The Road to Trustworthy Systems

Gernot Heiser,
June Andronick, Kevin Elphinstone, Gerwin Klein, Ihor Kuz and Leonid Ryzhyk
NICTA and University of New South Wales
Sydney, Australia
ertos@nicta.com.au

ABSTRACT

Computer systems are routinely deployed in life- and mission-critical situations, yet in most cases their security, safety or dependability cannot be assured to the degree warranted by the application. In other words, trusted computer systems are rarely really trustworthy.

We believe that this is highly unsatisfactory, and have embarked on a large research program aimed at bringing reality in line with expectations. In this paper we describe NICTA's research agenda for achieving true trustworthiness in systems. We report on what has been achieved to date, and what our plans are for the next 3–5 years.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Verification*

General Terms

Design, Reliability, Security, Verification

Keywords

Microkernels, L4, embedded systems, trusted systems, formal verification

1. THE TRUSTWORTHINESS GAP

1.1 Trust is Ubiquitous

Computer systems are an unavoidable part of everyday life. The vast majority of the population uses email for communication and web sites for information and commerce; these are accessed by personal computers, PDAs or phones and are powered by server systems.

Besides these obvious ways in which we interact with computers, there is the wealth of embedded systems, which are one or two orders of magnitude more numerous than servers or PCs. These range from kitchen appliances and home entertainment systems to cars, aeroplanes, cash-dispensing machines, the ubiquitous mobile phones, smart cards, medical devices, public-security surveillance equipment and countless others.

With integration comes dependence: as computer systems are increasingly part of everyday life, everyday life depends increasingly on the correct functioning of these systems. The effects of faulty operation range from serious nuisance (if the smart card fails to open the door or the ATM refuses to supply me with cash) to significant financial loss (if an e-commerce system fails to service customers, financial systems get hacked or device failures lead to recalls) to compromise of national security (when secret information gets stolen) to loss of life (if heavy machinery or medical devices get out of control).

Whether we like it or not, we put an increasing amount of trust into an increasing number of computer systems. In a modern advanced society it is simply no longer feasible to avoid trusting a large number of computer systems. And in the foreseeable future this ubiquitous need for trust will only intensify.

1.2 ... But Trustworthiness Isn't

This is in stark contrast to the *trustworthiness* (or lack thereof) of real systems. We are used to laptops crashing and servers being hacked. We are now getting used to phones crashing and getting hacked [ABC09]. We hear about hacking ATMs [Nar10], voting machines [FHF07] and even heart pacemakers [HCF+08].

These are scary scenarios. They all relate to systems we routinely *trust*, yet these events demonstrate their lack of *trustworthiness*. The reality is that *trustworthy systems are extremely rare*.

Where trustworthiness is important, people have to go to extraordinary lengths to achieve it. A high degree of redundancy is used in safety-critical components in airplanes [BT93]. Certification standards for aeronautics [RTC92] and defence [ISO99] impose extremely onerous requirements on development processes and testing. Where formal methods are employed [SWDD09] or required [ISO99], these augment the process and testing requirements, without replacing them. In the end, these standards cannot guarantee safety or security, despite the expense they impose on developers (estimated at $1,000 per line of code (LOC) for Common Criterial EAL6 [Har04]). In fact, their emphasis on process and testing is an admission of imperfection.

In order to achieve an approximation of trustworthiness with traditional means, and keep costs within limits, developers aim to keep the amount of code in critical systems as small as possible. This means that code is written to run stand-alone on bare hardware, without an operating system. This approach forces developers to isolate each piece of functionality on separate hardware. This further increases cost, not only by increasing the bill of materials for the additional processors, power supplies, communication ports, wires etc. It also increases weight, volume, power consumption,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
STC'10, October 4, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0095-7/10/10 ...$10.00.

