From a Verified Kernel Towards Verified Systems

Gerwin Klein

[1] NICTA, Australia

[2] School of Computer Science and Engineering, UNSW, Sydney, Australia

gerwin.klein@nicta.com.au

Abstract. The L4.verified project has produced a formal, machine-checked Isabelle/HOL proof that the C code of the seL4 OS microkernel correctly implements its abstract implementation. This paper briefly summarises the proof, its main implications and assumptions, reports on the experience in conducting such a large-scale verification, and finally lays out a vision how this formally verified kernel may be used for gaining formal, code-level assurance about safety and security properties of systems on the order of a million lines of code.

1 L4.verified

In previous work [13], we reported on the result of the L4.verified project: a machine-checked, formal verification of the seL4 operating system microkernel from a high-level model in Higher-Order logic down to low-level C code.

To the best of our knowledge, this is the first complete code-level proof of any general-purpose OS kernel, and in particular the first machine-checked such proof of full functional correctness.

Early pioneering attempts at formal OS verification like UCLA Secure Unix [20] or PSOS [9] did not proceed substantially over the specification phase. In the late 1980s, Bevier's KIT [2] is the first code-level proof of an OS kernel, albeit only a very simple one. There have been a number of formal verifications of either functional correctness, temporal, or information flow properties of OS kernels, recently for instance the Common Criteria EAL6+ certified INTEGRITY kernel [18]. None of these, however, truly formally verified the code-level implementation of the kernel. Instead, what is verified is usually a formal model of the code, which can range from very precise as in the INTEGRITY example to design-level or more abstract models. Correspondence between C code as seen by the compiler and the formal model is established by other means. In the L4.verified project, this critical missing step is for the first time formal and machine-checked.

Contemporary OS verification projects include Verisoft, Verisoft XT, and Verve. The Verisoft project has not yet fully completed all parts of its OS kernel proof, but it has conclusively demonstrated that formal verification of OS code can be driven down to verified hardware — similarly to the verified CLI stack [3] from the 1980s, but going up to a verified C0 compiler with support for inline assembly and up to substantial scale. The Verisoft XT project [7] has demonstrated that the technology exists to deal with concurrent C at a scale of tens of thousands lines of code. The Verve kernel [22] shows that type and memory safety properties can be established on the assembly level via type systems and therefore with much lower cost. Verve contains a formally verified runtime system, in particular a garbage collector that the type system relies on. Even though it only shows type safety, not functional correctness, the smaller cost of verification makes the approach attractive for larger code bases if full functional correctness is not required or too expensive to obtain.

The formal proof for the seL4 kernel establishes a classical functional correctness result: all possible behaviours of the C implementation are already contained in the behaviours of its abstract specification. In the L4.verified project, this proof was conducted in two stages in the interactive theorem prover Isabelle/HOL [17]. The first stage is comparable to other detailed model-level kernel verifications. It connects an abstract, operational specification with an executable design specification of the kernel. This design specification is low-level enough to clearly see a direct one-to-one correspondence to C code for the large majority of the code. The second step in the proof was to show that the C code implements this low-level design. The result is one concise overall theorem in Isabelle/HOL stating that the behaviour of the C code as specified by its operational semantics is contained in the behaviours of the specification.

Like any proof, this verification has assumptions. For the correctness of a running seL4 system on real hardware we need to assume correctness of the C compiler and linker, assembly code, hardware, correct use of low-level TLB and cache-flushing instructions, and correct boot code. The verification target was the ARM11 uniprocessor version of seL4. There also exists an (unverified) x86 port of seL4 with optional multi-processor and IOMMU support.

