IN PARTNERSHIP WITH:
CNRS

Université Pierre et Marie Curie (Paris 6)

Activity Report 2015

Project-Team WHISPER

Well Honed Infrastructure Software for Programming Environments and Runtimes

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

RESEARCH CENTER
Paris - Rocquencourt

THEME
Distributed Systems and middleware

Table of contents

Project-Team WHISPER

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

Keywords:

Computer Science and Digital Science:

1. - Architectures, systems and networks
1.1.1. - Multicore
2. - Software
2.1.10. - Domain-specific languages
2.1.11. - Proof languages
2.1.6. - Concurrent programming
2.2.1. - Static analysis
2.2.3. - Run-time systems
2.3.1. - Embedded systems
2.3.3. - Real-time systems
2.4. - Reliability, certification
2.4.3. - Proofs
2.5. - Software engineering
2.6. - Infrastructure software
2.6.1. - Operating systems
2.6.2. - Middleware
2.6.3. - Virtual machines

Other Research Topics and Application Domains:

5. - Industry of the future
5.2.1. - Road vehicles
5.2.3. - Aviation
5.2.4. - Aerospace
6.1. - Software industry
6.1.1. - Software engineering
6.1.2. - Software evolution, maintenance
6.3.3. - Network services
6.5. - Information systems
6.6. - Embedded systems

1. Members

Research Scientists

Gilles Muller [Team leader, Inria, Senior Researcher, HdR]
Julia Lawall [Inria, Senior Researcher]

Faculty Members

Pierre-Evariste Dagand [CNRS, Researcher]
Bertil Folliot [Univ. Paris VI, Professor, HdR]

Engineer
Quentin Lambert [Inria]

PhD Students
Brice Berna [ENS Cachan]
Antoine Blin [Renault]
Florian David [Univ. Paris VI, until Jun 2015]
Valentin Rothberg [Inria, until Jun 2015]
Peter Senna Tschudin [Inria, until Dec 2015]

Visiting Scientist
Gregory Kroah-Hartman [Linux Foundation, from Mar 2015 until May 2015]

Administrative Assistant
Helene Milome [Inria]

2. Overall Objectives

2.1. 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 [43], [37], [78]. 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], [8], [9] for transforming C programs (see Section 6.1.1), or domain-specific languages such as Devil [7] and Bossa [6] for designing drivers and kernel schedulers. Due to the small size of the team, Whisper will mainly target operating system kernels and runtimes for programming languages. We will 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. Scientific Foundations

3.1.1. 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 [30], [32], [33] or dynamic analysis [52], [54], [59]. 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* [32]. 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* [42] 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 [65]. More recently, more general forms of symbolic execution [30] 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 [27], [28].

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. [35] 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 [31], 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 [19]. 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) [48], but can also highlight trends that are not apparent at the low level of individual program statements. We have explored combining such statistical methods with more traditional static analysis in identifying faults in the usage of constants in Linux kernel code [47].

3.1.2. 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 for both programming and proving

as quickly as possible. Domain-specific languages (DSLs), also known as *little languages*, are a means to that end [5] [56].

3.1.2.1. Traditional approach.

Using little languages to aid in software development is a tried-and-trusted technique [67] 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 [7]. 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.

3.1.2.2. Embedding DSLs.

The idea of a DSL has yet to realize its full potential in the OS community. Indeed, with the notable exception of interface definition languages for remote procedure call (RPC) stubs, most OS code is still written in a low-level language, such as C. Where DSL code generators are used in an OS, they tend to be extremely simple in both syntax and semantics. We conjecture that the effort to implement a given DSL usually outweighs its benefit. We identify several serious obstacles to using DSLs to build a modern OS: specifying what the generated code will look like, evolving the DSL over time, debugging generated code, implementing a bug-free code generator, and testing the DSL compiler.

Filet-o-Fish (FoF) [3] addresses these issues by providing a framework in which to build correct code generators from semantic specifications. This framework is presented as a Haskell library, enabling DSL writers to *embed* their languages within Haskell. DSL compilers built using FoF are quick to write, simple, and compact, but encode rigorous semantics for the generated code. They allow formal proofs of the run-time behavior of generated code, and automated testing of the code generator based on randomized inputs, providing greater test coverage than is usually feasible in a DSL. The use of FoF results in DSL compilers that OS developers can quickly implement and evolve, and that generate provably correct code. FoF has been used to build a number of domain-specific languages used in Barrelfish, [21] an OS for heterogeneous multicore systems developed at ETH Zurich.

The development of an embedded DSL requires a few supporting abstractions in the host programming language. FoF was developed in the purely functional language Haskell, thus benefiting from the type class mechanism for overloading, a flexible parser offering convenient syntactic sugar, and purity enabling a more algebraic approach based on small, composable combinators. Object-oriented languages – such as Smalltalk [36] and its descendant Pharo [24] – or multi-paradigm languages – such as the Scala programming language [58] – also offer a wide range of mechanisms enabling the development of embedded DSLs. Perhaps suprisingly, a low-level imperative language – such as C – can also be extended so as to enable the development of embedded compilers [22].

3.1.2.3. Certifying DSLs.

Whilst automated and interactive software verification tools are progressively being applied to larger and larger programs, we have not yet reached the point where large-scale, legacy software – such as the Linux kernel –

could formally be proved "correct". DSLs enable a pragmatic approach, by which one could realistically strengthen a large legacy software by first narrowing down its critical component(s) and then focus our verification efforts onto these components.

Dependently-typed languages, such as Coq or Idris, offer an ideal environment for embedding DSLs [29], [25] in a unified framework enabling verification. Dependent types support the type-safe embedding of object languages and Coq's mixfix notation system enables reasonably idiomatic domain-specific concrete syntax. Coq's powerful abstraction facilities provide a flexible framework in which to not only implement and verify a range of domain-specific compilers [3], but also to combine them, and reason about their combination.

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

Furthermore, the fact that Coq is also a system for formalizing mathematics enables one to establish a close, formal connection between embedded DSLs and non-trivial domain-specific models. The possibility of developing software in a truly "model-driven" way is an exciting one. Following this methodology, we have implemented a certified compiler from regular expressions to x86 machine code [4]. Interestingly, our development crucially relied on an existing Coq formalization, due to Braibant and Pous, [26] of the theory of Kleene algebras.

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

3.2. Research direction: developing drivers using Genes

We believe that weaknesses of previous methods for easing device driver development arise from an insufficient understanding of the range and scope of driver functionality, as required by real devices and OSes. We propose a new methodology for understanding device drivers, inspired by the biological field of genomics. Rather than focusing on the input/output behavior of a device, we take the radically new methodology of studying existing device driver code itself. On the one hand, this methodology makes it possible to identify the behaviors performed by real device drivers, whether to support the features of the device and the OS, or to improve properties such as safety or performance. On the other hand, this methodology makes it possible to capture the actual patterns of code used to implement these behaviors, raising the level of abstraction from individual operations to collections of operations implementing a single functionality, which we refer to as *genes*. Because the requirements of the device remain fixed, regardless of the OS, we expect to find genes with common behaviors across different OSes, even when those genes have a different internal structure. This leads to a view of a device driver as being constructed as a composition of genes, thus opening the door to new methodologies to address the problems faced by real driver developers. Among these, we have so far identified the problems of developing drivers, porting existing drivers to other OSes, backporting existing drivers to older OS versions, and long-term maintenance of the driver code.

