IN PARTNERSHIP WITH:
CNRS

Université Pierre et Marie Curie (Paris 6)

Activity Report 2014

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

Team WHISPER

Keywords: Infrastructure Software, Operating System, Software Engineering, Safety, Proofs Of Programs

Creation of the Team: 2014 May 15.

1. Members

Research Scientists
Gilles Muller [Team leader, Inria, Senior Researcher, HdR]
Julia Lawall [Inria, Senior Researcher, HdR]
Pierre-Evariste Dagand [CNRS, from Oct. 2014]

Faculty Members
Bertil Folliot [Univ. Paris VI, Professor, HdR]
Gaël Thomas [Univ. Paris VI, Associate Professor, until Sep 2014, HdR]

Engineers
Xavier Clerc [Inria]
Quentin Lambert [Inria, from Nov 2014]

PhD Students
Brice Berna [ENS Cachan]
Antoine Blin [Cifre Renault]
Florian David [Univ. Paris VI]
Lisong Guo [Inria]
Valentin Rothberg [Inria, from Oct 2014]
Peter Senna Tschudin [Inria]

Administrative Assistant
Hélène 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 [50], [46], [76]. 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 5.1.1), 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 [39], [41], [42] or dynamic analysis [58], [60], [65]. Static analysis consists of approximating the behavior of the source code from the source code alone, while dynamic analysis draws conclusions from observations of sample executions, typically of test cases. While dynamic analysis can be more accurate, because it has access to information about actual program behavior, obtaining adequate test cases is difficult. This difficulty is compounded for infrastructure software, where many, often obscure, cases must be handled, and external effects such as timing can have a significant impact. Thus, we expect to primarily use static analyses. Static analyses come in a range of flavors, varying in the extent to which the analysis is *sound*, *i.e.*, the extent to which the results are guaranteed to reflect possible run-time behaviors.

One form of sound static analysis is *abstract interpretation* [41]. In abstract interpretation, atomic terms are interpreted as sound abstractions of their values, and operators are interpreted as functions that soundly manipulate these abstract values. The analysis is then performed by interpreting the program in a compositional manner using these abstracted values and operators. Alternatively, *dataflow analysis* [49] 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 [70]. More recently, more general forms of symbolic execution [39] have emerged as a means of understanding complex code. In symbolic execution, concrete values are used when available, and these are complemented by constraints that are inferred from terms for which only partial information is available. Reasoning about these constraints is then used to prune infeasible paths, and obtain more precise results. A number of works apply symbolic execution to operating systems code [36], [37].

While sound approaches are guaranteed to give correct results, they typically do not scale to the very diverse code bases that are prevalent in infrastructure software. An important insight of Engler et al. [44] 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 [40], in which a highly approximate analysis is complemented by a sound analysis that checks the individual reports of the approximate analysis, and then any errors in reasoning detected by the sound analysis are used to refine the approximate analysis. The CEGAR approach has been applied effectively on device driver code in tools developed at Microsoft [28]. The environment in which the driver executes, however, is still represented by possibly unsound approximations.

Going further in the direction of sacrificing soundness for scalability, the software engineering community has recently explored a number of approaches to code understanding based on techniques developed in the areas of natural language understanding, data mining, and information retrieval. These approaches view code, as well as other software-reated artifacts, such as documentation and postings on mailing lists, as bags of words structured in various ways. Statistical methods are then used to collect words or phrases that seem to be highly correlated, independently of the semantics of the program constructs that connect them. The obliviousness to program semantics can lead to many false positives (invalid conclusions) [54], 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 [53].

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] [62].

3.1.2.1. Traditional approach.

Using little languages to aid in software development is a tried-and-trusted technique [72] 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, [30] 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 [45] and its descendant Pharo [33] – or multi-paradigm languages – such as the Scala programming language [64] – also offer a wide range of mechanisms enabling the development of embedded DSLs. Perhaps suprisingly, a low-level imperative language – such as C – can also be extended so as to enable the development of embedded compilers [31].

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 [38], [34] in a unified framework enabling verification. Dependent types support the type-safe embedding of object languages and Coq's mixfix notation system enables reasonably idiomatic domain-specific concrete syntax. Coq's powerful abstraction facilities provide a flexible framework in which to not only implement and verify a range of domain-specific compilers [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, [35] of the theory of Kleene algebras.