The key benefit of a functional correctness proof is that proofs about the C implementation of the kernel can now be reduced to proofs about the specification if the property under investigation is preserved by refinement. Additionally, our proof has a number of implications, some of them desirable direct security properties. If the assumptions of the verification hold, we have mathematical proof that, among other properties, the seL4 kernel is free of buffer overflows, NULL pointer dereferences, memory leaks, and undefined execution. There are other properties that are not implied, for instance general security without further definition of what security is or information flow guaranties that would provide strict secrecy of protected data. A more in-depth description of high-level implications and limitations has appeared elsewhere [12,11].

2 What have we learned?

To be able to successfully complete this verification, we have contributed to the state of the art in theorem proving and programming languages on a number of occasions, including tool development [16], memory models [19], and scalable refinement frameworks [6,21]. These are published and do not need to be repeated in detail here. Other interesting aspects of the project concern lessons that are harder to measure such as proof engineering, teaching theorem proving to new team members, close collaboration between the kernel and verification teams, and a prototyping methodology for kernel development.

On a higher level, the main unique aspects of this project were its scale and level of detail in the proof. Neither would have been achievable without a mechanical proof assistant. The proof, about 200,000 lines of Isabelle script, was too large for any one person in the team to fully keep in their head, and much too large and technically involved to manually check and have any degree of confidence in the result. Software verifications like this are only possible with the help of tools.

The cost of the verification was around 25 person years counting all parts of the project, including exploratory work and models that were later not used in the verification. About twelve of these person years pertain to the kernel verification itself. Most of the rest was spent on developing frameworks, tools, proof libraries, and the C verification framework, including a precise memory model [19] and a C to Isabelle/HOL parser [21].

This means, we have demonstrated that proving functional correctness of low-level C code is possible and feasible at a scale of about 10,000 lines of code, but the cost is substantial. Clearly, we have to conclude that currently this approach does not lend itself to casual software development.

The story is different for high-assurance systems. It is currently very expensive to build truly trustworthy systems and to provide substantial assurance that they will indeed behave as expected. It is hard to get useful numbers for such comparisons, but one data point that is close enough, and where some experience and cost estimates are available, are Common Criteria (CC) security evaluations. CC on high evaluation levels prescribe the use of formal specifications and proofs down to the design level. Correspondence of models to code is established by testing and inspection.

L4.verified spent about $700 per line of code (loc) for the verification if we take the whole 25 person years, and less than $350/loc if we take the 12 actually spent on the kernel. We estimate that, with the experience gained and with the tools and libraries available now, the cost could be further reduced to 10, maybe 8 person years for a similar code base verified by the same team, i.e. about $230/loc. Even assuming $350/loc, the verification compares favourably with the quoted cost for CC EAL6 evaluation at $1000/loc [10]. EAL7 (the highest CC level) which arguably still provides less assurance than formal code-level proof, can safely be assumed to be more costly still. The comparison is not entirely fair, since the Common Criteria mostly address security properties and not functional correctness, and because the verification aspect is only one of the aspects of the certification process. On the other hand one can argue that general functional correctness is at least as hard to prove as a specific security property and that while verification is not the only aspect, it is the most expensive one. We believe that formal, code-level verification is cost attractive for the vendor as well as for the certification authority, while increasing assurance at the same time.

For the certification authority, risk is reduced. Since the proof is machine-checked, only the high-level specification and its properties as well as the bottom-level model need to be scrutinised manually and with care to trust the system. Validating the high-level properties is the same as in the current evaluation scheme. The bottom-level model, however, is different. In the current scheme, the bottom level model is different for each certification and needs to be connected to code by careful validation, testing and inspection which is expensive to conduct and hard to check. In our case, the model does not depend on the certification artefact: it is just the semantics of our subset of C. Once validated, this could be re-used over many certifications and amortised to gain even higher assurance than what would otherwise be cost effective.

Our result of feasible but high-cost verification at about 10,000 loc does not mean that formal verification could not scale further. In fact, microkernels such as seL4 typically lack two properties that make formal verification scale better: modularity and strong internal abstractions. We would expect application-level code and even user-level OS code to be much better targets for scalable, compositional verification techniques.