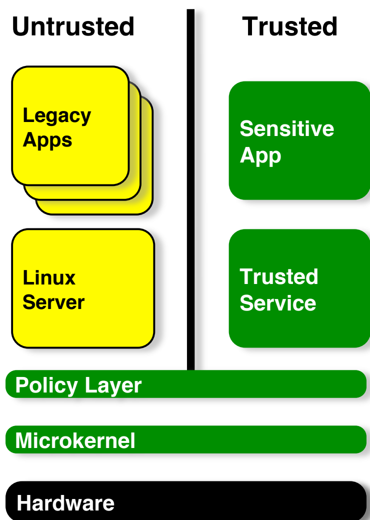

Figure 1: Trusted system with untrusted legacy components. Trusted components are dark.

and heat dissipation, which often results in significant secondary costs.

1.3 Affordable Trustworthiness

The minimalist approach to (an approximation of) trustworthiness is not only costly, it is also fundamentally at odds with increasing functionality requirements.

For example, life-supporting medical devices are moving out of intensive-care units and the control of well-trained specialist operators and are becoming wearable, and thus patient-operated. This implies a need for easy-to-use operator interfaces, which tend to require complex software stacks.

Similarly, there is a growing demand for financial transactions to be performed on mobile devices, which run software stacks comprising millions of lines of code. In automobiles, sophisticated driver-assist systems, like electronic vehicle stabilisation, integrate infotainment with the core propulsion and security functionality. And national security personnel is increasingly dependent on commercial of-the-shelf (COTS) devices for secure communication.

All these use cases have in common that they require a high degree of trustworthiness, at a cost which is orders of magnitude less than the current $1k/LOC price tag for serious (yet still far from perfect) certification regimes. In other words, there is a strong need for *affordable trustworthiness*.

Achieving affordable trustworthiness is the long-term goal of NICTA's Trustworthy Embedded Systems research agenda. In the rest of this paper we will outline this agenda, summarise achievements to date, and detail the research plan for the next four years.

2. TRUSTED SYSTEMS ARCHITECTURE

Real-world trusted systems will have to be able to incorporate large legacy components. For example, it is not cost-effective to re-design and -develop GUI stacks, as required for user-friendly interfaces, from scratch. And even if this was a real possibility, the resulting code base would be very large, even if unusually well-designed. Consequently, the cost of making the complete GUI stack truly trustworthy will remain prohibitive for a long time hence.

This means that trusted systems need to incorporate large, untrusted legacy components. These must not only co-exist, but cooperate with the critical system components. In other words, we need a design like the one shown in Figure 1, where a trusted system is designed from trusted and untrusted components.

This architecture requires an underlying trusted substrate which strongly separates the various subsystems. The approach is based on the idea of a *separation kernel* [Rus81], which forms the base of some trusted systems in the military domain [IAD07].

However, a classic separation kernel is insufficient for the needs of many trusted systems. For example, the scheduling model (based on cooperative scheduling) is too simplistic for most real-world embedded systems. Furthermore, the separation kernel is focussed on isolation of several subsystems, effectively emulating a design where each subsystem runs on a dedicated computing platform, and communication occurs via a network.

The strong separation model does not fit most real-world embedded systems, where the whole point of including the legacy components is that they need to cooperate closely with the trusted components to achieve the overall system mission. This frequently includes the need for high-bandwidth, low-latency communication and (secure) access to shared data structures.

The underlying kernel must provide a range of mechanisms which allow the efficient implementation of a wide range of possible system designs, ranging from strong isolation to tight (but secure) integration. It requires a high-performance, general-purpose microkernel, such as L4 [Lie95].

Obviously, the microkernel is always trusted, it is an inevitable part of the system's *trusted computing base* (TCB), and its trustworthiness is critical. The microkernel, being a general platform used in all trusted systems, should be free of any particular system's policies. Policies are implemented by a software layer outside the kernel, but enforced by kernel mechanisms. Note that the kernel is the only software executing in the most privileged mode of the hardware.