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

3.2. 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 [34], [63], [38]. 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 [50], 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 [55] and Nail [29] 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 [63] 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 [61]. 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. [43], numerous research tools have been applied to the Linux kernel, typically for finding bugs [42], [57], [66], [73] or for computing software metrics [47], [75]. In our work, we have studied generic C bugs in Linux code , bugs in function protocol usage [51], [52], issues related to the processing of bug reports [21] and crash dumps [19], and the problem of backporting (work in progress), 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 5.1.2 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, [37] a template-based approach for porting device drivers from one OS to another. Ryzhyk *et al.* propose Termite, [67], [68] 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 [48] observe that

these approaches make assumptions that are not satisfied by many drivers; for example, the assumption that a driver involves little computation other than the direct interaction between the OS and the device. At the same time, a number of tools have been developed for finding bugs in driver code. These tools include SDV, [28] Coverity [43], CP-Miner, [56] PR-Miner [57], 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. New Software and Platforms

5.1. Platforms

5.1.1. 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.

To meet these needs, we have developed the Coccinelle program matching and transformation tool for C code. Coccinelle has been under development for over 7 years, and is mature software, available in a number of Linux distributions (Ubuntu, Debian, Fedora, etc.). Coccinelle 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*.

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 [51]. 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

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

[53], extracting critical sections into procedures to allow the implementation of a centralized locking service [59], generating a debugging interface for Linux driver developers [32], detecting resource release omission faults in Linux and other infrastructure software [69], and understanding the structure of device driver code in our current DrGene project [71].

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 2000 patches have been accepted into the Linux kernel based on the use of Coccinelle, including around 700 by around 90 developers from outside our research group. Over 40 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 a summer intern financed by the Linux Foundation, as part of the GNOME Foundation's Outreach Program for Women.

Finally, we are aware of several companies that use Coccinelle for modernizing code bases. These include Metaware in Paris, with whom we have 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.

5.1.2. 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 [32] [31] and Hector [69], 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 will develop 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. We now present the other tools considered in the BtrLinux action in more detail.

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 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,

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

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, [44], [57] but these approaches have very high rates of false negatives and false positives. [54] 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 will first return the code to working condition and then 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. New Results

6.1. Highlights of the Year

The paper "Faults in Linux 2.6" was published in the ACM journal Transactions on Computer Systems in June 2014 . It has been downloaded from the ACM digital library almost 300 times since then. The paper was reviewed in the Linux Weekly News, in the German professional IT website golem.de, and was the subject of an invited presentation at a joint session of the Linux Kernel Summit and LinuxCon North America.

Julia Lawall was invited to the 2014 Linux Kernel Summit, an invitation-only meeting of core Linux developers. She was subsequently invited to participate in the plenary Linux Kernel Developer Panel at LinuxCon Europe, with 2000 attendees.

Julia Lawall was invited to give a keynote at the conference Modularity (formerly AOSD) on her work on Coccinelle [17].

BEST PAPER AWARD :

[15] **Faults in Linux 2.6 in ACM Transactions on Computer Systems**. N. PALIX, G. THOMAS, S. SAHA, C. CALVÈS, G. MULLER, J. L. LAWALL.

6.2. Lock profiling in Java servers

Today, Java is regularly used to implement large multi-threaded server-class applications that use locks to protect access to shared data. However, understanding the impact of locks on the performance of a system is complex, and thus the use of locks can impede the progress of threads on configurations that were not anticipated by the developer, during specific phases of the execution. In our paper, "Continuously Measuring Critical Section Pressure with the Free-Lunch Profiler" [26], presented at OOPSLA 2014, we propose Free Lunch, a new lock profiler for Java application servers, specifically designed to identify, *in-vivo*, phases where the progress of the threads is impeded by a lock. Free Lunch is designed around a new metric, *critical section pressure* (CSP), which directly correlates the progress of the threads to each of the locks. Using Free Lunch, we have identified phases of high CSP, which were hidden with other lock profilers, in the distributed Cassandra

NoSQL database and in several applications from the DaCapo 9.12, the SPECjvm2008 and the SPECjbb2005 benchmark suites. Our evaluation of Free Lunch shows that its overhead is never greater than 6%, making it suitable for *in-vivo* use.