However, even with nicely structured code, it appears infeasible at this stage to formally verify the functional correctness of systems with millions of lines of code. The field is making progress in scaling automated techniques for reasonably simple properties to such systems, but complex safety or security properties or properties that critically rely on functional correctness of at least parts of the system still appear without our reach.

3 A Secure System with Large Untrusted Components

This section presents a vision of how assurance even of complex safety properties could nevertheless be feasibly be achieved within (or close to) the current state of the art in code-level formal proof.

The key idea is the original microkernel idea that is also explored in the MILS (multiple independent levels of security and safety) space [4]: using system architectures that ensure security by construction, relying on basic kernel mechanisms to separate trusted from untrusted code. Security in these systems is not an additional feature or requirement, but fundamentally determines the core architecture of how the system is laid out, designed, and implemented.

This application space was one of the targets in the design of the seL4 kernel. Exploiting the verified properties of seL4, we should be able to architect systems such that the trusted computing base for the desired property is small and amenable to formal verification, and that the untrusted code base of the system provably cannot affect overall security.

The basic process for building a system in this vision could be summarised as follows:

1. Architect the system on a high level such that the trusted computing base is as small as possible for the security property of interest.

2. Map the architecture to a low-level design that preserves the security property and that is directly implementable on the underlying kernel.
3. Formalise the system, preferably on the architecture level.
4. Analyse, preferably formally prove, that it enforces the security property. This analysis formally identifies the trusted computing base.
5. Implement the system, with focus for high assurance on the trusted components.
6. Prove that the behaviour of the trusted components assumed in the security analysis is the behaviour that was implemented.

The key property of the underlying kernel that can make the security analysis feasible is the ability to reduce the overall security of the system to the security mechanisms of the kernel and the behaviour of the trusted components only. Untrusted components will be assumed to do anything in their power to subvert the system. They are constrained only by the kernel and they can be as big and complex as they need to be. Components that need further constraints on their behaviour in the security analysis need to be trusted to follow these constraints. They form the trusted components of the system. Ideally these components are small, simple, and few.

In the following subsections I demonstrate how such an analysis works on an example system, briefly summarise initial progress we have made in modelling, designing, formally analysing, and implementing the system, and summarise the steps that are left to gain high assurance of overall system security. A more detailed account is available elsewhere [1].

The case study system is a secure access controller (SAC) with the sole purpose of connecting one front-end terminal to either of two back-end networks one at a time. The back-end networks A and B are assumed to be of different classification levels (e.g. top secret and secret), potentially hostile and collaborating. The property the SAC should enforce is that no information may flow through it between A and B.

3.1 Architecture

Figure 1 shows the high-level architecture of the system. The boxes stand for software components, the arrows for memory or communication channel access. The main components of the SAC are the SAC Controller (SAC-C), the Router (R), and the Router Manager (RM). The Router Manager is the only trusted user-level component in the system. The system is implemented on top of seL4 and started up by a user-level booter component. The SAC Controller is an embedded Linux instance with a web-server interface to the front-end control network where a user may request to be connected to network A or B. After authenticating and interpreting such requests, the SAC Controller passes them on as simple messages to the Router Manager. The Router Manager receives such switching messages. If, for example, the SAC is currently connected to A, there will be a Router instance running with access to only the front-end data network card and the network card for A. Router instances are again embedded Linuxes

