The L4.verified Project — Next Steps

Gerwin Klein

[1] NICTA[⋆], Australia

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

gerwin.klein@nicta.com.au

Abstract. Last year, the NICTA L4.verified project produced a formal machine-checked Isabelle/HOL proof that the C code of the seL4 OS microkernel correctly implements its abstract implementation. This papers gives a brief overview of the proof together with its main implications and assumptions, and paints a vision on how this verified kernel can be used for gaining assurance of overall system security on the code level for systems of a million lines of code or more.

1 L4.verified

Last year, we reported on the full formal verification of the seL4 microkernel from a high-level model down to very low-level C code [7].

To build a truly trustworthy system, one needs to start at the operating system (OS) and the most critical part of the OS is its kernel. The kernel is defined as the software that executes in the privileged mode of the hardware, meaning that there can be no protection from faults occurring in the kernel, and every single bug can potentially cause arbitrary damage. The kernel is a mandatory part of a system's *trusted computing base* (TCB)—the part of the system that can bypass security [11]. Minimising this TCB is the core concept behind *microkernels*, an idea that goes back 40 years.

A microkernel, as opposed to the more traditional monolithic design of contemporary mainstream OS kernels, is reduced to just the bare minimum of code wrapping hardware mechanisms and needing to run in privileged mode. All OS services are then implemented as normal programs, running entirely in (unprivileged) user mode, and therefore can potentially be excluded from the TCB. Previous implementations of microkernels resulted in communication overheads that made them unattractive compared to monolithic kernels. Modern design and implementation techniques have managed to reduce this overhead to very competitive limits.

A microkernel makes the trustworthiness problem more tractable. A well-designed high-performance microkernel, such as the various representatives of

[⋆] 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

the L4 microkernel family, consists of the order of 10,000 lines of code. We have demonstrated that with modern techniques and careful design, an OS microkernel is entirely within the realm of full formal verification.

The approach we used was interactive, machine-assisted and machine-checked proof. Specifically, we used the theorem prover Isabelle/HOL [10]. Formally, our correctness statement is classic refinement: all possible behaviours of the C implementation are already contained in the behaviours of the abstract specification. The C code of the seL4 kernel is directly and automatically translated into Isabelle/HOL. The correctness theorem connects our abstract Isabelle/HOL specification of kernel behaviour with the C code. The main assumptions of the proof are correctness of the C compiler and linker, assembly code, hardware, and boot code. The verification target was the ARM11 uniprocessor version of seL4. There also exists an 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 direct security properties that other OS kernels will find hard to claim. 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 [6,5].

2 A Secure System with Large Untrusted Components

There are at least two dimensions in which work on the seL4 microkernel could progress from this state: The first is gaining even more assurance, either by working on the assumptions of the proof, e.g. by using a verified compiler [9] or verifying the assembly code in the kernel, or by proving more properties about the kernel such as a general access control model [3,2]. The second dimension is using the kernel and its proof to build large high-assurance systems. Below I explore this second dimension and try to convey a vision of how large, realistic high-assurance systems can feasibly be built with code-level formal proof.

The key idea is the original microkernel idea that is also widely explored in the MILS (multiple independent levels of security and safety) space [1]: 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.

The basic process for building a system in this vision could be summarised as follows (not necessarily in this order):

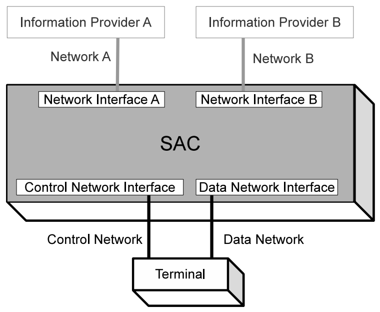

Fig. 1. Secure Access Controller (SAC)

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, report on some initial progress we had in modelling, designing, formally analysing, and implementing the system, and summarise the steps that are left to gain high assurance of overall system security.

The case study system is a secure access controller (SAC), depicted in Figure 1. It is a small box 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) and potentially hostile and collaborating. The property the SAC should enforce is that no information may flow through it between A and B. Information is allowed to flow from A to B through the trusted front-end terminal. The latter may not be a realistic assumption for a real system; the idea is merely to explore