6.3. Software engineering for infrastructure software

A kernel oops is an error report that logs the status of the Linux kernel at the time of a crash. Such a report can provide valuable first-hand information for a Linux kernel maintainer to conduct postmortem debugging. Recently, a repository has been created that systematically collects kernel oopses from Linux users. However, debugging based on only the information in a kernel oops is difficult. In a paper published at MSR [19], we consider the initial problem of finding the offending line, i.e., the line of source code that incurs the crash. For this, we propose a novel algorithm based on approximate sequence matching, as used in bioinformatics, to automatically pinpoint the offending line based on information about nearby machine-code instructions, as found in a kernel oops. Our algorithm achieves 92% accuracy compared to 26% for the traditional approach of using only the oops instruction pointer.

2014 was the second year of a two-year cooperation between Julia Lawall and David Lo of Singapore Management University, as part of the Merlion cooperation grant program of the Insitut Français. This cooperation resulted in four papers: two on word similarity [22], [27], one on bug localization [24], and one on an empirical study of testing practices in open source software [20]. As an offshoot of this work, Julia Lawall worked with the PhD student Ripon Saha of UT Austin and his advisors on the topic of assessing the effectiveness of a state-of-the-art bug localization technique on C programs as compared to Java programs [21]. This work built on the C parser developed for Coccinelle.

Finally, with colleagues from Aalborg University and with Nicolas Palix of Grenoble, Julia Lawall published an article in Science of Computer Programming assessing the applicability of Coccinelle to checking the coding style guidelines of the CERT C Secure Coding Standard [14].

6.4. Bugs in Linux 2.6

In August 2011, Linux entered its third decade. Ten years before, Chou et al. published a study of faults found by applying a static analyzer to Linux versions 1.0 through 2.4.1. A major result of their work was that the drivers directory contained up to 7 times more of certain kinds of faults than other directories. This result inspired numerous efforts on improving the reliability of driver code. Today, Linux is used in a wider range of environments, provides a wider range of services, and has adopted a new development and release model. What has been the impact of these changes on code quality? To answer this question, in an article published in ACM TOCS, we have transported Chou et al.'s experiments to all versions of Linux 2.6; released between 2003 and 2011. We find that Linux has more than doubled in size during this period, but the number of faults per line of code has been decreasing. Moreover, the fault rate of drivers is now below that of other directories, such as arch. These results can guide further development and research efforts for the decade to come. To allow updating these results as Linux evolves, we define our experimental protocol and make our checkers available.

6.5. Memory Monitoring in Smart Home gateways

Smart Home market players aim to deploy component-based and service-oriented applications from untrusted third party providers on a single OSGi execution environment. This creates the risk of resource abuse by buggy and malicious applications, which raises the need for resource monitoring mechanisms. Existing resource monitoring solutions either are too intrusive or fail to identify the relevant resource consumer in numerous multi-tenant situations. In our paper "Memory Monitoring in a Multi-tenant OSGi Execution Environment" [16], presented at CBSE 2014, we propose a system to monitor the memory consumed by each tenant, while allowing them to continue communicating directly to render services. We propose a solution based on a list of configurable resource accounting rules between tenants, which is far less intrusive than existing OSGi monitoring systems. We modified an experimental Java Virtual Machine in order to provide the

memory monitoring features for the multi-tenant OSGi environment. Our evaluation of the memory monitoring mechanism on the DaCapo benchmarks shows an overhead below 46%. This work has been done as part of the PhD of Koutheir Attouchi [10] who was supported by a CIFRE grant with Orange Labs.

7. Bilateral Contracts and Grants with Industry

7.1. Bilateral Contracts with Industry

A 5-month contract with the company Metaware to provide support for Metaware's use of Coccinelle ended in February 2014. This contract resulted in numerous improvements in Coccinelle of interest to the general Coccinelle user community, including better handling of declarations involving multiple variables and better pretty printing of the gernated code.

The PhD of Koutheir Attouchi [10] on managing resources in the context of Smart Home gateway was supported by a CIFRE grant with Orange Labs.

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].

8. Partnerships and Cooperations

8.1. National Initiatives

8.1.1. 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.

Chronos network, Time and Events in Computer Science, Control Theory, Signal Processing, Computer Music, and Computational Neurosciences and Biology

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.