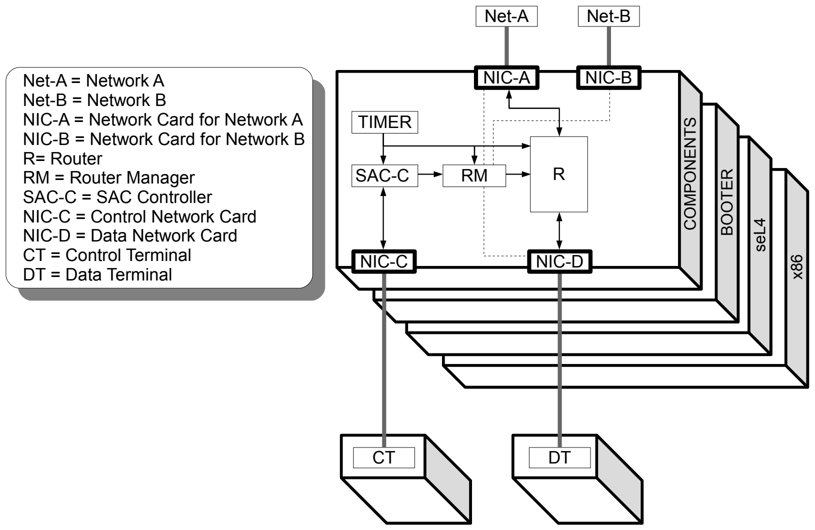


Fig. 1. SAC Architecture

with a suitable implementation of TCP/IP, routing etc. If the user requests a switch to network B, the Router Manager will tear down the current A-connected Linux instance, flush all network cards, create a new Router Linux and give it access to network B and the front end only.

The claim is that this architecture enforces the information flow property. Each Router instance is only ever connected to one back-end network and all storage it may have had access to is wiped when switching. The Linux instances are large, untrusted components in the order of a million lines of code each. The trusted Router Manager is small, about 2,000 lines of C.

For this architecture to work, there is an important non-functional requirement on the Linux instances: we must be able to tear down and boot Linux in acceptable time (less than 1-2 seconds). The requirement is not security-critical, so it does not need to be part of the analysis, but it determines if the system is practical. Our implementation achieves this.

So far, we have found an architecture of the system that we think enforces the security property. The next sections explore design/implementation and analysis.

3.2 Design and implementation

The main task of the low-level design is to take the high-level architecture and map it to seL4 kernel concepts. The seL4 kernel supports a number of objects for threads, virtual memory, communication endpoints, etc. Sets of these map to components in the architecture. Access to these objects is controlled by capabilities: pointers with associated access rights. For a thread to invoke any operation on an object, it must first present a valid capability with sufficient rights to that object.

Figure 2 shows a simplified diagram of the SAC low-level design as it is implemented on seL4. The boxes in the picture stand for seL4 kernel objects, the arrows for seL4 capabilities. The main message of this diagram is that it is significantly more complex than the architecture-level picture we started out with. For the system to run on an x86 system with IOMMU (which is necessary to achieve untrusted device access), a large number of details have to be taken care of. Access to hardware resources has to be carefully divided, large software components will be implemented by sets of seL4 kernel objects with further internal access control structure, communications channels and shared access need to be mapped to seL4 capabilities, and so forth.

