L4Cars

Kevin Elphinstone Gernot Heiser Ralf Huuck
Stefan M. Petters Sergio Ruocco

National ICT Australia*
and
University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract

Automotive components present unique challenges in reliability, security, performance and cost. Consolidation of different functions in multi-purpose units drives up complexity, and raises not only reliability concerns, but also the issue of liability for sub-component suppliers. It is of foremost importance to guarantee reliability and security right from the start when designing such systems. In this work we present a complete approach grounded in a flexible and secure microkernel, supported by a flexible operating system and component architecture on top. This is coupled with rigorous software development to assure the reliability and security.

1 Introduction

Security is becoming an increasingly important issue for cars and their complex software development process. While a decade ago security was mainly an issue of mechanical locks and anti-theft-units, it has largely grown into a design and software problem.

Cars are therefore subject to new threats from the outside, e.g., unauthorised access by cracking electronic locks, accessing and changing control software, or simple unauthorised "updates". There are also conceptual design threats posed by malicious or rogue software running in a system. How can we ensure that separated functionalities implemented on the same control unit remain separated? What happens if parts of the software fail?

Those threats are real and they have a direct impact — measurable in recalls, down-time, damaged image, or litigation. As such, security issues have a wide impact on operational safety and overall production cost.

We advocate a sound development process from the hardware up, focussed on security. This requires a small operating system kernel which is fast and highly reliable. Moreover, it has to provide strong separation not only for security issues, but also to guarantee a separation of concerns preventing the failure of a complete electronic control unit (ECU) or even the whole system if one functionality fails critically. Additionally, the trend toward ECU consolidation further increases system complexity, and is compelling motivation for moving away from the monolithic system image approach to embedded system development.

In this paper we present an approach which satisfies all of the above requirements. Figure 1 illustrates the general structure of the system design we advocate. We assume (and argue for) ECUs with hardware-based memory protection and a distinction between privileged mode of execution (for the operating-system kernel) and user-mode execution (for everything else). We believe the move to such ECUs is inevitable. Given memory protection, we argue the system's software should be based on a small, stable, flexible, high-performance microkernel that enables a wide class of systems to be constructed. Such a microkernel must be trustworthy and hence must be the result of a rigorous assurance process.

On top of the base microkernel we expect a su-

*National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

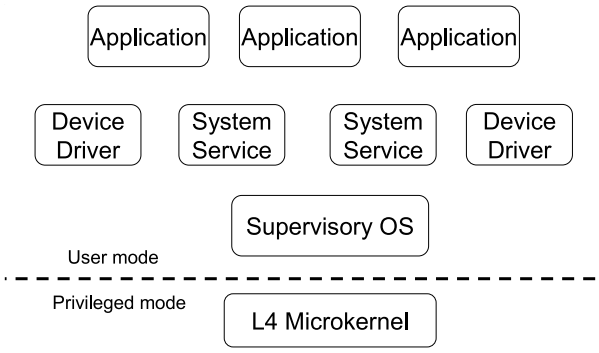

Figure 1: L4-based embedded system

pervisory operating system server that implements domain-specific management of the overall system. For example, it would be responsible for implementing and enforcing separation on the system using the microkernel's basic mechanisms. Device drivers and other system services run as unprivileged (user-mode) servers in their own address spaces, and form a higher level environment for domain-specific applications, also running at user-mode in their own address space.

We believe our approach is conducive to the construction of secure embedded systems as it provides a small, secure, highly assured microkernel as the fundamental building block, i.e., the *trusted computing base* (TCB). Upon this building block, systems components of varying degrees of security and assurance can be instantiated with strong separation guarantees from other components.

For the remainder of the paper, we provide some background on the L4 microkernel (Section 2), which describes the basic mechanisms we believe should be provided by the base of a secure embedded-system kernel. Section 3 describes several projects underway to further improve L4 as a secure embedded systems platform, and Section 4 presents our conclusions.

2 L4 Background

A secure system must be built upon secure foundations. The basic mechanisms of a microkernel-based system foundation are paramount to achieving the goals of security. Insecure mechanisms are an obvious problem, but inflexible or inefficient mechanisms are problematic too. Inflexibility or inefficiencies result in kernel optimisations, additional features, and work-arounds which potentially undermine an original secure design and any original assurance of correctness.