Our short term goal is to "sequence" the complete set of genes for a set of related drivers. In the longer term, we plan to develop methodologies based on genes for aiding in driver development and maintenance. This work is currently financed by a grant from the Direction Générale de l'Armement (DGA) that supports the PhD of Peter Senna Tschudin. Valentin Rothberg's PhD is supported by an Inria Cordi-S grant.

3.3. 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 [7] to interact with devices, Bossa [6] to implement the scheduling policies, and Zebu [2] to implement some networking protocols. However, much effort is duplicated in implementing the back-ends of the individual DSLs. Our long term goal is to design a unified framework for developing and composing DSLs, following our work on Filet-o-Fish [3]. 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 DSLs.

Beyond the software engineering aspects, a unified platform brings us closer to the implementation of mechanically-verified DSLs. Dagand's recent work using the Coq proof assistant as an x86 macro-assembler [4] is a step in that direction, which belongs to a larger trend of hosting DSLs in dependent type theories [25], [57], [29]. A key benefit of those approaches is to provide – by construction – a formal, mechanized semantics to the DSLs thus developed. This semantics offers a foundation on which to base further verification efforts, whilst allowing interaction with non-verified code. We advocate a methodology based on incremental, piece-wise verification. Whilst building fully-certified systems from the top-down is a worthwhile endeavor [43], we wish to explore a bottom-up approach by which one focuses first and foremost on crucial subsystems and their associated properties.

We plan to apply this methodology for implementing a certified DSL for describing serializers and deserializers of binary datastreams. This work will build on our experience in designing Zebu [2], a DSL for describing text-based protocols. Inspired by our experience implementing a certified regular expression compiler in x86 [4], we wish to extend Zebu to manipulate binary data. Such a DSL should require a single description of a binary format and automatically generate a serializer/deserializer pair. This dual approach – relating a binary format to its semantic model – is inspired by the Parsifal [49] and Nail [20] format languages. A second challenge consists in guaranteeing the functional correctness of the serializer/deserializer pair generated by the DSL: one would wish to prove that any serialized data can be deserialized to itself, and conversely. The RockSalt's project [57] provides the conceptual tools, in a somewhat simpler setting, to address this question.

Packet filtering is another sweet spot for DSLs. First, one needs a DSL for specifying the filtering rules. This is standard practice [55]. However, in our attempt to establish the correctness of the packet filter, we will be led to equip this DSL with a mechanized semantics, formally describing the precise meaning of each construct of the language. Second, packet filters are usually implemented through a matching engine that is, essentially, a bytecode interpreter. To establish the correctness of the packet filter, we shall then develop a mechanized semantics of this bytecode and prove that the *compilation* from filtering rules to bytecode preserves the intended semantics. Because a packet filter lies at the entry-point of a network, safety is crucial: we would like to guarantee that the packet filter cannot crash and is not vulnerable to an attack. Beyond mere safety, functional correctness is essential too: we must guarantee that the high-level filtering rules are indeed applied as expected by the matching engine. A loophole in the compilation could leave the network open to an attack or prevent legitimate traffic from reaching its destination. Finally, the safety of the packet filter *cannot* be established at the expense of performance. Indeed, if the packet filter were to become a bottleneck, the infrastructure it aimed at protecting would easily become subject to Denial of Service (DoS) attacks. Filtering rules should therefore be compiled efficiently: the corresponding optimizations will have to be verified [74].

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, v3.17, comprises over 12 million lines of code, and supports 29 different families of CPU architectures, 73 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. [34], numerous research tools have been applied to the Linux kernel, typically for finding bugs [33], [51], [60], [68] or for computing software metrics [39], [75]. In our work, we have studied generic C bugs in Linux code [9], bugs in function protocol usage [45], [46], issues related to the processing of bug reports [63] and crash dumps [38], and the problem of backporting [15], 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. Section 6.1.2 presents our dissemination efforts to the Linux community.

4.2. Device Drivers

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

The last fifteen years have seen a number of approaches directed towards easing device driver development. Réveillère, who was supervised by G. Muller, proposes Devil [7], a domain-specific language for describing the low-level interface of a device. Chipounov *et al.* propose RevNic, [28] a template-based approach for porting device drivers from one OS to another. Ryzhyk *et al.* propose Termite, [61], [62] 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 [40] 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 [19], Coverity [34], CP-Miner, [50] PR-Miner [51], and Coccinelle [8]. 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 guidelines and tools for the maintenance and the development of new drivers. Section 3.2 describes this research direction.

5. Highlights of the Year

5.1. Highlights of the Year

The main highlight of the year is the continuous spreading of Coccinelle within the developer community of the Linux kernel. We submitted the first patches to the Linux kernel based on Coccinelle in 2007. Since then, over 4500 patches have been accepted into the Linux kernel based on the use of Coccinelle, including around 3000 by over 500 developers from outside our research group. Another testimonial of the impact of our work is the visit of Greg Kroah-Hartman in March and April 2015, as an Inria invited researcher. Kroah-Hartman is one of the leading developers of the Linux kernel, and at the time only one of two developers employed by the Linux Foundation, with the other being Linus Torvalds. Greg participated in the activities of the Whisper team around the use of Coccinelle and research projects related to the Linux kernel, and he is a convinced ambassador of our research work.

Our work on Remote Core Locking (RCL) [10] was accepted in ACM Transaction in Computer Systems (TOCS) which is the most prestigious journal in systems. RCL is currently one of the most efficient locks for multicore architectures.

6. New Software and Platforms

6.1. Platforms

6.1.1. Coccinelle

Our recent research is in the area of code manipulation tools for C code, particularly targeting Linux kernel code. This work has led to the Coccinelle tool that we are continuing to develop. Coccinelle serves both as a basis for our future research and the foundation of our interaction with the Linux developer community.

The need to find patterns of code, and potentially to transform them, is pervasive in software development. Examples abound. When a bug is found, it is often fruitful to see whether the same pattern occurs elsewhere in the code. For example, the recent Heartbleed bug in OpenSSL partly involves the same fragment of code in two separate files. [1] Likewise, when the interface of an API function changes, all of the users of that function have to be updated to reflect the new usage requirements. This generalizes to the case of code modernization, in which a code base needs to be adapted to a new compiler, new libraries, or a new coding standards. Finding patterns of code is also useful in code understanding, *e.g.*, to find out whether a particular function is ever called with a particular lock held, and in software engineering research, *e.g.*, to understand the prevalence of various kinds of code structures, which may then be correlated with other properties of the software. For all of these tasks, there is a need for an easy to use tool that will allow developers to express patterns and transformations that are relevant to their source code, and to apply these patterns and transformations to the code efficiently and without disrupting the overall structure of the code base.

The Coccinelle program matching and transformation tool for C code addresses these needs. Coccinelle has been under development for over 10 years, and is mature software, available in a number of Linux distributions (Ubuntu, Debian, Fedora, etc.). It allows matching and transformation rules to be expressed in terms of fragments of C code, more precisely in the form of a *patch*, in which code to add and remove is highlighted by using + and −, respectively, in the leftmost column, and other, unannotated, code fragments may be provided to describe properties of the context. The C language is extended with a few operators, such as metavariables, for abstracting over subterms, and a notion of positions, which are useful for reporting bugs. The pattern matching rules can interspersed with rules written in Python or OCaml, for further expressiveness. The process of matching patterns against the source code furthermore takes into account some semantic information, such as the types of expressions and reachability in terms of a function's (intraprocedural) control-flow graph, and thus we refer to Coccinelle matching and transformation specifications as *semantic patches*.