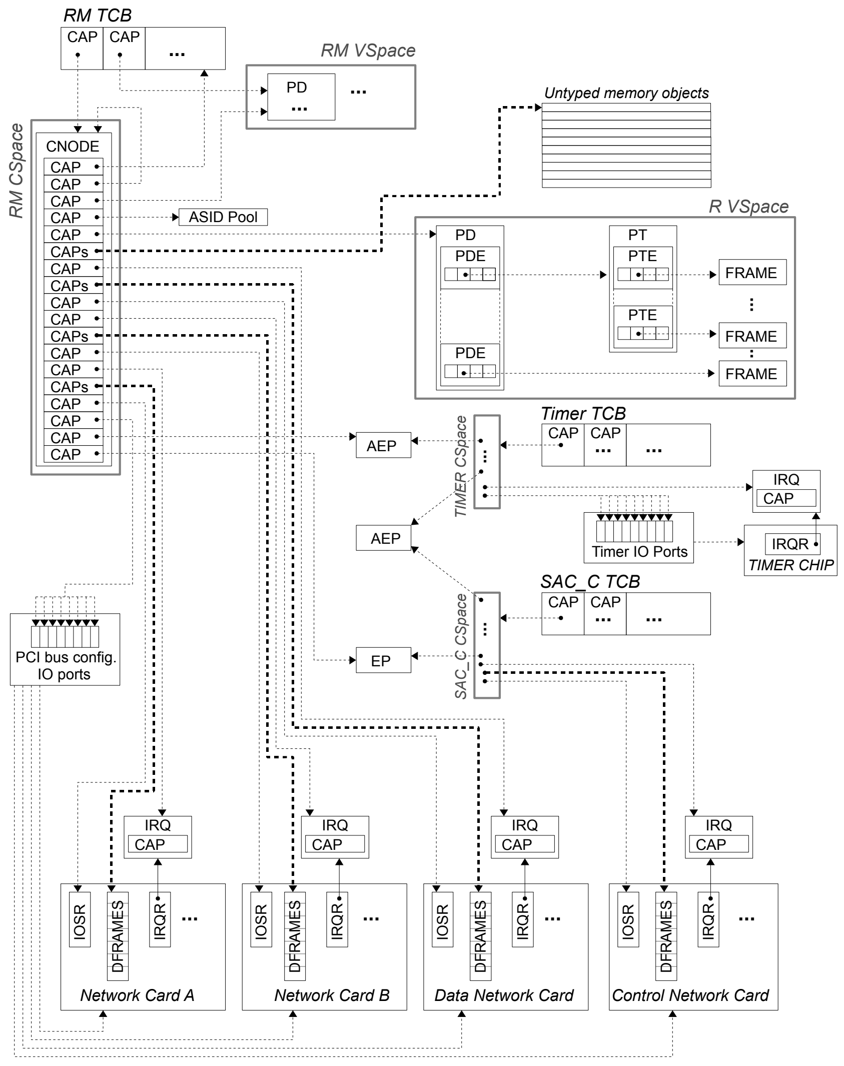
The traditional way to implement a picture such as the one in Figure 2 is by writing C code that contains the right sequence of seL4 kernel calls to create the required objects, to configure them with the right initial parameters, and to connect them with the right seL4 capabilities with the correct access rights. The resulting code is tedious to write, full of specific constants, and not easy to get right. Yet, this code is crucial: it provides the known-good initial capability state of the system that the security analysis is later reduced to.

To simplify and aid this task, we have developed the small formal domain-specific language capDL [15] (capability distribution language) that can be used to concisely describe capability and kernel object distributions such as Figure 2. A binary representation of this description is the input for a user-level library in the initial root task of the system and can be used to fully automatically set up the initial set of objects and capabilities. Since capDL has a formal semantics in Isabelle/HOL, the same description can be used as the basis of the security analysis. It can also be used to debug, inspect and visualise the capability state of a running system.

For further assurance, we plan to formally verify the user-level library that translates the static capDL description into a sequence of seL4 system calls. Its main correctness theorem will be that after the sequence of calls has executed, the global capability distribution is the one specified in the original description. This will result in a system with a known, fully controlled capability distribution, formally verified at the C code level.

For system architectures that do not rely on known behaviour of trusted components, such as a classic, static separation kernel setup or guest OS virtualisation with complete separation, this will already provide a very strong security argument.

The tool above will automatically instantiate the low-level structure and access-control design into implementation-level C code. What is missing is providing the behaviour of each of the components in the system. Currently, components are implemented in C, and capDL is rich enough to provide a mapping between threads and the respective code segments that implement their behaviour. If the behaviour of any of these components needs to be trusted, this code needs to be verified — either formally, or otherwise to the required level of assurance.