L4 has been used as the basis of a wide variety of systems with few modifications to its basic mechanisms. Besides its early development for high performance systems, work on versions targeted for embedded systems has progressed to a state of commercial deployment. This covers in particular the recent release of the embedded systems API. In this section, we provide an outline of L4's mechanisms and advocate their use to enable the construction of trustworthy embedded systems.

2.1 Threads & IPC

L4 provides kernel-scheduled threads as the model of execution. These are scheduled preemptively using a strict priority-based round-robin scheduler. The priority and time quantum can be assigned per thread. By varying these parameters, several standard schedulers can be emulated — examples being rate-monotonic (by combining distinct priorities with infinite time slices), or proportional share (by combining a single priority with time slices of different lengths to represent the thread's share of the CPU).

Threads interact via interprocess communication (IPC). IPC takes the form of synchronous messages that are efficiently transferred between threads at very near the hardware-dictated context switching cost [1].

2.2 Address Spaces

The microkernel provides mechanisms for creating, managing, and destroying address spaces. These mechanisms provide a level of abstraction above the raw memory management hardware. Address spaces allow the kernel to enforce separation (sometimes termed *partitioning*) between applications, and thus control interaction between applications and enable fault isolation.

Address-space manipulation is via the *map*, *grant*, and *unmap* model as illustrated in Figure 2. The figure shows rectangular boxes representing address spaces. σ_0 initially possesses all non-kernel physical memory; A is an operating system server; C and D are two clients of A. L4 implements a recursive virtual address space model which permits virtual memory management to be performed entirely at user level. It is recursive in the sense that each address space is defined in terms of other address spaces. Initially all physical memory is mapped within the root address space σ_0,

whose role is to make that physical memory available to new address spaces (in this case, the operating system server A and another server B, which might be a legacy OS for running user interfaces). A's address space is constructed by mapping regions of accessible virtual memory from σ_0's address space to the next, such that rights are either preserved or reduced.

Page faults are handled by the kernel transforming them into messages delivered via IPC. Every thread has a pager thread associated with it. The pager is responsible for managing a thread's address space. Whenever a thread takes a page fault, the kernel catches the fault, blocks the thread and synthesises a page-fault IPC message to the pager on the thread's behalf. The pager can then respond with a mapping and thus unblock the thread. Thus the pager can implement zero-fill on access, demand loading, shared memory, or enforce or relax partitioning.

This model has been successfully used to construct several very different systems as user-mode applications, including real-time systems and single-address-space systems [2–5].

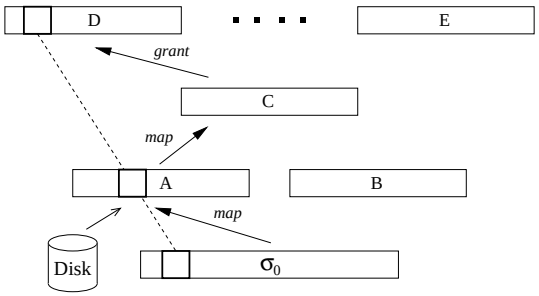

Figure 2: Virtual Memory Primitives

2.3 Device Drivers

Device drivers on an L4-based system run as applications in user-mode. The kernel itself only provides support for interrupt delivery and access to device registers and memory. Interrupts are delivered as messages from virtual *interrupt threads*, which uniquely identify the interrupt source. The real interrupt service routine within the kernel masks the interrupt, transforms the interrupt event into an IPC message from the virtual interrupt thread, which is delivered to the driver's interrupt service routine (itself a thread). The routine within the driver receives the message and performs the normal service-routine functionality. Upon completion, the driver sends a reply message to the virtual interrupt thread, resulting in the interrupt source being unmasked. Access to device registers is achieved using the normal address-space management mechanisms. Registers are mapped into the virtual address space such that driver applications can access them directly.

Device drivers running at user-mode do suffer a small performance penalty, however our experiments show that for a general purpose system, the penalty for running a Gigabit Ethernet driver at user-mode was a small increase in CPU utilisation (less than 10%) and approximately equal throughput [6]. Less performance-critical drivers have negligible impact on performance. The advantages of running device drivers outside the kernel is that the kernel itself remains stable and independent of the hardware platform, the system as a whole is more robust to driver faults, and importantly, leaving drivers outside the kernel provides a significant reduction in kernel verification complexity.

