforms, open data

6.1 Open data

"Should we balance" artifact

Contributors: Julia Lawall *(Whisper)*, Keisuke Nishimura *(Whisper)*, Jean-Pierre Lozi *(Whisper)*

Description: Artifact for the SAS 2024 NEAT paper "Should We Balance? Towards Formal Verification of the Linux Kernel Scheduler"

Dataset PID (DOI,...): https://zenodo.org/records/13132904

Project link:

Publications: [15]

Contact: Keisuke.Nishimura@inria.fr

Release contributions:

7 New results

The publications in 2024 are related to improving the robusteness of the Linux kernel.

7.1 Should We Balance? Towards Formal Verification of the Linux Kernel Scheduler

Participants: Julia Lawall *(Whisper)*, Keisuke Nishimura *(Whisper)*, Jean-Pierre Lozi *(Whisper)*.

The frequent tweaking of heuristics in the Linux kernel task scheduler suggests a need for formal verification, to ensure that important properties are maintained. Nevertheless, writing and verifying specifications for Linux kernel code have been considered to be impractical, leading other operating system verification efforts to propose co-design of new code and associated specifications instead. Furthermore, the Linux kernel evolves frequently, making any verification effort quickly out of date. Still, verification tools for C code are becoming more and more robust. In this paper, published at the 2024 Static Analysis Symposium (NEAT paper) [15], we explore whether it is now possible to apply formal verification directly to Linux kernel code, and to maintain the resulting specifications as the Linux kernel evolves. Our experiment focuses on the function `should_we_balance`, which is the gatekeeper to the Linux kernel scheduler's load balancer, and on the verification tool Frama-C.

This work was funded in part by the ANR grants VeriAMOS and EMASS.

7.2 Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times

Participants: Necip Fazil Yildiran *(University of Central Florida)*, Jeho Oh *(University of Texas)*, Julia Lawall *(Whisper)*, Paul Gazzillo *(University of Central Florida)*.

The Linux kernel is highly-configurable, with a build system that takes a configuration file as input and automatically tailors the source code accordingly. Configurability, however, complicates testing, because different configuration options lead to the inclusion of different code fragments. With thousands of patches received per month, Linux kernel maintainers employ extensive automated continuous integration testing. To attempt patch coverage, i.e., taking all changed lines into account, current approaches either use configuration files that maximize total statement coverage or use multiple randomly-generated configuration files, both of which incur high build times without guaranteeing patch coverage. To achieve patch coverage without exploding build times, we propose krepair, which automatically repairs configuration files that are fast-building but have poor patch coverage to achieve high patch coverage with little effect on build times. krepair works by discovering a small set of changes to a configuration file that will ensure patch coverage, preserving most of the original configuration file's settings. Our evaluation shows that, when applied to configuration files with poor patch coverage on a statistically-significant sample of recent Linux kernel patches, krepair achieves nearly complete patch coverage, 98.5% on average, while changing less than 1.53% of the original default configuration file in 99% of patches, which keeps build times 10.5x faster than maximal configuration files.

This work was published at the 2024 ACM conference on Foundations of Software Engineering (FSE) [14].

7.3 Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel

Participants: Yunbo Lyu *(Singapore Management University)*, Hong Jin Kang *(UCLA)*, Ratnadira Widyasari *(Singapore Management University)*, Julia Lawall *(Whisper)*, David Lo *(Singapore Management University)*.

The SZZ algorithm is used to connect bug-fixing commits to the earlier commits that introduced bugs. This algorithm has many applications and many variants have been devised. However, there are some types of commits that cannot be traced by the SZZ algorithm, referred to as "ghost commits". The evaluation of how these ghost commits impact the SZZ implementations remains limited. Moreover,

these implementations have been evaluated on datasets created by software engineering researchers from information in bug trackers and version-control histories.

Since Oct 2013, the Linux kernel developers have started labelling bug-fixing patches with the commit identifiers of the corresponding bug-inducing commit(s) as a standard practice. As of v6.1-rc5, 76,046 pairs of bug-fixing patches and bug-inducing commits are available. This provides a unique opportunity to evaluate the SZZ algorithm on a large dataset that has been created and reviewed by project developers, entirely independently of the biases of software engineering researchers.