8.1.2. Multicore Inria Project Lab

The Multicore IPL is an Inria initiative led by Gilles Muller, whose goal is to develop techniques for being able to deploy 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, we are working with Jens Gustedt 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 allow them to be deployed on multicore machines with varying architectures.

8.2. European Initiatives

8.2.1. Collaborations in European Programs, except FP7 & H2020

Program: COST Action IC1001
Project acronym: Euro-TM
Project title: Transactional Memories: Foundations, Algorithms, Tools, and Applications
Duration: 2011 - 2014
Coordinator: Dr. Paolo Romano (INESC)
Whisper member: Gilles Muller, leader of the working group on Hardware's & Operating System's Supports
Other partners: Austria, Czech Republic, Denmark, France, Germany, Greece, Israel, Italy, Norway, Poland, Portugal, Serbia, Spain, Sweden, Switzerland, Turkey, United Kingdom.
Abstract: Parallel programming (PP) used to be an area once confined to a few niches, such as scientific and high-performance computing applications. However, with the proliferation of multicore processors, and the emergence of new, inherently parallel and distributed deployment platforms, such as those provided by cloud computing, parallel programming has definitely become a mainstream concern. Transactional Memories(TMs) answer the need to find a better programming model for PP, capable of boosting developer's productivity and allowing ordinary programmers to unleash the power of parallel and distributed architectures avoiding the pitfalls of manual, lock based synchronization. It is therefore no surprise that TM has been subject to intense research in the last years. This Action aims at consolidating European research on this important field, by coordinating the European research groups working on the development of complementary, interdisciplinary aspects of Transactional Memories, including theoretical foundations, algorithms, hardware and operating system support, language integration and development tools, and applications.

8.3. International Initiatives

8.3.1. Participation In other International Programs

Julia Lawall obtained the renewal of a Merlion collaboration grant, started in 2013, for collaboration with David Lo of Singapore Management University. This collaboration resulted in a two-week visit of Julia Lawall to Singapore Management University, a one-week visit of David Lo to the Whisper team, and a two-week visit of Lo's PhD student Ferdian Thung to the Whisper team. It also resulted in four publications during 2014 [27], [22], [24], [20].

8.4. International Research Visitors

8.4.1. Visits of International Scientists

8.4.1.1. Internships

Julia Lawall supervised the remote internships of Himangi Saraogi (summer 2014) and Tapasweni Pathak (winter 2014, in progress) as part of the Gnome Outreach Program for Women (OPW). Both interns carried out projects related to Coccinelle and the Linux kernel. Julia Lawall has taken over the responsability for the coordination of the Linux kernel's participation in the OPW program in winter 2014.

Julia Lawall also supervised the internship of the undergraduate student (L2) Chi Pham from the University of Copenhagen. Pham developed a tool for transforming Coccinelle semantic patches to make them suitable for inclusion in the Linux kernel.

9. Dissemination

9.1. Promoting Scientific Activities

9.1.1. Scientific events organisation

9.1.1.1. Member of the organizing committee

Julia Lawall was a member of the SIGPLAN Executive Committee, which supervises the organization of the various conferences sponsored by SIGPLAN, as well as other SIGPLAN activities.

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

9.1.2. Scientific events selection

9.1.2.1. Member of the conference program committee

Gilles Muller was a PC member for the conferences VEE, TRIOS, for the workshop APSYS and for the jury of the best EuroSys PhD (Roger Needham award).

Julia Lawall was a PC member for the conferences Modularity and ICDCS.

Pierre-Évariste Dagand was a PC member for WGP 2014, the workshop on generic programming.

9.1.3. Journal

9.1.3.1. Member of the editorial board

Julia Lawall: Higher-Order and Symbolic Computation, Science of Computer Programming.