[1] http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db902

Coccinelle was originally motivated by the goal of modernizing Linux 2.4 drivers for use with Linux 2.6, and was originally validated on a collection of 60 transformations that had been used in modernizing Linux 2.4 drivers [8]. Subsequent research involving Coccinelle included a formalization of the logic underlying its implementation [1] and a novel mechanism for identifying API usage protocols [45]. More recently, Coccinelle has served as a practical and flexible tool in a number of research projects that somehow involve code understanding or transformation. These include identifying misuses of named constants in Linux code [47], extracting critical sections into procedures to allow the implementation of a centralized locking service [53], generating a debugging interface for Linux driver developers [23], detecting resource release omission faults in Linux and other infrastructure software [64], and understanding the structure of device driver code in our current DrGene project [66].

Throughout the development of Coccinelle, we have also emphasized contact with the developer community, particularly the developers of the Linux kernel. We submitted the first patches to the Linux kernel based on Coccinelle in 2007. Since then, over 4500 patches have been accepted into the Linux kernel based on the use of Coccinelle, including around 3000 by over 500 developers from outside our research group. Over 50 semantic patches are available in the Linux kernel source code itself, with appropriate infrastructure for developers to apply these semantic patches to their code within the normal `make` process. Many of these semantic are also included in a 0-day build-testing system for Linux patches maintained by Intel. [2] Julia Lawall was invited to the Linux Kernel Summit as a core attendee (invitation only) in 2010 and 2014, and has been invited to the internal 2014 SUSE Labs Conference. She has also presented Coccinelle at developer events such as LinuxCon Europe, Kernel Recipes (Paris), FOSDEM (Brussels), and RTWLS, and has supervised three interns using Coccinelle financed by the Linux Foundation, as part of the Outreachy internship program.

Finally, we are aware of several companies that use Coccinelle for modernizing code bases. These include Metaware in Paris, with whom we had a 5-month contract in 2013-2014 for the customization and maintenance of Coccinelle. We hope to be able to organize other such contracts in the future.

6.1.2. Better Linux

Over the past few years, Julia Lawall and Gilles Muller have designed and developed of a number of tools such as Coccinelle, Diagnosys [23] [22] and Hector [64], to improve the process of developing and maintaining systems code. The BtrLinux action aims to increase the visibility of these tools, and to highlight Inria's potential contributions to the open source community. We are developing a web site https://btrlinux.inria.fr/, to centralize the dissemination of the tools, collect documentation, and collect results. This action is supported by Inria by the means of a young engineer (ADT), Quentin Lambert. In the case of Coccinelle, we will focus on enhancing its visibility and its dissemination, by using it to find and fix faults in Linux kernel code, and by submitting the resulting patches to the Linux maintainers. Our work on Diagnosys and Hector is described below.

Diagnosys is a hybrid static and dynamic analysis tool that first collects information about Linux kernel APIs that may be misused, and then uses this information to generate wrapper functions that systematically log at runtime any API invocations or return values that may reflect such misuse. A developer can then use a specific make-like command to build an executable driver that transparently uses these wrapper functions. At runtime, the wrappers write log messages into a crash resilient region of memory that the developer can inspect after any crash. Diagnosys is complementary to Coccinelle in the kind of information that it provides to developers. While Coccinelle directly returns a report for every rule match across the code base, often including false positives that have to be manually isolated by the developer, Diagnosys only reports on conditions that occur in the actual execution of the code. Diagnosys thus produces less information, but the information produced is more relevant to the particular problem currently confronting the developer. As such, it is well suited to the case of initial code development, where the code is changing frequently, and the developer wants to debug a specific problem, rather than ensuring that the complete code base is fault free. Diagnosys is a complete functioning system, but it needs to be kept up to date with changes in the kernel API functions. As part of the

[2] E.g., http://comments.gmane.org/gmane.linux.kernel.kbuild/269

BtrLinux action, we will regularly run the scripts that collect information about how to create the wrappers, and then validate and make public the results.

Hector addresses the problem of leaking resources in error-handling code. Releasing resources when they are no longer needed is critical, so that adequate resources remain available over the long execution periods characteristic of systems software. Indeed, when resource leaks accumulate, they can cause unexpected resource unavailability, and even single leaks can put the system into an inconsistent state that can cause crashes and open the door to possible attacks. Nevertheless, developers often forget to release resources, because doing so often does not make any direct contribution to a program's functionality. A major challenge in detecting resource-release omission faults is to know when resource release is required. Indeed, the C language does not provide any built-in support for resource management, and thus resource acquisition and release are typically implemented using ad hoc operations that are, at best, only known to core developers. Previous work has focused on mining sequences of such functions that are used frequently across a code base, [35], [51] but these approaches have very high rates of false negatives and false positives [48]. We have proposed Hector, a static analysis tool that finds resource-release omission faults based on inconsistencies in the operations performed within a single function, rather than on usage frequency. This strategy allows Hector to have a low false positive rate, of 23% in our experiments, while still being able to find hundreds of faults in Linux and other systems.

Hector was developed as part of the PhD thesis of Suman Saha and was presented at DSN 2013, where it received the William C. Carter award for the best student paper. Hector is complementary to Coccinelle, in that it has a more restricted scope, focusing on only one type of fault, but it uses a more precise static analysis, tailored for this type of fault, to ensure a low false positive rate. Hector, like Coccinelle, is also complementary to Diagnosys, in that it exhaustively reports on faults in a code base, rather than only those relevant to a particular execution, and is thus better suited for use by experienced developers of relatively stable software. Over 70 patches have been accepted into Linux based on the results of Hector. The current implementation, however, is somewhat in a state of disarray. As part of the BtrLinux action, we are currently working on returning the code to working condition and then will actively use it to find faults in Linux. Based on these results, we will either submit appropriate patches to the Linux developers or notify the relevant developer when the corresponding fix is not clear.

6.2. New Software

6.2.1. coq-bitset library

As part of Arthur Blot's internship, we have developed the `coq-bitset` library, a certified library implementing bitsets in the Coq proof assistant [17]. It enables abstract and formal reasoning about efficient low-level code within a proof assistant, thus paving the way for further certified results in the field of low-level system code (such as device drivers).

As part of this effort, we have also extended a pre-existing formalization of bit vectors in Coq [41] with a trustworthy extraction mechanism. This enables manipulating and reasoning about native integers in the Coq proof assistant, while supporting an efficient execution in OCaml.

Both libraries have been made available on Github as well as on the `Coq-opam` repository.

7. New Results

7.1. Software engineering for infrastructure software

Tracking code fragments of interest is important in monitoring a software project over multiple versions. Various approaches, including our previous work on Herodotos, exploit the notion of Longest Common Subsequence, as computed by readily available tools such as GNU Diff, to map corresponding code fragments. Nevertheless, the efficient code differencing algorithms are typically line-based or word-based, and thus do

not report changes at the level of language constructs. Furthermore, they identify only additions and removals, but not the moving of a block of code from one part of a file to another. Code fragments of interest that fall within the added and removed regions of code have to be manually correlated across versions, which is tedious and error-prone. When studying a very large code base over a long time, the number of manual correlations can become an obstacle to the success of a study. In a paper published at the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) [14], we investigate the effect of replacing the current line-based algorithm used by Herodotos by tree-matching, as provided by the algorithm of the differencing tool GumTree. In contrast to the line-based approach, the tree-based approach does not generate any manual correlations, but it incurs a high execution time. To address the problem, we propose a hybrid strategy that gives the best of both approaches.