In this paper, published in IEEE Transactions on Software Engineering [13], we apply six SZZ implementations to 76,046 pairs of bug-fixing patches and bug-introducing commits from the Linux kernel. Our findings reveal that SZZ algorithms experience a more significant decline in recall on our dataset (↓ 13.8%) as compared to prior findings reported by Rosa et al., and the disparities between the individual SZZ algorithms diminish. Moreover, we find that 17.47% of bug-fixing commits are ghost commits. Finally, we propose Tracing-Commit SZZ (TC-SZZ), that traces all commits in the change history of lines modified or deleted in bug-fixing commits. Applying TC-SZZ to all failure cases, excluding ghost commits, we found that TC-SZZ could identify 17.7% of them. Our further analysis based on git log found that 34.6% of bug-inducing commits were in the function history, 27.5% in the file history (but not in the function history), and 37.9% not in the file history. We further evaluated the effectiveness of ChatGPT in boosting the SZZ algorithm's ability to identify bug-inducing commits in the function history, in the file history and not in the file history.

8 Bilateral contracts and grants with industry

Extension of the Linux kernel with safe user programs (CIFRE)

Participants: Julia Lawall *(Whisper)*, Kahina Lazri *(Orange Labs)*, Maxime Derri *(Orange Labs)*.

The operating system kernel represents a privileged point of observation, which enables collecting data both at the infrastructure level and at the application level. The kernel also provides an observation point for both network and system activities. The language eBPF (extended Berkeley Packet Filter) has revolutionized observation practices by enabling observation programs provided by user-space to be executed at the Linux kernel level. The eBPF technology comprises multiple components, including helpers, maps and verification. Together, these components allow eBPF programs to benefit from a wide access to kernel functions, while at the same time guaranteeing safety.

Several network and system management services have taken advantage of the programmability offered by eBPF to enhance infrastructures with supervision, tracing and security services that run natively at the kernel level. This ensures high performance and a high degree of visibility of application activity. The criticality of the kernel nevertheless restricts the instructions that an eBPF program can execute. The component responsible for verifying program safety is the eBPF verifier. The verifier's objective is to ensure that the programs to be executed by the kernel are safe. This, however, has the effect of significantly reducing the expressiveness offered by eBPF by restricting certain programming mechanisms, such as variable-size loops, or by limiting the number of instructions a given program can execute. These constraints complicate the development of eBPF security applications, such as http traffic processing or managing TLS connections.

The aim of this project is to reconsider the design of program verification as implemented by the eBPF verifier in order to improve the robustness of verification on the one hand, and to increase the expressiveness of programs on the other. The results of this project are expected to compare the verification choices adopted by the Linux community with the state of the art of formal verification on the one hand, and with that of so-called safe programming languages such as Rust, in order to propose an improvement in the usability of the eBPF infrastructure. A subsidiary result of the project would be to reduce the need for helpers (Linux kernel functions).

9 Partnerships and cooperations

9.1 International initiatives

9.1.1 Participation in other International Programs

Automating the Systematic Evolution of HPC Codes

Participants: Gerald Mathias *(LRZ (PI))*, Michele Martone *(LRZ)*, Julia Lawall *(Whisper)*.

Program: BayFrance

Title: Automating the Systematic Evolution of HPC Codes

Partner Institution(s): Leibniz Supercomputing Centre (LRZ), Germany and Inria, Whisper team.

Date/Duration: 2023-2024 (1 year)

Additional info/keywords: The project contributed to extending Coccinelle to support C++ code and the application of Coccinelle to HPC code.

9.2 International research visitors

9.2.1 Visits of international scientists

Other international visits to the team

Michele Martone

Status researcher

Institution of origin: Leibniz Supercomputing Centre

Country: Germany

Dates: July 8-12, 2024

Context of the visit: BayFrance collaboration project

Mobility program/type of mobility: Research stay

9.2.2 Visits to international teams

Research stays abroad

Julia Lawall

Visited institution: Leibniz Supercomputing Centre

Country: Germany

Dates: August 18-24, 2024

Context of the visit: BayFrance collaboration project

Mobility program/type of mobility: Research stay

9.3 National initiatives

9.3.1 ANR

VeriAmos

Participants: Xavier Rival *(Antique (PI))*, Nicolas Palix *(UGA (Erods))*, Gilles Muller *(Whisper)*, Julia Lawall *(Whisper)*, Keisuke Nishimura *(Whisper)*.