9.2. Teaching - Supervision - Juries

9.2.1. Teaching

9.2.2. Supervision

PhD : Koutheir Attouchi, Managing Resource Sharing Conflicts in an Open Embedded Software Environment, Université Pierre et Marie Curie, 11 juillet 2014, Gilles Muller et Gaël Thomas, CIFRE Orange

PhD : Jean-Pierre Lozi, Towards more scalable mutual exclusion for multicore architectures, Université Pierre et Marie Curie, 16 juillet 2014, Gilles Muller et Gaël Thomas

PhD : Lisong Guo, Boost the Reliability of the Linux Kernel: Debugging Kernel Oopses, Université Pierre et Marie Curie, 18 décembre 2014, Julia Lawall et Gilles Muller

PhD in progress : Florian David, A profiler for locks in Java servers, octobre 2011, Gilles Muller et Gaël Thomas

PhD in progress : Peter Senna Tschudin, Développement Rapide de Pilotes de Périphériques, mai 2014, Julia Lawall et Gilles Muller

PhD in progress : Valentin Rothberg, Exploration de la génétique des pilotes périphériques, octobre 2014, Julia Lawall et Gilles Muller

PhD in progress : Antoine Blin, Execution of real-time applications on a small multicore embedded system, avril 2012, Gilles Muller et Julien Sopena (Regal), CIFRE Renault

9.2.3. Juries

- Gilles Muller:
  - HDR: David Bromberg (University of Bordeaux, reviewer), Michaël Hauspie (reviewer).
  - PhD: Jigar Solanki (University of Bordeaux, rapporteur), Pierre-Louis Aublin (University of Grenoble, reviewer), Baptiste Lepers (University of Grenoble), Etienne Millon (University of Pierre et Marie Curie, President)
- Julia Lawall:
  - HDR: David Bromberg (University of Bordeaux, reviewer), Nicolas Anquetil (University of Lille).
  - PhD: Alexandre Lissy (University of Tours, reviewer), Lucia (Singapore Management University)

9.3. Popularization

Julia Lawall presented a tutorial on Coccinelle at ENS (Masters students), the IT University of Copenhagen (PhD students), at the Suse Labs conference (developers), at LinuxCon Europe (developers), and at Middleware (PhD students and researchers).

10. 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^{o} 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: Ten Years Later*, in "ASPLOS", Newport Beach, CA, USA, ACM, March 2011, pp. 305–318

Publications of the year

Doctoral Dissertations and Habilitation Theses

[10] K. ATTOUCHI. *Managing Resource Sharing Conflicts in an Open Embedded Software Environment*, Université Pierre et Marie Curie, July 2014, https://hal.archives-ouvertes.fr/tel-01088028

[11] L. GUO. *Boost the Reliability of the Linux Kernel: Debugging Kernel Oopses*, UPMC, Paris Sorbonne, December 2014, https://hal.inria.fr/tel-01096662

[12] J.-P. LOZI. *Towards more scalable mutual exclusion for multicore architectures*, Université Pierre et Marie Curie - Paris VI, July 2014, https://tel.archives-ouvertes.fr/tel-01067244

Articles in International Peer-Reviewed Journals

[13] T. F. BISSYANDÉ, L. RÉVEILLÈRE, J. LAWALL, G. MULLER. *Ahead of Time Static Analysis for Automatic Generation of Debugging Interfaces to the Linux Kernel*, in "Automated Software Engineering", May 2014, pp. 1-39 [*DOI :* 10.1007/s10515-014-0152-4], https://hal.archives-ouvertes.fr/hal-00992283

[14] M. C. OLESEN, R. R. HANSEN, J. L. LAWALL, N. PALIX. *Coccinelle: Tool support for automated CERT C Secure Coding Standard certification*, in "Science of Computer Programming", October 2014, vol. 91, n^{o} B, pp. 141–160, https://hal.inria.fr/hal-01096185

[15] *Best Paper*
N. PALIX, G. THOMAS, S. SAHA, C. CALVÈS, G. MULLER, J. L. LAWALL. *Faults in Linux 2.6*, in "ACM Transactions on Computer Systems", June 2014, vol. 32, n^{o} 2, pp. 1–40 [*DOI :* 10.1145/2619090], https://hal.archives-ouvertes.fr/hal-01022704.