3 Pieces of the Puzzle

The core building blocks that support this vision exist already, such as the microkernel and the supervisory operating system (called *Iguana*). These are mature enough to be deployed in commercial embedded systems. However, more work is required in order to deliver on the promise of secure and reliable embedded systems. We are consequently working on a number of projects which aim at providing the missing building blocks over the next two years. This is schematically shown in Figure 3, and explained in this section.

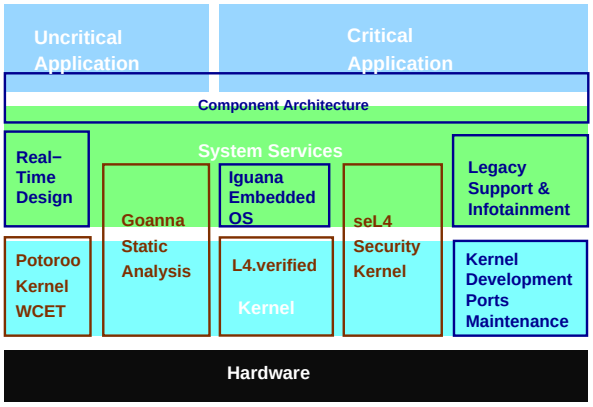

Figure 3: Projects creating a secure platform

3.1 Software architecture

The microkernel approach derives much of its power from the ability to support highly componentised systems *with hardware-enforced encapsulation* of components. The microkernel provides the mechanisms to achieve this, but a support framework is required to make it feasible to develop (potentially very large) software systems in a componentised fashion. One project, dubbed *CAmkES*, aims at developing such a software component architecture, specifically for next-generation embedded systems. It introduces mechanisms that enable the separation of concerns, while supporting the performance, security, and reliability needs of embedded system software.

Organising software as interacting components with well-defined interfaces improves the reusability of code. It also improves flexibility by allowing components to be added and removed from a system (possibly at run-time), as well as allowing components developed in different languages to interact with each other. Components provide natural units of protection, fault containment, and resource management. Finally, componentisation with hardware-enforced interfaces also reduces the complexity of assurance.

3.2 Secure Foundation

L4 has proved itself to be a flexible platform for system development in general. However, there are some shortcomings with the current design [7, 8]. These shortcomings can be summarised as inflexible or inefficient control of communication, and insufficient control of kernel memory management. The *secure embedded L4* (seL4) project aims to address these and other issues so as to ensure L4 is a suitable platform for secure embedded systems development.

Communication control is the key to controlling interaction between applications. The only direct interaction between L4 applications is via IPC, or is established via IPC (e.g. a shared-memory mapping established via IPC). Secure communication control enables both integrity (an unforgeable identity in messages) and confidentiality (control of information flow) guarantees to be enforced by the microkernel. It has been observed that system partitioning can be viewed as a security policy on information flow [9]. If partitions are unable to detect the presence of another partition, then faults in other partitions can have no effect.

The seL4 project is exploring the use of capability-based communication to provide L4 with a basic, flexible, efficient mechanism for enforcing communication policy. Such a mechanism would be policy neutral, i.e. could be customisable by the specific system to match the application domain. Systems could have no communication restrictions, or be strictly partitioned.

Kernel memory resources require controlled and predictable management. A system needs to be able to precisely control the memory used by the kernel for implementing kernel abstractions. Precise control ensures that memory cannot be exhausted by malicious or malfunctioning applications. Predictable kernel-memory allocation also improves the predictability of the kernel's temporal behaviour, in contrast to secure systems that use kernel memory as a cache of kernel objects [10].

While L4 is a small kernel by general-purpose OS standards (roughly 10,000 lines of code), it is still a moderate-sized kernel by embedded standards. The secure embedded L4 project aims to further reduce the kernel size by simplifying some kernel abstractions where features of the kernel are not generally used by embedded applications. Additionally, we are working toward a single-stack version of the kernel to reduce the kernel's dynamic memory footprint.

3.3 Assurance

The idea of a microkernel is to establish a small trusted computing base which is secure and reliable. Therefore, it is imperative that this computing base is in fact functionally correct as all other system and application services will depend on it. We are convinced that the highest possible assurance has to be given and that these guarantees have to be established rigorously.