- Awarded in 2018, duration 2018 - 2022 (extended through June 2024)
- 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.

EMASS

Participants: Matthieu Lemerre *(CEA, EMASS PI)*, Sébastien Bardin *(CEA)*, Frédéric Recoules *(CEA)*, Xavier Rival *(Antique (Inria PI))*, Julia Lawall *(Whisper)*, Keisuke Nishimura *(Whisper)*, Kosmatov Nikolai *(Thales RT)*, Delphine Longuet *(Thales RT)*, Romain Soulat *(Thales RT)*.

- Awarded in 2023, duration 42 months (2023 - 2026).
- Members: Inria (Antique, Whisper), CEA, Thales RT
- Funding: ANR, 162,720 euros awarded to Inria.
- Objectives: Current systems programs, like OS kernels, hypervisors, or core libraries found in the bottom layer of every application stack, are today mostly written in systems programming languages, such as C, C++, or assembly, that give programmers low-level control over resource management at the expense of safety. This leads to frequent memory corruption errors in systems programs, which is one of the major cybersecurity issues today. Backed by the consortium's strong experience in advanced memory analyses (shape analyses), source and binary-level program analysis for security, software engineering by scalable static analysis on industrial case studies, and OS kernel verification, we want to provide an effective solution to this problem, even for the most challenging systems software, i.e., OS kernels and hypervisors.

The EMASS project targets a solution to this problem relying on a static analysis of memory, using types as a base abstraction. Our goal is to be able to automatically verify the most challenging systems code, by developing a scalable compositional analysis, and by tackling other important security properties such as non-interference between the tasks of the OS, or functional contracts on the OS system calls, with the guidance of a low amount of intuitive type annotations. The project will result in strong scientific results in type-based static analysis, and in the EMASS toolkit, a pragmatic open-source solution for verifying and certifying secure systems software without formal methods expertise, whose practical use will be validated in challenging industrial case studies.

The contribution of the Whisper team will focus on using our tools such as Coccinelle [3] and Prequel [4] to find challenging case studies, and on developing strategies for continuous verification of systems software.

DiVa: DIsaggregated VirtuAlization

Participants: Gaël Thomas *(Inria Benagil (DiVa PI))*, Mathieu Bacou *(Inria Benagil)*, Vivien Quema *(Grenoble INP)*, Jean-Pierre Lozi *(Whisper)*, Julia Lawall *(Whisper)*, Alain Tchana *(Grenoble INP)*, Daniel Hagimont *(INP Toulouse)*, Boris Teabe *(INP Toulouse)*, Julien Sopena *(Sorbonne University)*.

- Awarded in 2023, duration 7 years (2023-2030)
- Members: Inria (Benagil, Whisper), Grenoble INP, INP Toulouse, Sorbonne University
- Funding: ANR-PEPR, 321,839 euros.
- Objectives: Cloud infrastructures are currently undergoing major changes with the advent of disaggregated and edge clouds. The objective of the DiVA (DIsaggregated VirtuAlization) project is to prepare the system stack for these next generations of cloud infrastructures. In order to use these infrastructures more efficiently and to avoid wasting hardware resources, we propose to revisit the system mechanisms at the basis of any cloud infrastructure.

  In the context of disaggregated infrastructures, it becomes possible, at the scale of a few clusters of machines, to fully mutualize hardware resources. In detail, using the recent CXL protocol, any processor on any machine in the cluster can transparently access any hardware resource (memory, network, disk, accelerator) on any other server. This evolution calls for new hypervisors and operating system designs because virtual machines will become elastic (i.e., it will be possible to add new resources to a virtual machine at runtime) and distributed (i.e., the hardware resources used by a virtual machine are physically distributed). In the DiVA project, we will therefore (i) define new virtualization interfaces in order to allow a virtual machine to dynamically allocate or release hardware resources, (ii) study new scheduling and placement mechanisms in the guest operating system in order to efficiently use these distributed resources, (iii) study how we could use programmable networks to optimize the performance of virtual machines, and (iv) study how we could design transparent replication mechanisms since, in this distributed context, faults are no longer an exception but become the norm.