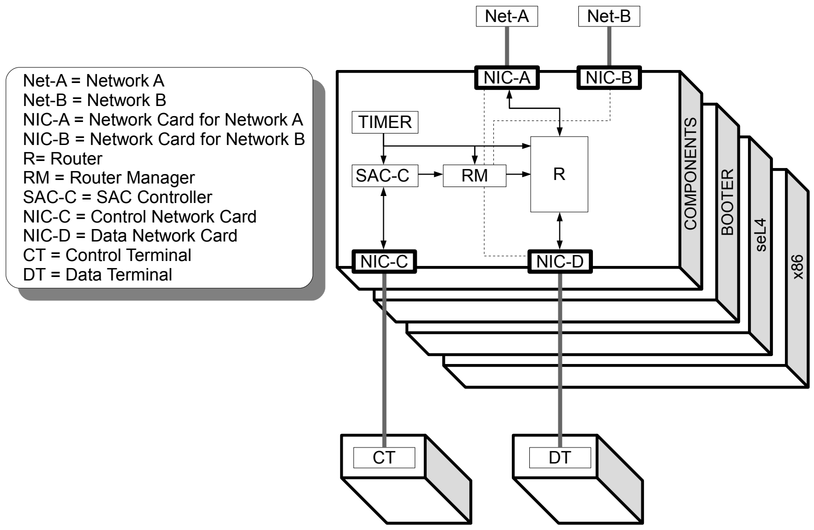

Fig. 2. SAC Architecture

system architectures for the SAC, not to build a multi-level secure product with a secure front-end terminal.

2.1 Architecture

Figure 2 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 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.

2.2 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 3 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 3 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 [8] (capability distribution language) that can be used to concisely describe capability and kernel object distributions such as Figure 3. 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.

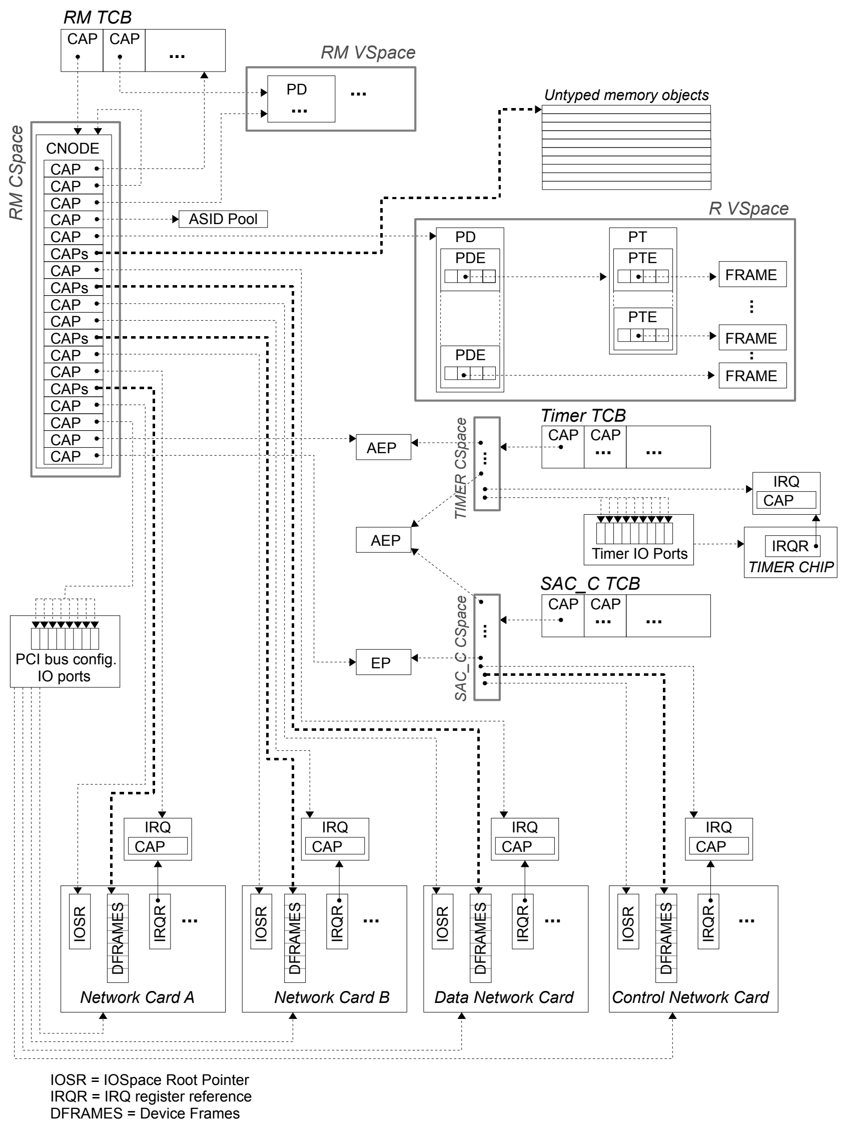

Fig. 3. Low-Level Design

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 described 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. 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.

3 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 explores how we can gain confidence that the SAC enforces its security property.