Understanding the severity of reported bugs is important in both research and practice. In particular, a number of recently proposed mining-based software engineering techniques predict bug severity, bug report quality, and bug-fix time, according to this information. Many bug tracking systems provide a field "severity" offering options such as "severe", "normal", and "minor", with "normal" as the default. However, there is a widespread perception that for many bug reports the label "normal" may not reflect the actual severity, because reporters may overlook setting the severity or may not feel confident enough to do so. In many cases, researchers ignore "normal" bug reports, and thus overlook a large percentage of the reports provided. On the other hand, treating them all together risks mixing reports that have very diverse properties. In a study published at the Working Conference on Mining Software Repositories (MSR) 2015 [16], we investigate the extent to which "normal" bug reports actually have the "normal" severity. We find that many "normal" bug reports in practice are not normal. Furthermore, this misclassification can have a significant impact on the accuracy of mining-based tools and studies that rely on bug report severity information.

Software is continually evolving, to fix bugs and add new features. Industry users, however, often value stability, and thus may not be able to update their code base to the latest versions. This raises the need to selectively backport new features to older software versions. Traditionally, backporting has been done by cluttering the backported code with preprocessor directives, to replace behaviors that are unsupported in an earlier version by appropriate workarounds. This approach however involves writing a lot of error-prone backporting code, and results in implementations that are hard to read and maintain. In a paper published at the 2015 European Dependable Computing Conference (EDCC) [15], we consider this issue in the context of the Linux kernel, for which older versions are in wide use. We present a new backporting strategy that relies on the use of a backporting compatability library and on code that is automatically generated using the program transformation tool Coccinelle. This approach reduces the amount of code that must be manually written, and thus can help the Linux kernel backporting effort scale while maintaining the dependability of the backporting process.

Logging is a common and important programming practice, but choosing how to log is challenging, especially in a large, evolving software code base that provides many logging alternatives. Insufficient logging may complicate debugging, while logging incorrectly may result in excessive performance overhead and an overload of trivial logs. The Linux kernel has over 13 million lines of code, over 1100 different logging functions, and the strategies for when and how to log have evolved over time. To help developers log correctly we propose, in a paper published at BEneVol 2015 [18], a framework that will learn existing logging practices from the software development history, and that will be capable of identifying new logging strategies, even when the new strategies just start to be adopted.

7.2. Java runtime support

Java class loaders are commonly used in application servers to load, unload and update a set of classes as a unit. However, unloading or updating a class loader can introduce stale references to the objects of the outdated class loader. A stale reference leads to a memory leak and, for an update, to an inconsistency between the outdated classes and their replacements. To detect and eliminate stale references, in a paper published at DSN 2015 [12], we propose Incinerator, a Java virtual machine extension that introduces the notion of an outdated class loader. Incinerator detects stale references and sets them to null during a garbage collection cycle. We evaluate

Incinerator in the context of the OSGi framework and show that Incinerator correctly detects and eliminates stale references, including a bug in Knopflerfish. We also evaluate the performance of Incinerator with the DaCapo benchmark on VMKit and show that Incinerator has an overhead of at most 3.3%.

7.3. Parallel and Distributed Computing

The scalability of multithreaded applications on current multicore systems is hampered by the performance of lock algorithms, due to the costs of access contention and cache misses. In an article published in ACM Transactions on Computer Systems (TOCS), we present a new locking technique, Remote Core Locking (RCL) [10], that aims to accelerate the execution of critical sections in legacy applications on multicore architectures. The idea of RCL is to replace lock acquisitions by optimized remote procedure calls to a dedicated server hardware thread. RCL limits the performance collapse observed with other lock algorithms when many threads try to acquire a lock concurrently and removes the need to transfer lock-protected shared data to the hardware thread acquiring the lock because such data can typically remain in the server's cache. Other contributions presented in this article include a profiler that identifies the locks that are the bottlenecks in multithreaded applications and that can thus benefit from RCL, and a reengineering tool that transforms POSIX lock acquisitions into RCL locks. Eighteen applications were used to evaluate RCL: the nine applications of the SPLASH-2 benchmark suite, the seven applications of the Phoenix 2 benchmark suite, Memcached, and Berkeley DB with a TPC-C client. Eight of these applications are unable to scale because of locks and benefit from RCL on an x86 machine with four AMD Opteron processors and 48 hardware threads. By using RCL instead of Linux POSIX locks, performance is improved by up to 2.5 times on Memcached, and up to 11.6 times on Berkeley DB with the TPC-C client. On a SPARC machine with two Sun Ultrasparc T2+ processors and 128 hardware threads, three applications benefit from RCL. In particular, performance is improved by up to 1.3 times with respect to Solaris POSIX locks on Memcached, and up to 7.9 times on Berkeley DB with the TPC-C client.

Software Transactional Memory (STM) is an optimistic concurrency control mechanism that simplifies parallel programming. Still, there has been little interest in its applicability for reactive applications in which there is a required response time for certain operations. In an article published in ACM Transactions on Parallel Computing (TOPC) [11], we propose supporting such applications by allowing programmers to associate time with atomic blocks in the forms of deadlines and QoS requirements. Based on statistics of past executions, we adjust the execution mode of transactions by decreasing the level of optimism as the deadline approaches. In the presence of concurrent deadlines, we propose different conflict resolution policies. Execution mode switching mechanisms allow meeting multiple deadlines in a consistent manner, with potential QoS degradations being split fairly among several threads as contention increases, and avoiding starvation. Our implementation consists of extensions to a STM runtime that allow gathering statistics and switching execution modes. We also propose novel contention managers adapted to transactional workloads subject to deadlines. The experimental evaluation shows that our approaches significantly improve the likelihood of a transaction meeting its deadline and QoS requirement, even in cases where progress is hampered by conflicts and other concurrent transactions with deadlines.

A challenge in designing a peer-to-peer (P2P) system is to ensure that the system is able to tolerate selfish nodes that strategically deviate from their specification whenever doing so is convenient. In a paper published at SRDS 2015 [13], we propose *RACOON*, a framework for the design of P2P systems that are resilient to selfish behaviours. While most existing solutions target specific systems or types of selfishness, *RACOON* proposes a generic and semi-automatic approach that achieves robust and reusable results. Also, *RACOON* supports the system designer in the performance-oriented tuning of the system, by proposing a novel approach that combines Game Theory and simulations. We illustrate the benefits of using *RACOON* by designing two P2P systems: a live streaming and an anonymous communication system. In simulations and a real deployment of the two applications on a testbed comprising 100 nodes, the systems designed using *RACOON* achieve both resilience to selfish nodes and high performance.

7.4. From Sets to Bits in Coq

Sets form the building block of mathematics, while finite sets are a fundamental data structure of computer science. In the world of mathematics, finite sets enjoy appealing mathematical properties, such as a proof-irrelevant equality and extensionality of functions. Computer scientists, on the other hand, have devised efficient algorithms for set operations based on the representation of finite sets as bit vectors and on bit twiddling, exploiting the hardware's ability to efficiently process machine words.

With interactive theorem provers, sets are reinstituted as mathematical objects. While there are several finite set libraries in Coq, these implementations are far removed from those used in efficient code. Recent work on modeling low-level architectures, such as x86 [41] processors, however, have brought the world of bit twiddling within reach of our proof assistants. We are now able to specify and reason about low-level programs.