Moreover, for all components within or outside the TCB, we advocate the use of additional automatic software analysis tools to detect standard bugs during the implementation process. The advantages are lower production costs by early detection and, of course, higher assurance right from the start of the implementation process.

It is virtually impossible to guarantee correctness of a system, and in turn the absence of bugs, by standard software engineering practice such as

code review, systematic testing and good software design alone. The complexity of system software is typically too high to be manageable by informal reasoning or reasoning that is not tool-supported. The formal methods community has developed various rigorous, mathematically sound techniques and tools that have matured enough within the last decades to allow the formal analysis of system software.

Three projects are working on various aspects of assurance. The *L4.verified* project is concerned with maximal assurance for the microkernel itself, using mathematical proofs. The *Goanna* project focusses on less strict, yet more generally and quickly applicable techniques for increasing the trustworthiness of systems code. The *Potoroo* project is working towards a model of the timing behaviour of the L4 microkernel, in order to support the real-time analysis of L4-based software.

The main ideas of the projects are outlined below. For a more comprehensive introduction see also [11].

3.3.1 L4.verified

The highest level of assurance possible is a mathematical proof that a system satisfies the desired properties. Given a suitably small TCB, it is possible to construct such a proof, machine checked, and with the help of an interactive proof assistant. For any nontrivial system, the kernel, albeit small compared to the full system size, will still be a very large construct to verify formally. We therefore follow an approach where one first (manually) builds a small, abstract specification of the system, then (machine-supported) proves the desired properties about the specification, and finally (again machine-supported) proves that the implementation of the system is a formal refinement of the specification [12,13]. This approach is not applicable to all kinds of properties, but to all that are preserved under the refinement process, including the functional correctness of the system. This approach requires a significant amount of resources, as well as expertise in formal verification and the problem domain. It can guarantee that the kernel behaves exactly as specified.

We use higher-order logic and the theorem prover Isabelle as our tool set to describe the behaviour of the kernel at an abstract level in the form of an operational specification. This description is then *refined* inside the prover into a program written in a standard, imperative, C-like language. The last step consists of showing that the actual C code is indeed a refinement of the C-like program. This is done by automatically parsing the C code into an Isabelle model and then establishing the refinement relation formally.

The abstract description is at the level of a reference manual and relatively easy to understand. At that level, address translation for instance is modelled as an abstract function from virtual to physical addresses, explicitly not mentioning that this lookup is implemented by complex page table data structures in the real system. This is the level we use for analysing the behaviour of the system and for proving additional simple safety properties, such as the requirement that the same virtual address can never be translated to two different physical addresses. At the end of the refinement process stands a formally verified imperative program — the kernel implementation in full detail.

Proof-based OS verification has been tried in the past [14,15]. The rudimentary tools available at the time meant that the proofs had to end at the design level; full implementation verification was not feasible. The verification of Kit [16] down to object code demonstrated the feasibility of this approach to kernel verification, although on a system that is far simpler than any real-life OS kernel in use in secure systems today.

Since the early attempts at kernel verification there have been dramatic improvements in the power of available theorem proving tools. Proof assistants like ACL2, Coq, PVS, HOL and Isabelle have been used in a number of successful verifications, ranging from mathematics and logics to microprocessors [17], compilers [18], and full programming platforms like JavaCard [19].

We are currently among several groups engaged in microkernel verification, notably VFiasco [20], VeriSoft [21] and Coyotos [22].

3.3.2 Goanna

While full functional correctness can be shown only by proof-based methods, the effort is high, i.e., there is substantial expert manpower needed over an extended period of time. This is not always practical. In particular for less sensitive system parts, a lower-level assurance can be accepted if this can be achieved in a short period of time. Furthermore, any small change to the implementation of a formally-verified kernel (leave alone a port

to a different hardware platform) requires proofs to be redone, which is expensive and may not be justified for applications which are not mission critical.

The Goanna project aims at developing a software-analysis tool that works automatically and delivers results in minutes, ideally even faster. This will be integrated in the software development process such that it becomes as natural to use as the compiler itself. Since it does not require any user interaction it will also be applicable by non-formal methods experts.

In contrast to the heavyweight theorem proving approach in the L4.verified project, the Goanna approach considers the application and development of rather lightweight *static analysis* techniques [23, 24]. Static analysis is a general term comprising a number of analysis techniques which can be applied at compile-time, i.e., prior to the execution of the actual code. In fact, some of these techniques can be applied in even earlier design stages when the code does not yet compile and, therefore, is not executable.