One practical requirement on the architecture is that it provides virtual machines, in order to allow running a complete legacy operating system (OS) as part of the software stack, such as the *Linux server* shown in Figure 1. The L4 microkernel has been successfully used as a hypervisor for para-virtualized Linux systems for many years [HHL+97, LvSH05], so is potentially a suitable platform.

3. TRUSTWORTHY MICROKERNEL

As the microkernel is a critical (and fully reusable) part of the overall architecture, we focussed on it first. We had in the past evolved the L4 microkernel for use in resource-constrained embedded systems [NIC05] without sacrificing the excellent performance that is the hallmark of L4; the outcome of that evolution was NICTA's L4-embedded kernel.

A pilot project, which formalised parts of the L4 API and investigated the use of theorem-proving techniques on some part of the implementation showed that, in principle, formal verification of such a kernel is feasible [TKH05]. However, we also found that L4-embedded, as it existed then, was not the most suitable platform for such work.

There were several main reasons for that conclusion. One was that the kernel had been designed for portability and high performance, but not for verification. For example, it had known problems where performance had been chose over consistency, simplicity and correctness. In general, the implementation made little attempt to make correctness obvious. The design was not explicit or well documented, much critical information only existed in people's heads.

Then the kernel was implemented in C++, a language of significantly more complexity and ambiguity than C. While some progress has been made elsewhere on formalising the semantics of C++ [HT03], we had already come to the conclusion that C was a better implementation vehicle for the kind of kernel we were looking for, and that the extra effort for formalising C++ was not warranted.

Finally, the kernel API had been designed for flexibility, not for the requirements of safety- or security-critical systems. For example it used globally unique thread names as destination addresses for message-passing (IPC) operations. Such global names represent a covert storage channel and should not be used in a security-oriented design [Sha03]. Also, L4's approach to management of kernel resources (while not worse than most other kernels of similar generality) made it difficult to reason about isolation of processes running on top of the kernel.

We therefore embarked on a project aimed at designing and implementing a new kernel, dubbed *seL4* ("secure embedded L4"). The kernel was to satisfy the following requirements:

- suitable for use in security- and safety-critical systems, including the ability to enforce strong isolation of subsystems;
- suitable for formal verification;
- high performance, specifically an IPC operation should not be more than 10 % slower than that of the fastest L4 kernel (which translates into 150–200 cycles per one-way IPC on an ARM processor).

The first of these requirements was partially met by a new approach to managing kernel memory. Access to all kernel objects in seL4 is controlled by kernel-protected capabilities [DVH66]. Furthermore, seL4 does not have a heap, it performs no dynamic memory allocation once the kernel is fully initialised. Instead, all memory required by the kernel to implement kernel objects (such as address spaces, threads and IPC endpoints) must be provided to the kernel by user-level threads. Hence the kernel will only use memory on behalf of a user-level activity if that activity possessed the memory in the first place. This together with the rest of the seL4 design enables strong memory isolation; in particular it is crucial in ruling out denial-of-service attacks against the kernel or isolated subsystems.

The formal verification had to proceed concurrently with the design and implementation of the kernel. This posed additional challenges, as for well over half of the duration of the project, the verification team was chasing a moving target. Very close interaction on a daily basis between the kernel-design and the verification teams was essential.

The key to making this work turned out to be a fast-prototyping approach based around an intermediate representation of seL4: a Haskell implementation [EKD+07]. This allowed us to follow design changes quickly with an executable implementation, which could readily be imported into the theorem prover. The Haskell-based design was straightforward to (manually) implement in high-performance C code including the optimisations typical to the L4 family.

The approach lead to the iterative design process shown in Figure 2. Compared to our previous experience with kernel design and implementation, as well as cost estimates produced by the SLOC-Count program [Whe01], we estimate that this process has given us at least a two-fold productivity increase, besides producing the formal artefacts essential for the verification effort.