In this work, we have implemented bitsets and their associated operations in the Coq proof assistant, thus allowing us to transparently navigate between the concrete world of bit vectors and the abstract world of finite sets. This work grew from a puzzled look at the first page of Warren's *Hacker's Delight* [77], where lies the cryptic formula $x\&(x-1)$ to turn off the rightmost bit in a word. How do we translate the English specification given in the book into a formal definition? How do we prove that this formula meets its specification? Could Coq generate efficient and trustworthy code from it? And how efficiently could we simulate it within Coq itself?

In our work, we have established a bijection between bitsets and sets over finite types. Following a refinement approach, we have shown that a significant part of SSReflect *finset* library can be refined to operations manipulating bitsets. We have also developed a trustworthy extraction of bitsets down to OCaml's machine integers. While we were bound to axiomatize machine integers, we adopted a methodology based on exhaustive testing to gain greater confidence in our model. Finally, we have demonstrateed the usefulness of our library through two applications, a certified implementation of Bloom filters and a verified implementation of the n-queens algorithm.

8. Bilateral Contracts and Grants with Industry

8.1. Bilateral Contracts with Industry

Julia Lawall participates in the OSADL project SIL2LinuxMP (http://www.osadl.org/SIL2LinuxMP.sil2-linux-project.0.html). This project aims at the certification of the base components of an embedded GNU/Linux RTOS running on a single-core or multi-core industrial COTS computer board.

Together with Julien Sopena from REGAL, we are collaborating with Renault, in the context of the PhD of Antoine Blin (CIFRE), on hierarchical scheduling in multicore platforms for real-time embedded systems. This work is a dissemination of our previous research on the Bossa domain-specific language [6].

9. Partnerships and Cooperations

9.1. National Initiatives

9.1.1. ANR

InfraJVM - (2012 - 2015)

Members: LIP6 (Regal-Whisper), Ecole des Mines de Nantes (Constraint), IRISA (Triskell), LaBRI (LSR).

Coordinator: Gaël Thomas

Whisper members: Julia Lawall, Gilles Muller

Funding: ANR Infra, 202 000 euros.

Objectives: The design of the Java Virtual Machine(JVM) was last revised in 1999, at atime when a single program running on a uniprocessor desktop machine was the norm. Today's computing environment, however, is radically different, being characterized by many different kinds of computing devices, which are often mobile and which need to interact within the context of a single application. Supporting such applications, involving multiple mutually untrusted devices, requires resource management and scheduling strategies that were not planned for in the 1999 JVM design. The goal of InfraJVM is to design strategies that can meet the needs of such applications and that provide the good performance that is required in an MRE. The PhD of Florian David was supported in part by InfraJVM.

Chronos network, Time and Events in Computer Science, Control Theory, Signal Processing, Computer Music, and Computational Neurosciences and Biology - (2015 - 2016)

Coordinator: Gerard Berry

Whisper member: Gilles Muller

Funding: ANR 2014, Défi "Société de l'information et de la communication".

The Chronos interdisciplinary network aims at placing in close contact and cooperation researchers of a variety of scientific fields: computer science, control theory, signal processing, computer music, neurosciences, and computational biology. The scientific object of study will be the understanding, modeling, and handling of time- and event-based computation across the fields.

Chronos will work by organizing a regular global seminar on subjects ranging from open questions to concrete solutions in the research fields, workshops gathering subsets of the Chronos researchers to address specific issues more deeply, a final public symposium presenting the main contributions and results, and an associated compendium.

9.1.2. Multicore Inria Project Lab

The Multicore IPL is an Inria initiative, led by Gilles Muller, whose goal is to develop techniques for deploying parallel programs on heterogeneous multicore machines while preserving scalability and performance. The IPL brings together researchers from the ALF, Algorille, CAMUS, Compsys, DALI, REGAL, Runtime and Whisper Inria Teams. These connections provide access to a diversity of expertise on open source development and parallel computing, respectively. In this context, Gilles Muller and Julia Lawall are working with Jens Gustedt and Mariem Saeid of Inria Lorraine and on developing a domain-specific language that eases programming with the ordered read-write lock (ORWL) execution model. The goal of this work is to provide a single execution model for parallel programs and to allow them to be deployed on multicore machines with varying architectures.

9.2. International Initiatives

9.2.1. Inria International Partners

9.2.1.1. Informal International Partners

David Lo, of Singapore Management University, is an expert in the use of statistical methods in understanding software and associated artifacts, with over 140 publications in this area. Julia Lawall has worked with Lo over the past 5 years, exploiting the complementarity of her expertise in Linux code and in program analysis with Lo's expertise in statistical methods, resulting in 10 joint publications [47], [44], [66], [70], [71], [72], [73], [69], [76]. This collaboration has been reinforced in the form of a Merlion collaboration grant from the Institut Français for the years 2013 and 2014, resulting in the exchange of researchers and PhD students between Whisper and Singapore Management University.

Wouter Swierstra is lecturer in the Software Technology Group of Utrecht University. His work lies at the crossroad between dependent type theory, generic programming, and domain-specific languages embedded in type theory. As part of his PEPS JCJC, Pierre-Évariste Dagand visited him to apply for a joint Van Gogh grant on the topic of extending type theory with language interoperability, allowing unsafe operations to be performed in a type-safe framework.

Timothy Roscoe is a Professor in the Institute for Pervasive Computing at ETH Zurich. His research areas are operating systems, distributed systems, and networking. As part of his PEPS JCJC, Pierre-Évariste Dagand visited the Systems group at ETH to explore avenues for collaboration on applying synchronous programming concepts to the design and implementation of network stacks.

9.3. International Research Visitors

9.3.1. Visits of International Scientists

Greg Kroah-Hartman visited the Whisper team in March and April 2015, as an Inria invited researcher. Kroah-Hartman is one of the leading developers of the Linux kernel, and at the time only one of two developers employed by the Linux Foundation, with the other being Linus Torvalds. During his visit, he gave a number of courses and seminars at UPMC, Paris Diderot University, and Ecole Normale Supérieur, and a keynote in a conference on the Internet of Things organized by the GTLL. He also participated throughout his visit in the activities of the Whisper team around the use of Coccinelle and research projects related to the Linux kernel.

9.3.1.1. Internships

Iago Abal, a PhD student at the IT University in Copenhagen, Denmark, visited the Whisper team from January 14, 2015 to March 1, 2015.

9.3.1.2. Research stays abroad

As part of Academics Without Borders, Pierre-Évariste Dagand was a visiting researcher at the University of Cape Coast (Ghana) during 2 months. Aside from his teaching duties, his role was to foster the research activity of the university's Computer Science department. He was thus in charge of the organisation of a weekly research seminar, whose purpose was to perform scientific dissemination and to transmit academic best practices.

10. Dissemination

10.1. Promoting Scientific Activities

10.1.1. Scientific events organisation

10.1.1.1. Member of the organizing committees

Gilles Muller was a member of the organizing committe of the Programming Language and Operating Systems workshop (PLOS). He was the sponsor co-chair and the grant co-chair of the EuroSys conference.

Julia Lawall was the publicity co-chair of the EuroSys conference.

10.1.2. Scientific events selection

10.1.2.1. Member of the conference program committees

Julia Lawall was a member of the program committee for the following conferences that took place in 2015: Automated Software Engineering (ASE), EuroSys, IEEE International Conference on Distributed Computing System (ICDCS), International Conference on Software Maintenance and Evolution (ICSME, ERA track), and OOPSLA.

Gilles Muller was a member of the program committee for the following conferences that took place in 2015: European Dependable Conference (EDCC), Dependable Systems and Network (DSN), Timely Results In Operating Systems (TRIOS). He was also a member of the program committee for the upcoming Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016).