The drawback of any automatic software analysis technique is that almost all properties are generally undecidable, as the problem can be reduced to the halting problem. To make them nonetheless usable in practice, decidable approximations are computed. These can be either *over-* or *under-approximations.*

With regard to safety properties, over-approximations consider abstract programs which exhibit more behaviour than the actual concrete program. If an abstract program still satisfies a given safety property although it exhibits more behaviour then the concrete program will do so as well (since it has less behaviour that can violate that property). However, if the abstract program does violate the given property it does not necessarily mean that the concrete program does violate it as well, since the violation might just be in the over-approximated part of the program behaviour which is not in the concrete program. In this case we have a *false alarm* or *false positive.* It is a major research challenge in the area of automatic software analysis to minimise the number of false alarms.

Under-approximations on the other hand consider program approximations which exhibit less behaviour than the original program. In this case any violation of a safety property in the approximated program is certainly a violation in the original program while the absence of a violation does not guarantee that the program has no harmful (i.e., property violating) behaviour. Again, it is a major research challenge to keep the gap between the actual behaviour and the approximated one as small as possible.

Under-approximations are well suited to exploit bugs in programs while over-approximations are used to establish correctness. We seek to attack the problem from both sides to gain maximum results.

In a first step we only consider a fixed set of properties and violations such as buffer-overruns, division by zero, unreachable code, uninitialised variables etc. We tailor our analysis techniques specifically to the properties to be established, using different techniques as appropriate (such as data flow analysis, abstract interpretation, and, to some extend, model checking).

Similar approaches are used, e.g., by Microsoft's Static Driver Verifier [25] and the Coverity toolset [26].

3.3.3 Potoroo

Automotive systems are real-time by nature, and hence functional correctness needs to be complemented by temporal correctness. This is motivated by the fact that a number of ECUs in cars have time-critical functionality and security requirements. An example of this are driver-assistance systems intervening with steering and breaking, which require a secure logging mechanism for litigation prevention. This is particular relevant in the context of ECU consolidation in which software from several suppliers is executed on the same high-performance ECUs. Especially these ECUs which are equipped with caches are hard to analyse and the widely-used end-to-end measurements with safety factors have led to increasing number of problems in the industry, as the safety factors do not necessarily cover worst-case situations.

The analysis of the real-time behaviour of systems relies on knowing the worst-case execution time (WCET) of all software components. While application-level code has been targeted in the past, little (if any) work has been done on establishing credible timing models of the operating-system kernel. The Potoroo project specifically targets the temporal behaviour of the kernel. The work is based on the measurement-based approach [27] which is now commercialised by Rapita Systems

Ltd. [28]. It measures the code on basic block level and creates a reliable and tight estimate of the WCET of the overall program using a syntax tree representation. Besides the worst-case execution time, the method is able to provide insights into the probability with which this worst case may actually appear. The method is adapted within the Potoroo project to work on kernel code.

3.4 Infotainment

Despite some lag in the uptake of entertainment devices in the car, we expect similar trends as in other industries, like the telecommunications sector. There we can observe a move from a simple single-function device (for making voice calls) to a extensible multi-functional device that includes productivity applications as well as multimedia features for entertainment. This increases the level of security threats, as systems are expected to deal with downloaded software, such as games or non-certified *upgrades*. It also leads to a desire for support of legacy APIs, such as Linux or Windows. Such APIs generally do not satisfy the requirements of secure embedded systems, and legacy code must be forced to execute in a carefully contained environment.

Our *Wombat* server [29] is a slimline deprivileged Linux server that is integrated into the L4-based environment. It allows user-level applications to run without imposing security risks on the rest of the system. Similar user-level environments can be created to support other APIs.

4 Conclusions

In this work we presented a flexible and complete approach for high performance and secure embedded systems. Moreover, we stressed the need to integrate rigorous safety and security analysis right from the start. In fact, we believe that only verified software will be successful in the long run.

The L4 microkernel has reached production status now and is distributed as open source under a BSD-style license. Commercial products based on the framework are expected to hit the stores early next year.

Besides the projects listed above, we are exploring novel architectural designs, such as a flexible component structure, system-wide approaches to power management and the use of reflective schedulers for improved control and improved adaptability of temporal behaviour. Also under investigation are hardware- and resource-related security aspects such as covert channels.