The design and implementation of seL4 was then formally verified [KEH+09] by means of interactive theorem proving. Specifically, we produced a formal proof, machine-checked in the theorem prover Isabelle/HOL [NPW02], of *functional correctness of the implementation*. This means that the set of all possible behaviours of the C code of seL4, according to the operational semantics of C [WKS+09], is a subset of the behaviours allowed by the specification. This makes seL4 the first general-purpose operating-system kernel formally proved to be functionally correct.

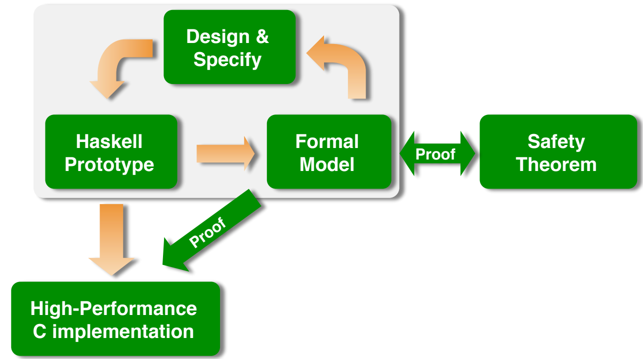

Figure 2: Iterative design process of seL4.

4. BUILDING TRUSTWORTHY SYSTEMS

The verified seL4 microkernel is, of course, only a first step. It provides a rock-solid foundation for the next phase of our research agenda. Our aim for the next 3–5 years is to develop approaches and frameworks which allow the design and implementation of real-world systems incorporating large legacy subsystems comprising millions of lines of code with strong dependability guarantees (meaning guarantees of system-wide safety, security or reliability properties).

Achieving this overall goal will require significant progress in a number of related areas:

- system architecture combining trusted and untrusted components in a way which ensures that untrusted components cannot interfere with trusted ones;
- information-flow analysis of the microkernel and systems built on top;
- trustworthy device drivers;
- worst-case execution-time analysis of the microkernel;
- support for multicore platforms;
- ability to ensure non-functional system-wide properties, such as timeliness or limits on energy use.

4.1 Full-system dependability guarantees

Our approach to constructing trustworthy systems is illustrated in Figure 3. In the case of a secure system, the design begins with the specification of required system security properties (here an information flow-property). In the general case, this may also be a safety or reliability property such as integrity or fault isolation.

Given this specification, the system's software architecture (labelled ① in Figure 3) is designed such that it achieves the desired security goal with a TCB. In particular, the architecture description will specify the components involved, must specify the connections between the components, and must identify the trusted components

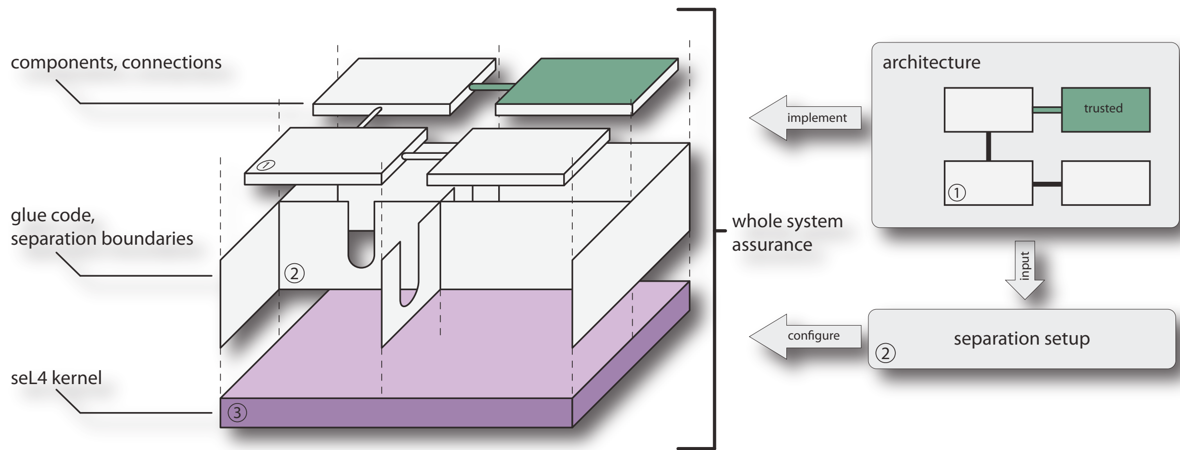

Figure 3: seL4-based system with multiple independent levels of assurance

as well as any isolation and communication restrictions. The architecture is designed using existing design methodologies, allowing existing architecture analysis methods to be used for these high assurance systems.