The capDL specification corresponding to Figure 3 is too detailed for this analysis. It contains information that is irrelevant for a security analysis, but is necessary to construct a running system. For instance, for security we need to know which components share virtual memory, but we do not necessarily need to know under which virtual address these shared areas are available to each component. 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 [3,2]. 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 4 for the SAC. The figure shows the initial system state after boot, the objects in

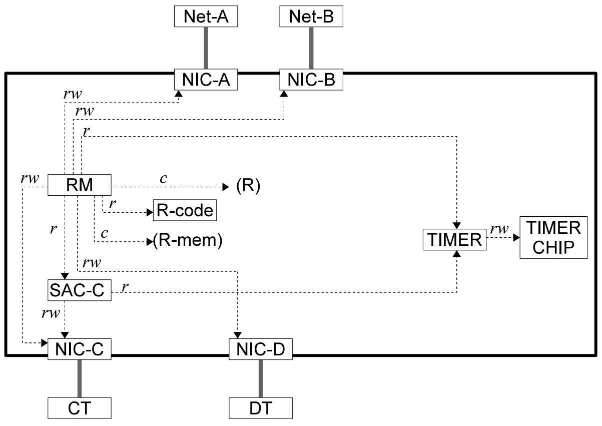

Fig. 4. SAC Abstraction

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.

The picture is simple enough to analyse. If we proceed with a simple information flow analysis based solely on the capabilities in Figure 4, we would have to conclude that the system is not secure: the component RM possess read/write capabilities to both network A and B and therefore, without further restriction, information may flow between A and B. Of course, RM is the trusted component in the architecture — specifically we trust that it will not transport information between A and B —, and the security analysis should take its behaviour into account.

Details on our experience with this analysis will appear elsewhere, below I only give a short summary.

For a formal analysis, we first need to formally 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. Any other formal language would be possible, as long as it has a formal semantics that can be interleaved with the rest of the system. 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 4 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. This state-based property is slightly weaker than a non-interference based approach [4], because it ignores indirect flows.

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.

4 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.

5 Conclusion

In this paper I have not aimed to present finished results, but instead to convey a vision of how one can use a formally verified kernel like seL4 to achieve code-level security proofs of large-scale systems. I have presented some initial, completed steps in this vision, and have shown that even if there is clearly still quite some way to go, there appears to be a feasible path to such theorems.

In an ideal world outcome, the complex proofs such as the information flow analysis based on a precise machine model, could be done once and for all for a given platform, and remaining proofs for specific systems and architectures could be largely or even fully automated: trusted components could be implemented in high-level languages with verified runtimes and compilers, abstractions for security analysis could be derived automatically with minimal user input and automatic correctness proofs, and the security analysis itself could be conducted fully automatically by model checking, potentially exporting proofs. This would mean such systems could be implemented with fairly low cost and extremely high assurance.

Even in a less ideal outcome, high levels of assurance could already be gained. Not all steps have to be justified by formal proof. Once trusted components and protection boundaries are clearly identified, the behaviour of a small trusted component could be assured by code review or testing, or abstractions for the security analysis could be done manually without proof. There is already value in merely following the process and only doing a high-level analysis. In our case study, we found security bugs mainly in the manual, but rigorous abstraction process from low-level design to high-level security model. What the proof provides is assurance that the analysis is complete, at the level of abstraction the theorem provides.

Acknowledgements The formal security model of the SAC was mostly created by June Andronick. The proof mentioned above was conducted almost entirely by David Greenaway with minor contributions from June Andronick, Xin Gao, and myself. The following people have contributed to the verification and/or design and implementation of seL4 (in alphabetical order): June Andronick, Timothy Bourke, Andrew Boyton, David Cock, Jeremy Dawson, Philip Derrin Dhammika Elkaduwe, Kevin Elphinstone, Kai Engelhardt, Gernot Heiser, Gerwin Klein, Rafal Kolanski, Jia Meng, Catherine Menon, Michael Norrish, Thomas Sewell, David Tsai, Harvey Tuch, and Simon Winwood.

References

1. C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Paul, MN, Oct 2008.
2. A. Boyton. A verified shared capability model. In G. Klein, R. Huuck, and B. Schlich, editors, *4th SSV*, volume 254 of *ENTCS*, pages 25–44, Aachen, Germany, Oct 2009. Elsevier.
3. D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. Technical Report NRL-1474, NICTA, Oct 2007. Available from `http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf`.
4. J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. Security & Privacy*, pages 11–20, Oakland, California, USA, Apr 1982. IEEE Computer Society.
5. G. Klein. Correct OS kernel? proof? done! *USENIX ;login:*, 34(6):28–34, Dec 2009.
6. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, 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. *CACM*, 53(6):107–115, Jun 2010.
7. 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 *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.
8. I. Kuz, G. Klein, C. Lewis, and A. Walker. capDL: A language for describing capability-based systems. In *1st APSys*, New Delhi, India, Aug 2010. To appear.
9. X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd POPL*, pages 42–54, New York, NY, USA, 2006. ACM.
10. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
11. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63:1278–1308, 1975.