References

[1] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger, "Achieved IPC performance (still the foundation for extensibility)," in *6th HotOS*, (Cape Cod, MA, USA), pp. 28–31, May 1997.

[2] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefi t of separate address spaces in real-time operating systems," in *23rd RTSS*, (Austin, TX, USA), 2002.

[3] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke, "The Mungi single-address-space operating system," *Softw.: Pract. & Exp.*, vol. 28, pp. 901–928, Jul 1998.

[4] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *16th SOSP*, (St. Malo, France), pp. 66–77, Oct 1997.

[5] A. Geffhut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther, "The Sawmill multiserver approach," in *9th SIGOPS Eur. WS*, (Kolding, Denmark), pp. 109–114, ACM Press, 2000.

[6] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser, "User-level device drivers: Achieved performance," *J. Comput. Sci. & Technol.*, vol. 20, Sep 2005.

[7] L4Ka Team, "L4Ka::Pistachio kernel." http://l4ka.org/projects/pistachio/.

[8] K. Elphinstone, "Future directions in the evolution of the L4 microkernel," in *Proceedings of the NICTA workshop on OS verifi cation 2004, Technical Report 0401005T-1* (G. Klein, ed.), (Sydney, Australia), National ICT Australia, Oct 2004.

[9] J. Rushby, "Partitioning for safety and security: Requirements, mechanisms, and assurance," NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[10] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A fast capability system," in *17th SOSP*, (Charleston, SC, USA), pp. 170–185, Dec 1999.

[11] G. Klein and R. Huuck, "High assurance system software," in *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software*, (Sydney, Australia), To appear., 2005.

[12] G. Klein and H. Tuch, "Towards verified virtual memory in L4," in *TPHOLs Emerging Trends '04* (K. Slind, ed.), (Park City, Utah, USA), 2004.

[13] H. Tuch and G. Klein, "Verifying the L4 virtual memory subsystem," in *Proc. NICTA FM Workshop on OS Verification* (G. Klein, ed.), pp. 73–97, Technical Report 0401005T-1, National ICT Australia, 2004.

[14] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, "A provably secure operating system: The system, its applications, and proofs," Tech. Rep. CSL-116, SRI International, 1980.

[15] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA Unix security kernel," *Communications of the ACM*, vol. 23, no. 2, pp. 118–131, 1980.

[16] W. R. Bevier, "Kit: A study in operating system verification," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1382–1396, 1989.

[17] B. C. Brock, W. A. Hunt, Jr., and M. Kaufmann, "The FM9001 microprocessor proof," Tech. Rep. 86, Computational Logic, Inc., 1994.

[18] S. Berghofer and M. Strecker, "Extracting a formally verified, fully executable compiler from a proof assistant," in *Proc. COCV'03*, Electronic Notes in Theoretical Computer Science, pp. 33–50, 2003.

[19] "VeriCard project." http://verificard.org, 2005.

[20] M. Hohmuth, H. Tews, and S. G. Stephens, "Applying source-code verification to a microkernel —the VFiasco project," Tech. Rep. TUD-FI02-03-März, TU Dresden, 2002.

[21] "VeriSoft project." http://www.verisoft.de, 2005.

[22] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller, "Towards a verified, general-purpose operating system kernel," in *Proc. NICTA FM Workshop on OS Verification* (G. Klein, ed.), pp. 1–19, Technical Report 0401005T-1, National ICT Australia, 2004.

[23] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[24] F. Nielson, H. R. Nielson, and C. L. Hankin, *Principles of Program Analysis*. Springer, 1999.

[25] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft," in *Integrated Formal Methods: 4th International Conference, IFM 2004*, vol. 2999 of *LNCS*, pp. 1–20, Springer-Verlag, Jan. 2004.

[26] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, Oct. 2000.

[27] G. Bernat, A. Colin, and S. M. Petters, "pWCET: a tool for probabilistic worst case execution time analysis of real–time systems," technical report YCS353 (2003), University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Apr 2003.

[28] "Rapita systems limited." http://www.rapitasystems.com/.

[29] B. Leslie, C. van Schaik, and G. Heiser, "Wombat: A portable user-mode Linux for embedded systems," in *6th Linux.Conf.Au*, (Canberra), Apr 2005.