In the architecture, trusted components are those that could circumvent the reliability or security properties of the system, i.e. the system's TCB. In order to maintain the required security or reliability properties, trusted components are expected to behave in a particular way, and these behavioural requirements are specified as part of the architecture.

The architecture description is then validated against the system policy to ensure that it does in fact provide the required properties. For specific sets of security and safety properties (such as label-based security policies) we will provide tools that can automate this analysis and produce formal proofs.

The next step is to provide implementations of the components specified by the architecture. These components can be OS components (drivers, resource management, etc.), middleware components (such as naming and communication servers), or application specific components. The components are either implemented from scratch or chosen from a library of pre-existing reusable components.

For assuring trusted components there are many existing approaches and much research has investigated component creation. We will not mandate any particular approach, however, the approach taken should provide sufficient confidence that the components fulfil their requirements. We will then provide a formal framework into which the given individual assurance can be integrated. One such approach is formal code proof, as was done in the formal verification of seL4. Another such approach is the synthesis of component implementations from formal specifications, as demonstrated in our work on device-driver synthesis [RCK+09]. This approach is less resource-intensive than manual proof and can produce a similar level of assurance.

Finally, given the architecture description and component implementations, we will generate the code required to combine the component implementations into a running system. This includes generated glue and communication code as well as initialisation and bootstrap code (which is responsible for loading and initialising the required components, creating the appropriate partitions—shown as ② in Figure 3—and configuring communication channels as specified by the architecture). This step will be performed automatically by a tool (evolved from our CAmkES project [KLGH07] that developed a lightweight component framework for embedded systems) which will also automatically integrate the proofs of architecture correctness, component correctness, partition configuration and kernel correctness into one overall statement on the whole system.

Summarising with reference to Figure 3, our approach will result in components (the top layer in the diagram, labelled ①) assured to the individual levels that the overall system requires, and a system architecture shown to enforce the component isolation and communications requirements (the walls in the diagram, labelled ②). These are composed such that individual component assurance combines into strong system-level assurance, and are supported by a platform that provides the system's execution environment and isolation guarantees (the seL4 kernel, labelled ③). The important part is that the separation walls come with a strong foundation that enables us to ignore non-critical components for assurance purposes.

The approach requires us to solve a number of issues, including the formal semantics of components and their composition, the correct-by-construction generation of glue code, formal analysis of information flow, and concurrency issues which arise from the (even in the embedded space) increasingly ubiquitous multicore platforms. More detail is available in the project plan for the Trustworthy Embedded Systems project [ERTO09].

4.2 Device drivers

Device drivers are well known to be the leading hazard to operating-system (OS) trustworthiness: they are responsible for the majority of OS failures [GGP06], and have been shown to have 3–7 times the defect density of other OS code [CYC+01].

In a microkernel-based system, such as one based on seL4, device drivers run in user mode, as normal processes, encapsulated in their own address space. This encapsulation can be enforced through hardware mechanisms, such as the memory-management unit (MMU) and, for devices performing direct memory access (DMA) an I/O MMU. This approach, if done right, does not impose much overhead [LCFD+05], and allows isolating the rest of the system from device-driver failures.

While this approach removes a device driver from the TCB of a subsystem which does not use the device, many devices are essential to the overall system function, and a system may be unable to operate correctly if a critical driver fails even temporarily.