Gilles Muller was a member of the EuroSys 2015 PhD prize R. Needham award.

Pierre-Évariste Dagand was a member of the EuroSys 2015 shadow program committee and member of the program committee of the Programming Languages and Operating Systems (PLOS) workshop.

10.1.3. Journal

10.1.3.1. Member of the editorial boards

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

10.1.3.2. Reviewer - Reviewing activities

Julia Lawall reviewed an article for the journal Empirical Software Engineering and an article for the journal Computer Languages, Systems & Structures.

Pierre-Évariste Dagand reviewed an article for the Journal of Computer Programming and an article for Computer Surveys.

10.1.4. Invited talks

Julia Lawall gave an invited talk on "Analysis and Manipulation of Linux Kernel Source Code using Coccinelle" at the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM, http://www.ieee-scam.org/2015/).

Julia Lawall gave an invited talk on "Coccinelle for the Working Programmer" at Curry On (http://www.curry-on.org/)

Julia Lawall was invited to give several presentations on Coccinelle at Intel, in Portland, Oregon, in June 2015.

Gilles Muller was invited to give a talk at the NICTA Summer school in Sydney in January 2015, and at the FuDiCo IV workshop in Cornell in October 2015 on his joint work with Renault on mixing real-time and best effort applications on an embedded multicore processor.

Pierre-Évariste Dagand gave an invited talk at the EuroSys program committee meeting on the subject of certified binary parsing, in Cambridge, UK, in January 2015.

10.1.5. Leadership within the scientific community

Julia Lawall is a member of the IFIP TC2 working group WG2.11, on Program Generation. She participated in the WG2.11 meetings in January and November, 2015.

Gilles Muller is a member of the IFIP 10.4 working group on Dependable Computing and Fault Tolerance. He participated in the meetings in January and July, 2015. He gave a talk during the July meeting on his joint work with Renault on mixing real-time and best effort applications on an embedded multicore processor.

10.1.6. Scientific expertise

Julia Lawall served on selection committees for a professor position at UPMC, a professor position at Rennes, and an assistant professor position at the University of Copenhagen. Julia Lawall also served on the jury for the Prix Inria (Grand Prix Inria – Académie des sciences, Prix Inria - Académie des sciences du jeune chercheur, and Prix de l'innovation Inria – Académie des sciences – Dassault Systèmes), and on the Inria national commission for selecting young engineer positions (ADT).

10.1.7. Research administration

Julia Lawall was a Member at Large of the SIGPLAN Executive Committee, for a three-year term ending in June 2015.

Gilles Muller is a member of the steering committee of the EuroSys conference.

Gilles Muller is representing Inria in the Sorbonne Universités advisory committee for research. He is a member of the project committe board of the Inria Paris Center and a member of the Paris committee for allocating post-doc, PhD and sabbatical.

10.2. Teaching - Supervision - Juries

10.2.1. Teaching

Master/Doctorat : Pierre-Évariste Dagand, **Algorithms**, 60 hours, University of Cape Coast, Ghana

10.2.2. Supervision

Julia Lawall and Gilles Muller co-supervised the PhD work of Valentin Rothberg and Peter Senna Tschudin.

Julia Lawall serves as the external supervisor of Krishna Narasimhan, a PhD student at Goethe University, Frankfurt, supervised by Christoph Reichenbach.

Gilles Muller is the supervisor of the PhD work of Antoine Blin who is supported by a CIFRE grant with Renault.

Pierre-Évariste Dagand and Julia Lawall co-supervised the bachelor's internship of Arthur Blot.

PhD : Florian David, Efficient and Precise Monitoring of Java Locking algorithms on multicore architectures, Université Pierre et Marie Curie, 8 juillet 2015, Gilles Muller and Gaël Thomas

10.2.3. Juries

Julia Lawall was a member of the PhD jury of Shaowei Wang at Singapore Management University.

Gilles Muller was a reviewer of the PhD of Inti Yulien Gonzalez Herrera at the University of Rennes1, a reviewer of the PhD of Amadou Diara at the University of Grenoble, and a reviewer of the PhD of Julien Tanguy at the University of Nantes.

Gilles Muller was a member of the HDR jury of Vincent Gramoli at the Université Pierre et Marie Curie.

10.3. Popularization

Julia Lawall presented a tutorial on Coccinelle at the 2015 KVM Forum in Seattle, Washington, USA in August 2015 ("Getting Started with Coccinelle (KVM edition)", http://kvmforum2015.sched.org/)

Julia Lawall organized a Development Tools Tutorial microconference at the 2015 Linux Plumbers Conference in Seattle, Washington, USA in August 2015 (http://linuxplumbersconf.org/2015/ocw/events/LPC2015/tracks/423). She also gave a talk "Introduction to Coccinelle" at that event.

Julia Lawall served on the program committee for the joint Linuxcon Linux Plumbers session at Linuxcon 2015, and on the program committee of the 17th Real Time Linux Workshop (industry workshop).

Julia Lawall served as a mentor for the Linux kernel in the Outreachy internship program for women and other underrepresented groups in winter and summer 2015 (https://www.gnome.org/outreachy). She also served as the Linux kernel coordinator for the Outreachy internship program, and organized panel presentations about the program at LinuxCon North America in Seattle, Washington, USA in August 2015 (http://events.linuxfoundation.org/events/archive/2015/linuxcon-north-america), and at LinuxCon Europe in Dublin, Ireland in October 2015 (http://events.linuxfoundation.org/events/archive/2015/linuxcon-europe).

Julia Lawall participated in the Linux Kernel developer panel at LinuxCon Europe in Dublin, Ireland in October 2015 (http://events.linuxfoundation.org/events/archive/2015/linuxcon-europe).

11. Bibliography

Major publications by the team in recent years

[1] J. BRUNEL, D. DOLIGEZ, R. R. HANSEN, J. L. LAWALL, G. MULLER. *A foundation for flow-based program matching using temporal logic and model checking*, in "POPL", Savannah, GA, USA, ACM, January 2009, pp. 114–126

[2] L. BURGY, L. RÉVEILLÈRE, J. L. LAWALL, G. MULLER. *Zebu: A Language-Based Approach for Network Protocol Message Processing*, in "IEEE Trans. Software Eng.", 2011, vol. 37, n° 4, pp. 575-591

[3] P.-É. DAGAND, A. BAUMANN, T. ROSCOE. *Filet-o-Fish: practical and dependable domain-specific languages for OS development*, in "Programming Languages and Operating Systems (PLOS)", 2009, pp. 51–55

[4] A. KENNEDY, N. BENTON, J. B. JENSEN, P.-É. DAGAND. *Coq: The World's Best Macro Assembler?*, in "PPDP", Madrid, Spain, ACM, 2013, pp. 13–24

[5] G. MULLER, C. CONSEL, R. MARLET, L. P. BARRETO, F. MÉRILLON, 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, 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, October 2005, pp. 56–65

[7] F. MÉRILLON, L. RÉVEILLÈRE, C. CONSEL, R. MARLET, 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, October 2000, pp. 17–30

[8] Y. PADIOLEAU, J. L. LAWALL, R. R. HANSEN, G. MULLER. *Documenting and Automating Collateral Evolutions in Linux Device Drivers*, in "EuroSys", Glasgow, Scotland, March 2008, pp. 247–260

[9] N. PALIX, G. THOMAS, S. SAHA, C. CALVÈS, J. L. LAWALL, G. MULLER. *Faults in Linux 2.6*, in "ACM Transactions on Computer Systems", June 2014, vol. 32, n° 2, pp. 4:1–4:40