  In the context of edge infrastructures, small clusters of machines are connected by slow networks to powerful data centers. The small clusters perform computations close to the data sources, which avoids expensive data transfers, while having the possibility to use the computing power of a data center. This evolution requires revisiting several system mechanisms to support greater heterogeneity in terms of hardware and software architecture, and to support slow edge/core cloud network communication. In the DiVA project, we will therefore (i) design new virtual machine migration mechanisms in order to migrate virtual machines between heterogeneous machines (instruction set and hypervisor), (ii) transparently optimize the data paths of multi-tier applications distributed between the cloud and the edge in order to minimize the cloud/edge network links, and (iii) define new virtualization interfaces to optimize micro virtual machines with a short lifetime.

In the context of DiVa, Jean-Pierre Lozi is leading a task on rack-scale scheduling, with the goal of making it possible for the scheduler to transparently migrate tasks across machines, while preserving access to open files and network connections.

10 Dissemination

10.1 Promoting scientific activities

10.1.1 Scientific events: organisation

Member of the organizing committees

- Julia Lawall is a member of the steering committee of the conference Foundations of Software Engineering (FSE), since 2024.
- Julia Lawall is a publications chair of FSE 2025.

10.1.2 Scientific events: selection

Chair of conference program committees

- Julia Lawall was a program chair of DSN 2024.
- Julia Lawall is a program chair of EuroSys 2025.
- Julia Lawall is a program chair of the ASE 2025 journal-first track.

Member of the conference program committees

- Julia Lawall was a PC member of GPCE 2024.
- Julia Lawall was a PC member of the ML Family Workshop 2024.
- Julia Lawall was a PC member of CARI 2024.
- Julia Lawall was a PC member of the industry track of ICSME 2024.
- Julia Lawall is a PC member of MSR 2025.
- Jean-Pierre Lozi was a PC member of DSN 2024.
- Jean-Pierre Lozi is a PC member of EuroSys 2025.

10.1.3 Journal

Member of the editorial boards

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

Reviewer - reviewing activities

- Julia Lawall reviewed papers for Science of Computer Programming the Journal of Computer Languages, and IEEE Access.

10.1.4 Invited talks

- Julia Lawall gave an invited talk at QCon London2024 on "Opening the Box: Diagnosing Operating-System Task-Scheduler Behavior on Highly Multicore Machines".
- Julia Lawall gave an invited talk at SRECon24 Europe/Middle East/Africa on "Opening the Box: Diagnosing Operating-System Task-Scheduler Behavior on Highly Multicore Machines".
- Julia Lawall gave a talk at the University of Copenhagen on "Towards Verification of Linux Kernel Code".
- Julia Lawall gave a talk at the 2024 Frama-C Days on "Towards Verification of Linux Kernel Code".
- Himadri Pandya gave an invited talk at Kernel Recipes 2024 on "Dual level of task scheduling for VM workloads – Pain points & Solutions".

10.1.5 Leadership within the scientific community

- Julia Lawall is the secretary of IFIP technical committee 2.
- Julia Lawall is a member of the advisory board of Software Heritage.

10.1.6 Scientific expertise

- Julia Lawall was a member of the jury for an Associate Professor (Lektor) position at the University of Copenhagen.

10.2 Teaching - Supervision - Juries

10.2.1 Supervision

PhD students in Whisper:

- Maxime Derri, Orange Labs, CIFRE, from Nov 2024. On the BPF verifier. Supervised by Kahina Lazri (Orange Labs) and Julia Lawall.
- Papa Assane Fall, INRIA. On a user-level interface to kernel memory management. Supervised by Alain Tchana (Grenoble) and Jean-Pierre Lozi.
- Victor Laforet, INRIA. On optimized lock algorithms. Supervised by Jean-Pierre Lozi and Julia Lawall.
- Keisuke Nishimura, INRIA, from Dec 2024. On verification of Linux kernel code. Supervised by Julia Lawall and Jean-Pierre Lozi.
- Himadri Pandya, INRIA. On task scheduling and virtualization. Supervised by Julia Lawall and Jean-Pierre Lozi.

PhD students outside Whisper:

- Amelie Gonzalez, Inria Rennes. On Rust and networking. Supervised by David Bromberg (Rennes) and Djob Mvondo (Rennes), and Julia Lawall.
- Cesaire Mounah, Inria Rennes. On wireguard performance. Supervised by David Bromberg (Rennes) and Djob Mvondo (Rennes), and Julia Lawall.