We are therefore continuing to work on making the drivers them-

selves reliable. Specifically we are pursuing a number of approaches, distinguished by the time to practical use.

The first, short-term approach is a change of the established driver model from a multi-threaded to an event-driven paradigm. We have shown that this approach avoids several classes of the most frequent driver bugs by construction [RCKH09]. Drivers designed in this model can co-exist with traditional drivers, easing adoption. We are working on turning the original proof-of-concept into a practical framework for driver development [RZH10].

The second, medium-term approach combines driver development with hardware "verification" (i.e. testing), re-using the testing code as the actual device driver [LCFD+05]. This approach is designed to integrate with industrial device-development workflows, but eliminates the biggest problem of those workflows: they develop elaborate testing code during device development, which contains all the core logic of a device driver, but is discarded once the device is completed, while the actual driver is developed under time pressure on the basis of second-hand and frequently incomplete and inaccurate information (contained in the device's datasheet).

The third, long-term approach extends our earlier work on driver synthesis [RCK+09], which generated a high-performance device driver from a formal spec of the OS interface, a formal spec of the hardware interface, and a high-level behavioural description of the device class. The Achilles heel of that work is the impracticality of obtaining the formal spec of the hardware interface. We are working on bypassing this problem by automatically extracting the required specifications from the device design itself, as represented in a high-level specification language such as SystemVerilog or SystemC or a register transfer level description in Verilog or VHDL.

4.3 Temporal properties

Two kinds of timing issues are relevant to our overall research agenda: timeliness of execution, as required for real-time systems, and the security threats created by information leakage through covert timing channels.

Being able to build real-time systems on top of our microkernel requires a timing model of the kernel, specifically the knowledge of the worst-case latencies of all kernel operations which are used during the steady-state operation of a system. No practical OS kernel for a high-end processor, such as a recent ARM CPU, has such a timing model with credible latency guarantees. All prior work is based on over-simplified kernels or simple (and outdated) architectures.

We believe that we can achieve more, despite our kernel being more complex than some academic toy systems, because we know more about the kernel. Specifically, the formal verification of seL4 has proved a large number of invariants about the kernel, many of which are of direct help for solving the worst-case latency problem. These include termination proofs, limits on iteration counts, and alignment guarantees.

In the past, worst-case latency analysis has focussed either on static analysis (requiring information about the internal timings of the processor which in reality is impossible to obtain) or measurements (which alone cannot give any guarantees). We believe that a combination of both approaches is the key to a practical, yet sound solution [PZH07].

The other timing issue regards covert channels. We are working on analysing the bandwidth of covert timing channels in seL4, as a precursor to steps which suppress this bandwidth. In this context we are developing an information-theoretical framework for dealing with covert channels in the kernel, which will help us to evaluate various mitigation strategies.

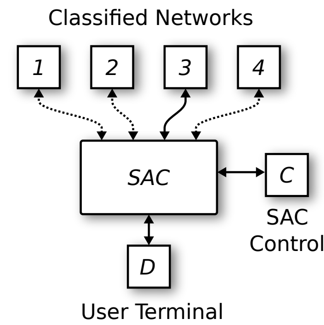

Figure 4: Secure access controller connecting a secure terminal to multiple classified networks.

4.4 Practical applications

We strongly believe that a project like ours will only produce results of practical use if the techniques developed are applied to concrete use cases from an early stage. We are therefore actively engaging with actual or potential end-users of our technology to build prototype trustworthy devices and use them as drivers for our research.

One such prototype is a *secure access controller* which multiplexes several networks of different classification on a single connection to a secure terminal, as shown in Figure 4. We have built such a prototype on seL4 [AGE10], including several Linux systems running in separate virtual machines as wrappers for legacy device drivers. We have analysed the TCB of the system and the feasibility of formally verifying all code which makes up the TCB. We have also proved the basic security properties of the SAC (under the assumption that the TCB is correct).

5. CONCLUSIONS

We believe that truly trustworthy systems are not only needed, but that they are achievable within a few years. A first step has been taken with the formal verification of the seL4 microkernel, but much more remains to be done.