Publications of the year

Articles in International Peer-Reviewed Journals

[10] J.-P. LOZI, F. DAVID, G. THOMAS, J. LAWALL, G. MULLER. *Fast and Portable Locking for Multicore Architectures*, in "ACM Transactions on Computer Systems", January 2016 [*DOI :* 10.1145/2845079], https://hal.inria.fr/hal-01252167

[11] W. MALDONADO, P. MARLIER, P. FELBER, J. LAWALL, G. MULLER, E. RIVIÈRE. *Supporting Time-Based QoS Requirements in Software Transactional Memory*, in "ACM Transactions on Parallel Computing", July 2015, vol. 2, n° 2, 27 p. [*DOI :* 10.1145/2779621], https://hal.inria.fr/hal-01240225

International Conferences with Proceedings

[12] K. ATTOUCHI, G. THOMAS, G. MULLER, J. L. LAWALL, A. BOTTARO. *Preventing Memory and Information LeakageIncinerator – Eliminating Stale References in Dynamic OSGi Applications*, in "Dependable Systems and Networks", Rio de Janeiro, Brazil, 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, September 2015, https://hal.inria.fr/hal-01198922

[13] G. LENA COTA, S. BEN MOKHTAR, J. LAWALL, G. MULLER, G. GIANINI, E. DAMIANI, L. BRUNIE. *A Framework for the Design Configuration of Accountable Selfish-Resilient Peer-to-Peer Systems*, in "SRDS

2015 - 34th International Symposium on Reliable Distributed Systems", Montreal, Canada, 34th International Symposium on Reliable Distributed Systems, September 2015, https://hal.inria.fr/hal-01250717

[14] N. PALIX, J.-R. FALLERI, J. LAWALL. *Improving pattern tracking with a language-aware tree differencing algorithm*, in "SANER 2015 - 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering", Montreal, Canada, SANER 2015 - 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, March 2015, pp. 43-52 [*DOI :* 10.1109/SANER.2015.7081814], https://hal.inria.fr/hal-01213907

[15] L. R. RODRIGUEZ, J. LAWALL. *Increasing Automation in the Backporting of Linux Drivers Using Coccinelle*, in "11th European Dependable Computing Conference - Dependability in Practice", Paris, France, 11th European Dependable Computing Conference - Dependability in Practice, November 2015, https://hal.inria.fr/hal-01213912

[16] R. K. SAHA, J. LAWALL, S. KHURSHID, D. E. PERRY. *Are These Bugs Really "Normal"?*, in "MSR 2015 - The 12th Working Conference on Mining Software Repositories", Florence, Italy, MSR 2015 - The 12th Working Conference on Mining Software Repositories, May 2015, https://hal.inria.fr/hal-01240036

Conferences without Proceedings

[17] A. BLOT, P.-É. DAGAND, J. LAWALL. *From Sets to Bits in Coq*, in "FLOPS 2016", Kochi, Japan, March 2016, https://hal.archives-ouvertes.fr/hal-01251943

[18] P. SENNA TSCHUDIN, J. LAWALL, G. MULLER. *3L: Learning Linux Logging*, in "BElgian-NEtherlands software eVOLution seminar (BENEVOL 2015)", Lille, France, December 2015, https://hal.inria.fr/hal-01239980

References in notes

[19] T. BALL, E. BOUNIMOVA, B. COOK, V. LEVIN, J. LICHTENBERG, C. MCGARVEY, B. ONDRUSEK, S. K. RAJAMANI, A. USTUNER. *Thorough Static Analysis of Device Drivers*, in "EuroSys", 2006, pp. 73–85

[20] J. BANGERT, N. ZELDOVICH. *Nail: A Practical Tool for Parsing and Generating Data Formats*, in "11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)", October 2014, pp. 615–628

[21] A. BAUMANN, P. BARHAM, P.-É. DAGAND, T. HARRIS, R. ISAACS, S. PETER, T. ROSCOE, A. SCHÜPBACH, A. SINGHANIA. *The multikernel: A new OS architecture for scalable multicore systems*, in "SOSP", 2009, pp. 29–44

[22] T. F. BISSYANDÉ, L. RÉVEILLÈRE, J. L. LAWALL, Y.-D. BROMBERG, G. MULLER. *Implementing an embedded compiler using program transformation rules*, in "Software: Practice and Experience", 2013

[23] T. F. BISSYANDÉ, L. RÉVEILLÈRE, J. L. LAWALL, G. MULLER. *Diagnosys: automatic generation of a debugging interface to the Linux kernel*, in "IEEE/ACM International Conference on Automated Software Engineering (ASE)", 2012, pp. 60–69

[24] A. P. BLACK, S. DUCASSE, O. NIERSTRASZ, D. POLLET. *Pharo by Example*, Square Bracket Associates, 2010

[25] E. BRADY, K. HAMMOND. *Resource-Safe Systems Programming with Embedded Domain Specific Languages*, in "14th International Symposium on Practical Aspects of Declarative Languages (PADL)", LNCS, Springer, 2012, vol. 7149, pp. 242–257

[26] T. BRAIBANT, D. POUS. *An Efficient Coq Tactic for Deciding Kleene Algebras*, in "1st International Conference on Interactive Theorem Proving (ITP)", LNCS, Springer, 2010, vol. 6172, pp. 163–178

[27] C. CADAR, D. DUNBAR, D. R. ENGLER. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, in "OSDI", 2008, pp. 209–224

[28] V. CHIPOUNOV, G. CANDEA. *Reverse Engineering of Binary Device Drivers with RevNIC*, in "EuroSys", 2010, pp. 167–180

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

[30] L. A. CLARKE. *A system to generate test data and symbolically execute programs*, in "IEEE Transactions on Software Engineering", 1976, vol. 2, n^{o} 3, pp. 215–222

[31] E. CLARKE, O. GRUMBERG, S. JHA, Y. LU, H. VEITH. *Counterexample-guided abstraction refinement for symbolic model checking*, in "J. ACM", 2003, vol. 50, n^{o} 5, pp. 752–794

[32] P. COUSOT, R. COUSOT. *Abstract Interpretation: Past, Present and Future*, in "CSL-LICS", 2014, pp. 2:1–2:10

[33] I. DILLIG, T. DILLIG, A. AIKEN. *Sound, complete and scalable path-sensitive analysis*, in "PLDI", June 2008, pp. 270–280

[34] D. R. ENGLER, B. CHELF, A. CHOU, S. HALLEM. *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, in "OSDI", 2000, pp. 1–16

[35] D. R. ENGLER, D. Y. CHEN, A. CHOU, B. CHELF. *Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code*, in "SOSP", 2001, pp. 57–72

[36] A. GOLDBERG, D. ROBSON. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983

[37] L. GU, A. VAYNBERG, B. FORD, Z. SHAO, D. COSTANZO. *CertiKOS: A Certified Kernel for Secure Cloud Computing*, in "Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)", 2011, pp. 3:1–3:5

[38] L. GUO, J. L. LAWALL, 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

[39] A. ISRAELI, D. G. FEITELSON. *The Linux kernel as a case study in software evolution*, in "Journal of Systems and Software", 2010, vol. 83, n^{o} 3, pp. 485–501

[40] A. KADAV, M. M. SWIFT. *Understanding modern device drivers*, in "ASPLOS", 2012, pp. 87–98

[41] A. KENNEDY, N. BENTON, J. B. JENSEN, P.-É. DAGAND. *Coq: The World's Best Macro Assembler?*, in "PPDP", Madrid, Spain, ACM, 2013, pp. 13–24