Other:

- Jean-Pierre Lozi and Julia Lawall supervised the M2 internship of Clement Gachod, completing his masters at TU Munich.
- Julia Lawall supervised the internship of Corinn Tiffany, an undergraduate student at Brown University.

10.2.2 Juries

- Julia Lawall was a member of the jury of the PhD student Abdoul Kader Kabore at the University of Luxembourg in April 2024.
- Julia Lawall was a member of the jury of the PhD student Josselin Giet (ENS) defended in September 2024.
- Julia Lawall was a reporter for the PhD thesis of Léo Andrès (University of Saclay) defended in December 2024.

10.3 Popularization

10.3.1 Participation in Live events

- Julia Lawall gave a talk at the Linux Lund Conference 2024 on "Should we balance? An adventure with formal verification of Linux kernel code".
- Julia Lawall gave a talk at the Linux Plumbers Conference 2024 on "Program verification for the Linux kernel: Potential costs and benefits".
- Julia Lawall led a panel at the Open Source Summit Europe 2024 on "Kernel Internship Report (Outreachy)".
- Himadri Pandya gave a talk at the Linux Plumbers Conference 2024 in the sched-ext Microconference on "A case for using para-virtualized scheduling information with sched-ext schedulers".
- Tathagata Roy gave a talk at the Linux Plumbers Conference 2024 in the Rust Microconference on "Coccinelle for Rust".

10.3.2 Others science outreach relevant activities

- Julia Lawall coordinated the Linux Kernel's participation in the Outreachy internship program for the winter 2023-2024 round.

11 Scientific production

11.1 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 (cit. on p. 3).

[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 (cit. on p. 3).

[3] J. Lawall and G. Muller. 'Coccinelle: 10 Years of Automated Evolution in the Linux Kernel'. In: 2018 USENIX Annual Technical Conference. Boston, MA, United States, 11th July 2018. URL: https://inria.hal.science/hal-01853271 (cit. on pp. 3, 4, 11).

[4] J. 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, 12th July 2017, p. 12. URL: https://inria.hal.science/hal-01556589 (cit. on p. 11).

[5] 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 (cit. on p. 3).

[6] J.-P. Lozi, F. David, G. Thomas, J. L. Lawall and G. Muller. 'Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications'. In: *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. 2012 USENIX Annual Technical Conference (USENIX ATC '12). Boston, MA, United States: ACM, 2012. DOI: 10.5555/2342821.2342827. URL: https://inria.hal.science/hal-00779908 (cit. on p. 4).

[7] 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 (cit. on p. 3).

[8] 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 (cit. on p. 3).

[9] 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 (cit. on p. 3).

[10] 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 (cit. on p. 6).

[11] 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 (cit. on p. 5).

[12] 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 (cit. on p. 3).

11.2 Publications of the year

International journals

[13] Y. Lyu, H. J. Kang, R. Widyasari, J. Lawall and D. Lo. 'Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel'. In: *IEEE Transactions on Software Engineering* 50.9 (Sept. 2024), pp. 2219–2239. DOI: 10.6084/m9.figshare.23889792.v2. URL: https://inria.hal.science/hal-04856284 (cit. on p. 8).

[14] N. F. Yıldıran, J. Oh, J. Lawall and P. Gazzillo. 'Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times'. In: *Proceedings of the ACM on Software Engineering*. ACM International Conference on the Foundations of Software Engineering (FSE) 1.FSE (12th July 2024), pp. 427–449. DOI: 10.1145/3643746. URL: https://inria.hal.science/hal-04856277 (cit. on p. 7).

International peer-reviewed conferences

[15] J. Lawall, K. Nishimura and J.-P. Lozi. 'Should We Balance? Towards Formal Verification of the Linux Kernel Scheduler'. In: *Static Analysis - 31st International Symposium*. SAS 2024 - 31st Static Analysis Symposium. Pasadena (California), United States, 20th Oct. 2024. URL: https://inria.hal.science/hal-04708445 (cit. on pp. 6, 7).

11.3 Cited publications

[16] 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 (cit. on p. 6).

[17] V. Chipounov and G. Candea. 'Reverse Engineering of Binary Device Drivers with RevNIC'. In: *EuroSys*. 2010, pp. 167–180 (cit. on p. 5).

[18] J. Corbet. 'The extensible scheduler class'. In: *Linux Weekly News* (Feb. 2023). URL: https://lwn.net/Articles/922405/ (cit. on p. 4).