We have presented NICTA's research roadmap which aims to deliver within the next 3–5 years truly trustworthy systems of real-life complexity, comprising millions of lines of code, including large, untrusted legacy components. Their trustworthiness will be established with mathematical rigor. Nothing less ought to be good enough.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

This material is in part based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-09-1-4160. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

We also gratefully acknowledge a Google Faculty Research Award.

6. REFERENCES

[ABC09] Australian admits creating first iPhone virus. http://www.abc.net.au/news/stories/2009/11/09/2737673.htm, November 2009.

[AGE10] J. Andronick, D. Greenaway, and K. Elphinstone. Towards proving security in the presence of large untrusted components. In G. Klein, R. Huuck, and B. Schlich, editors, *Proceedings of the 5th Workshop on Systems Software Verification*, Vancouver, Canada, October 2010. USENIX.

[BT93] D. Brière and P. Traverse. Airbus A320/A330/A340 electrical flight controls: A family of fault-tolerant systems. In *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing*, pages 616–623, 1993.

[CYC+01] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[DVH66] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

[EKD+07] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, San Diego, CA, USA, May 2007.

[ERTO09] Trustworthy embedded systems—ERTOS-2 project plan 2009–2013. http://ertos.nicta.com.au/publications/papers/ERTOS_09.abstract, July 2009. NICTA.

[FHF07] A. J. Feldman, J. A. Halderman, and E. W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. In *USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Boston, MA, USA, August 2007. http://citp.princeton.edu/voting.

[GGP06] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th USENIX Large Installation System Administration Conference*, pages 101–111, Washington, DC, USA, 2006.

[Har04] B. Hart. SDR security threats in an open source world. In *Software Defined Radia Conference*, pages 3.5–3 1–4, Phoenix, AZ, USA, November 2004.

[HCF+08] D. Halperin, S. S. Clark, K. Fu, T. S. Heydt-Benjamin, B. Defend, T. Kohno, B. Ransford, W. Morgan, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 129–142, Oakland, CA, USA, May 2008.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, October 1997.

[HT03] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In *Emerging Trends Track, TPHOLs*, Rome, Italy, September 2003.

[IAD07] Information Assurance Directorate. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, June 2007. Version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

[ISO99] *Information Technology — Security Techniques — Evaluation Criteria for IT Security*, 1999. ISO/IEC International Standard 15408.

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[KLGH07] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):687–699, May 2007.

[LCFD+05] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[Lie95] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[LvSH05] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, April 2005.

[Nar10] R. Naraine. Hacker demos remote attacks against ATMs. http://threatpost.com/en_us/blogs/hacker-demos-remote-attacks-against-atms-072810, July 2010.

[NIC05] NICTA. *NICTA L4-embedded Kernel Reference Manual Version N1*, October 2005. http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[PZH07] S. M. Petters, P. Zadarnowski, and G. Heiser. Measurements or static analysis or both? In *Proceedings of the 7th Workshop on Worst-Case Execution-Time Analysis*, Pisa, Italy, July 2007. Satellite Workshop of the 19th Euromicro Conference on Real-Time Systems.

[RCK+09] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009.

[RCKH09] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, April 2009.

[RTC92] RTCA. *DO-178B: Software Considerations in*

Airborne Systems and Equipment Certification, December 1992.

[Rus81] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 12–21, 1981.

[RZH10] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, New Delhi, India, August 2010.

[Sha03] J. S. Shapiro. Vulnerabilities in synchronous IPC designs. In *Proceedings of the IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2003.

[SWDD09] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proceedings of the 16th International Symposium on Formal Methods*, pages 532–546, Eindhoven, The Netherlands, November 2009.

[TKH05] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, Santa Fe, NM, USA, June 2005. USENIX.

[Whe01] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2001.

[WKS+09] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515, Munich, Germany, August 2009. Springer.