Invited Conferences

[16] K. ATTOUCHI, G. THOMAS, A. BOTTARO, G. MULLER. *Memory Monitoring in a Multi-tenant OSGi Execution Environment*, in "CBSE '14 -17th international ACM Sigsoft symposium on Component-based software engineering", Marcq-en-Baroeul, France, ACM, June 2014 [*DOI :* 10.1145/2602458.2602467], https://hal.archives-ouvertes.fr/hal-01080634

[17] J. L. LAWALL. *Coccinelle: reducing the barriers to modularization in a large C code base*, in "MODULARITY - 13th International Conference on Modularity", Lugano, Switzerland, W. BINDER, E. ERNST, A. PETERNIER, R. HIRSCHFELD (editors), ACM, April 2014, pp. 5-6 [*DOI :* 10.1145/2584469.2584661], https://hal.inria.fr/hal-01001894

International Conferences with Proceedings

[18] F. DAVID, G. THOMAS, J. LAWALL, G. MULLER. *Continuously Measuring Critical Section Pressure with the Free-Lunch Profiler*, in "OOPSLA 2014", Portland, Oregon, United States, ACM, October 2014 [*DOI :* 10.1145/2660193.2660210], https://hal.inria.fr/hal-01080277

[19] L. GUO, J. LAWALL, G. MULLER. *Oops! Where did that code snippet come from?*, in "11th Working Conference on Mining Software Repositories", Hyderabad, India, P. T. DEVANBU, S. KIM, M. PINZGER (editors), ACM, May 2014, pp. 52-61 [*DOI :* 10.1145/2597073.2597094], https://hal.inria.fr/hal-01001878

[20] K. PAVNEET SINGH, F. THUNG, D. LO, J. LAWALL. *An Empirical Study on the Adequacy of Testing in Open Source Projects*, in "21st Asia-Pacific Software Engineering Conference", Jeju, South Korea, December 2014, https://hal.inria.fr/hal-01096132

[21] R. K. SAHA, J. L. LAWALL, S. KHURSHID, 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", Victoria, Canada, IEEE, September 2014, pp. 161-170 [*DOI :* 10.1109/ICSME.2014.38], https://hal.inria.fr/hal-01086082

[22] Y. TIAN, D. LO, J. LAWALL. *Automated construction of a software-specific word similarity database*, in "2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE", Antwerp, Belgium, IEEE, February 2014, pp. 44-53, https://hal.inria.fr/hal-01086077

[23] Y. TIAN, D. LO, J. LAWALL. *SEWordSim: software-specific word similarity database*, in "ICSE'14 - 36th International Conference on Software Engineering, Companion Proceedings", Hyderabad, India, P. JALOTE, L. C. BRIAND, A. VAN DER HOEK (editors), ACM/IEEE, June 2014, pp. 568-571 [*DOI :* 10.1145/2591062.2591071], https://hal.inria.fr/hal-01001892

[24] S. WANG, D. LO, J. LAWALL. *Compositional Vector Space Models for Improved Bug Localization*, in "30th International Conference on Software Maintenance and Evolution", Victoria, Canada, IEEE, September 2014, pp. 171-180, https://hal.inria.fr/hal-01086084

Research Reports

[25] K. ATTOUCHI, G. THOMAS, A. BOTTARO, J. L. LAWALL, G. MULLER. *Incinerator - Eliminating Stale References in Dynamic OSGi Applications*, Inria, February 2014, n^{o} RR-8485, 22 p. , https://hal.inria.fr/hal-00952327

[26] F. DAVID, G. THOMAS, J. LAWALL, G. MULLER. *Continuously Measuring Critical Section Pressure with the Free Lunch Profiler*, Inria Whisper, March 2014, n^{o} RR-8486, 24 p. , https://hal.inria.fr/hal-00957154

Other Publications

[27] Y. TIAN, D. LO, J. LAWALL. *SEWordSim: software-specific word similarity database*, ACM, May 2014, pp. 568-571, ICSE Companion 2014 - Companion Proceedings of the 36th International Conference on Software Engineering [*DOI :* 10.1145/2591062.2591071], https://hal.inria.fr/hal-01086079

References in notes

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

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

[34] 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

[35] 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

[36] 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

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

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

[39] 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

[40] 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

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

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

[43] 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

[44] 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

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

[46] 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

[47] 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

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

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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

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

[56] 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

[57] 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

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

[59] 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

[60] 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

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

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

[63] 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

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

[65] 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

[66] 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

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

[68] 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

[69] 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

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

[71] 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

[72] M. SHAPIRO. *Purpose-built languages*, in "Commun. ACM", 2009, vol. 52, n^{o} 4, pp. 36–41

[73] 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

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