[19] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles and B. Yakobowski. 'Frama-C: A Software Analysis Perspective'. In: *Software Engineering and Formal Methods - 10th International Conference (SEFM)*. Thessaloniki, Greece, Oct. 2012. URL: http://pathcrawler-online.com/pubs/final_sefm12.pdf (cit. on p. 5).

[20] T. David, R. Guerraoui and V. Trigonakis. 'Everything you always wanted to know about synchronization but were afraid to ask'. In: *Symposium on Operating Systems Principles (SOSP)*. 2013, pp. 33–48 (cit. on p. 4).

[21] I. Dillig, T. Dillig and A. Aiken. 'Sound, complete and scalable path-sensitive analysis'. In: *PLDI*. June 2008, pp. 270–280 (cit. on p. 5).

[22] 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 (cit. on pp. 5, 6).

[23] H. Guiroux, R. Lachaize and V. Quéma. 'Multicore Locks: The Case Is Not Closed Yet'. In: *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. Ed. by A. Gulati and H. Weatherspoon. USENIX Association, 2016, pp. 649–662 (cit. on p. 4).

[24] L. Guo, J. Lawall and G. Muller. 'Oops! Where Did That Code Snippet Come From?' In: *MSR 2014 - 11th Working Conference on Mining Software Repositories*. Hyderabad, India: ACM, May 2014, pp. 52–61. DOI: 10.1145/2597073.2597094. URL: https://inria.hal.science/hal-01080397 (cit. on p. 5).

[25] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner and C. Kozyrakis. 'GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling'. In: *Symposium on Operating Systems Principles*. Association for Computing Machinery, 2021, pp. 588–604 (cit. on p. 4).

[26] 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 (cit. on p. 5).

[27] A. Kadav and M. M. Swift. 'Understanding modern device drivers'. In: *ASPLOS*. 2012, pp. 87–98 (cit. on p. 5).

[28] J. Koschel, P. Borrello, D. C. D'Elia, H. Bos and C. Giuffrida. 'Uncontained: Uncovering Container Confusion in the Linux Kernel'. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5055–5072. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/koschel (cit. on p. 5).

[29] J. 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 and Experience* 43.1 (Jan. 2013), pp. 67–92. DOI: 10.1002/spe.2102. URL: https://hal.science/hal-00940320 (cit. on p. 5).

[30] J. Lawall, B. Laurie, R. R. Hansen, N. Palix and G. Muller. 'Finding Error Handling Bugs in OpenSSL Using Coccinelle'. In: *European Dependable Computing Conference*. Valencia, Spain, Apr. 2010, pp. 191–196. DOI: 10.1109/EDCC.2010.31. URL: https://hal.science/hal-00940375 (cit. on p. 5).

[31] T. Li, J. Bai, Y. Sui and S. Hu. 'Path-sensitive and alias-aware typestate analysis for detecting OS bugs'. In: *ASPLOS*. ACM, 2022, pp. 859–872 (cit. on p. 5).

[32] 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 (cit. on p. 6).

[33] 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 (cit. on p. 6).

[34] F. Mérillon, L. Réveillère, C. Consel, R. Marlet and G. Muller. 'Devil: An IDL for Hardware Programming'. In: *4th Symposium on Operating System Design and Implementation (OSDI)*. 2000, pp. 17–30 (cit. on p. 5).

[35] J. Oh, N. F. Yildiran, J. Braha and P. Gazzillo. 'Finding broken Linux configuration specifications by statically analyzing the Kconfig language'. In: *ESEC/FSE*. Ed. by D. Spinellis, G. Gousios, M. Chechik and M. D. Penta. ACM, 2021, pp. 893–905 (cit. on p. 5).

[36] 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 (cit. on p. 5).

[37] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur and G. Heiser. 'Automatic device driver synthesis with Termite'. In: *SOSP*. 2009, pp. 73–86 (cit. on p. 5).

[38] 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 (cit. on p. 5).

[39] 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://inria.hal.science/hal-01086082 (cit. on p. 5).

[40] 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 (cit. on p. 5).

[41] W. Wang and M. Godfrey. 'A Study of Cloning in the Linux SCSI Drivers'. In: *Source Code Analysis and Manipulation (SCAM)*. IEEE, 2011 (cit. on p. 5).