IOSR = IOSpace Root Pointer
IRQR = IRQ register reference
DFRAMES = Device Frames

Fig. 2. Low-Level Design

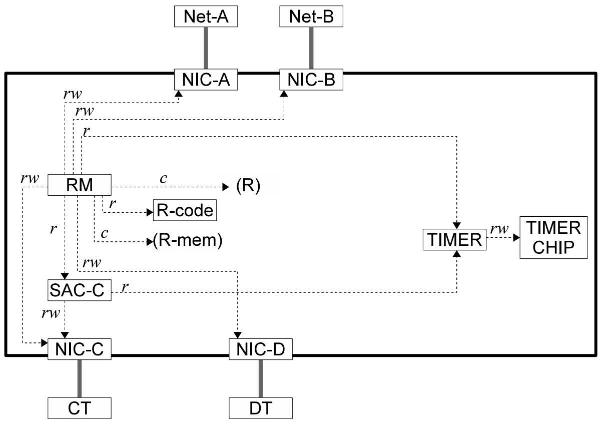


Fig. 3. SAC Abstraction

There is no reason component behaviour has to be described in C — higher-level languages such as Java or Haskell are being ported to seL4 and may well be better suited for providing assurance.

4 Security Analysis

Next to the conceptual security architecture of the SAC, we have at this stage of the exposition a low-level design mapping the architecture to the underlying platform (seL4), and an implementation in C. The implementation is running and the system seems to perform as expected. This section now explores how we can gain confidence that the SAC enforces its security property.

The capDL specification corresponding to Figure 2 is too detailed for this analysis. Instead, we would like to conduct the analysis on a more abstract level, closer to the architecture picture that we initially used to describe the SAC.

In previous work, we have investigated different high-level access control models of seL4 that abstract from the specifics of the kernel and reduce the system state to a graph where kernel objects are the nodes and capabilities are the edges, labelled with access rights [8,5]. We can draw a simple formal relationship between capDL specifications and such models, abstracting from seL4 capabilities into general access rights. We can further abstract by grouping multiple kernel objects together and computing the capability edges between these sets of objects as the union of the access rights between the elements of the sets. With suitable grouping of objects, this process results in Figure 3 for the SAC. The figure shows the initial system state after boot, the objects in parentheses (R) and (R-mem) are areas of memory which will later be turned into the main Router thread and its memory frames using the *create* operation, an abstraction of the seL4 system call that will create the underlying objects.

This picture now describes an abstract version of the design. We have currently not formally proved the connection between this model and the capDL specification, neither have we formally proved that the grouping of components is a correct abstraction, but it is reasonably clear that both are possible in principle.

For a formal security analysis, we first need to express the behaviour of RM in some way. In this case, we have chosen a small machine-like language with conditionals, jumps, and seL4 kernel calls as primitive operations. For all other components, we specify that at each system step, they may nondeterministically attempt any operation — it is the job of the kernel configured to the capability distribution in Figure 3 to prevent unwanted accesses.

To express the final information flow property, we choose a label-based security approach in this example and give each component an additional bit of state: it is set if the component potentially has had access to data from NIC A. It is easy to determine which effect each system operation has on this state bit. The property is then simple: in no execution of the system can this bit ever be set for NIC B.

Given the behaviour of the trusted component, the initial capability distribution, and the behaviour of the kernel, we can formally define the possible behaviours of the overall system and formally verify that the above property is true. This verification took a 3-4 weeks in Isabelle/HOL and less than a week to conduct in SPIN, although we had to further abstract and simplify the model to make it work in SPIN.

A more detailed description of this analysis has appeared elsewhere [1].

5 What is Missing?

With the analysis described so far, we do not yet have a high-assurance system. This section explores what would be needed to achieve one.

The main missing piece is to show that the behaviour we have described in a toy machine language for the security analysis is actually implemented by the 2,000 lines of C code of the Router Manager component. Most of these 2,000 lines are not security critical. They deal with setting up Linux instances, providing them with enough information and memory, keeping track of memory used etc. Getting them wrong will make the system unusable, because Linux will fail to boot, but it will not make it break the security property. The main critical parts are the possible sequence of seL4 kernel calls that the Router Manager generates to provide the Linux Router instance with the necessary capabilities to access network cards and memory. Classic refinement as we have used it to prove correctness of seL4 could be used to show correctness of the Router Manager.