[42] G. A. KILDALL. *A Unified Approach to Global Program Optimization*, in "POPL", 1973, pp. 194–206

[43] G. KLEIN, K. ELPHINSTONE, G. HEISER, J. ANDRONICK, D. COCK, P. DERRIN, D. ELKADUWE, K. ENGELHARDT, R. KOLANSKI, M. NORRISH, T. SEWELL, H. TUCH, S. WINWOOD. *seL4: formal verification of an OS kernel*, in "SOSP", 2009, pp. 207–220

[44] P. S. KOCHHAR, F. THUNG, D. LO, J. L. LAWALL. *An Empirical Study on the Adequacy of Testing in Open Source Projects*, in "Asia-Pacific Software Engineering Conference (APSEC)", Jeju, Korea, December 2014

[45] J. L. LAWALL, J. BRUNEL, N. PALIX, R. R. HANSEN, H. STUART, G. MULLER. *WYSIWIB: Exploiting fine-grained program structure in a scriptable API-usage protocol-finding process*, in "Software, Practice Experience", 2013, vol. 43, n^{o} 1, pp. 67–92

[46] J. L. LAWALL, B. LAURIE, R. R. HANSEN, N. PALIX, G. MULLER. *Finding Error Handling Bugs in OpenSSL using Coccinelle*, in "Proceeding of the 8th European Dependable Computing Conference (EDCC)", Valencia, Spain, April 2010, pp. 191–196

[47] J. L. LAWALL, D. LO. *An automated approach for finding variable-constant pairing bugs*, in "25th IEEE/ACM International Conference on Automated Software Engineering", Antwerp, Belgium, September 2010, pp. 103–112

[48] C. LE GOUES, W. WEIMER. *Specification Mining with Few False Positives*, in "TACAS", York, UK, Lecture Notes in Computer Science, March 2009, vol. 5505, pp. 292–306

[49] O. LEVILLAIN. *Parsifal: a Pragmatic Solution to the Binary Parsing Problem*, in "LangSec Workshop at IEEE Security & Privacy", May 2014

[50] Z. LI, S. LU, S. MYAGMAR, Y. ZHOU. *CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code*, in "OSDI", 2004, pp. 289–302

[51] Z. LI, 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

[52] D. LO, S. KHOO. *SMArTIC: towards building an accurate, robust and scalable specification miner*, in "FSE", 2006, pp. 265–275

[53] J.-P. LOZI, F. DAVID, G. THOMAS, J. L. LAWALL, G. MULLER. *Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications*, in "USENIX Annual Technical Conference", Boston, MA, USA, June 2012, pp. 65–76

[54] S. LU, S. PARK, Y. ZHOU. *Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing*, in "IEEE Transactions on Software Engineering", 2012, vol. 38, n^{o} 4, pp. 844–860

[55] S. MCCANNE, V. JACOBSON. *The BSD Packet Filter: A New Architecture for User-level Packet Capture*, in "USENIX Winter", 1993, pp. 259–269

[56] M. MERNIK, J. HEERING, A. M. SLOANE. *When and How to Develop Domain-specific Languages*, in "ACM Comput. Surv.", December 2005, vol. 37, n° 4, pp. 316–344, http://dx.doi.org/10.1145/1118890.1118892

[57] G. MORRISETT, G. TAN, J. TASSAROTTI, J.-B. TRISTAN, E. GAN. *RockSalt: better, faster, stronger SFI for the x86*, in "PLDI", 2012, pp. 395-404

[58] M. ODERSKY, T. ROMPF. *Unifying functional and object-oriented programming with Scala*, in "Commun. ACM", 2014, vol. 57, n° 4, pp. 76–86

[59] T. REPS, T. BALL, M. DAS, J. LARUS. *The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem*, in "ESEC/FSE", 1997, pp. 432–449

[60] C. RUBIO-GONZÁLEZ, H. S. GUNAWI, B. LIBLIT, R. H. ARPACI-DUSSEAU, A. C. ARPACI-DUSSEAU. *Error propagation analysis for file systems*, in "PLDI", Dublin, Ireland, ACM, June 2009, pp. 270–280

[61] L. RYZHYK, P. CHUBB, I. KUZ, E. LE SUEUR, G. HEISER. *Automatic device driver synthesis with Termite*, in "SOSP", 2009, pp. 73–86

[62] L. RYZHYK, A. WALKER, J. KEYS, A. LEGG, A. RAGHUNATH, M. STUMM, M. VIJ. *User-Guided Device Driver Synthesis*, in "OSDI", 2014, pp. 661–676

[63] R. SAHA, J. L. LAWALL, S. KHURSHID, 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, September 2014

[64] S. SAHA, J.-P. LOZI, G. THOMAS, J. L. LAWALL, G. MULLER. *Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software*, in "43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)", June 2013, pp. 1–12

[65] D. A. SCHMIDT. *Data Flow Analysis is Model Checking of Abstract Interpretations*, in "POPL", 1998, pp. 38–48

[66] P. SENNA TSCHUDIN, L. RÉVEILLÈRE, L. JIANG, D. LO, J. L. LAWALL, G. MULLER. *Understanding the Genetic Makeup of Linux Device Drivers*, in "PLOS", November 2013

[67] M. SHAPIRO. *Purpose-built languages*, in "Commun. ACM", 2009, vol. 52, n° 4, pp. 36–41

[68] R. TARTLER, D. LOHMANN, J. SINCERO, W. SCHRÖDER-PREIKSCHAT. *Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem*, in "EuroSys", 2011, pp. 47–60

[69] F. THUNG, D. LO, J. L. LAWALL. *Automated library recommendation*, in "20th Working Conference on Reverse Engineering (WCRE)", Koblenz, Germany, 2013, pp. 182–191

[70] F. THUNG, S. WANG, D. LO, J. L. LAWALL. *Automatic Recommendation of API Methods from Feature Requests*, in "28th IEEE/ACM International Conference on Automated Software Engineering", Palo Alto, CA, USA, November 2013

[71] Y. TIAN, J. L. LAWALL, D. LO. *Identifying Linux bug fixing patches*, in "ICSE", Zurich, Switzerland, June 2012, pp. 386–396

[72] Y. TIAN, D. LO, J. L. LAWALL. *Automated construction of a software-specific word similarity database*, in "CSMR-WCRE", Antwerp, Belgium, IEEE, February 2014, pp. 44–53

[73] Y. TIAN, D. LO, J. L. LAWALL. *SEWordSim: software-specific word similarity database*, in "ICSE Companion", Hyderabad, India, ACM, May 2014, pp. 568–571

[74] J.-B. TRISTAN, X. LEROY. *Formal verification of translation validators: a case study on instruction scheduling optimizations*, in "POPL", 2008, pp. 17–27

[75] W. WANG, M. GODFREY. *A Study of Cloning in the Linux SCSI Drivers*, in "Source Code Analysis and Manipulation (SCAM)", IEEE, 2011

[76] S. WANG, D. LO, J. L. LAWALL. *Compositional Vector Space Models for Improved Bug Localization*, in "International Conference on Software Maintenance and Evolution (ICSME)", Victoria, BC, Canada, September 2014

[77] H. S. WARREN. *Hacker's Delight*, Addison-Wesley Professional, 2012

[78] J. YANG, C. HAWBLITZEL. *Safe to the Last Instruction: Automated Verification of a Type-safe Operating System*, in "PLDI", 2010, pp. 99–110