Even with this done, there are a number of issues left that I have glossed over in the description so far. Some of these are:

– The SAC uses the unverified x86/IOMMU version of seL4, not the verified ARM version. Our kernel correctness proof would need to be ported first.
– We need to formally show that the security property is preserved by the existing refinement.

– We need to formally connect capDL and access control models. This includes extending the refinement chain of seL4 upwards to the levels of capDL and access control model.
– We need to formally prove that the grouping of components is a correct, security preserving abstraction.
– We need to formally prove that the user-level root task sets up the initial capability distribution correctly and according to the capDL specification of the system.
– We need to formally prove that the information flow abstraction used in the analysis is a faithful representation of what happens in the system. This is essentially an information flow analysis of the kernel: if we formalise in the analysis that a Read operation only transports data from A to B, we need to show that the kernel respects this and that there are no other channels in the system by which additional information may travel. The results of our correctness proof can potentially be used for this, but it goes beyond the properties we have proved so far.

6 Conclusion

We have demonstrated that formal code verification at a scale of about 10,000 lines of code is possible and feasible. We have argued that, for high-assurance systems, it is also cost-effective. There are no real barriers to conducting verifications like this routinely.

The bad news is that while these techniques may optimistically scale in the next few years up to 100,000s lines of code for nicely structured, appropriate code bases, realistic systems beyond that size still seem out of reach for the near future. Modern embedded systems frequently comprise millions of lines of code. None of these large systems are high-assurance systems yet, but a clear trend towards larger and more complex systems is observable even in this space, and some of these large systems, e.g. automobile code, should become high-assurance systems, because current practices are unsatisfactory [14].

Even though we may not be able to prove full functional correctness of such systems in the near future, our thesis is that it is nevertheless possible to provide formal, code-level proof of specific safety and security properties of systems in the millions of lines of code. We plan to achieve this by exploiting formally verified microkernel isolation properties, suitable security architectures, and code-level formal proofs for the small trusted computing base of such systems.

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.

References

1. June Andronick, David Greenaway, and Kevin Elphinstone. Towards proving security in the presence of large untrusted components. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 5th Workshop on Systems Software Verification*, Vancouver, Canada, October 2010. USENIX.
2. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.
4. Carolyn Boettcher, Rance DeLong, John Rushby, and Wilmar Sifre. The MILS component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Paul, MN, October 2008.
5. Andrew Boyton. A verified shared capability model. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 4th Workshop on Systems Software Verification*, volume 254 of *Electronic Notes in Computer Science*, pages 25–44, Aachen, Germany, October 2009. Elsevier.
6. David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182, Montreal, Canada, August 2008. Springer.
7. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer.
8. Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. Technical Report NRL-1474, NICTA, October 2007. Available from `http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf`.
9. Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conference Proceedings, 1979 National Computer Conference*, pages 329–334, New York, NY, USA, June 1979.
10. Bill 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.
11. Gerwin Klein. Correct OS kernel? proof? done! *USENIX ;login:*, 34(6):28–34, December 2009.
12. Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

13. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon 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.
14. Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 447–462, Oakland, CA, USA, May 2010.
15. Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Walker. capDL: A language for describing capability-based systems. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, New Delhi, India, August 2010. To appear.
16. Jia Meng, Lawrence C. Paulson, and Gerwin Klein. A termination checker for Isabelle Hoare logic. In Bernhard Beckert, editor, *Proceedings of the 4th International Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*, pages 104–118, Bremen, Germany, July 2007.
17. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
18. Raymond J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In David S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer–Verlag, 2010.
19. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, January 2007.
20. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.
21. Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius 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.
22. Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110, Toronto, Ontario, Canada, June 2010. ACM.