Provable Security: How feasible is it?

Gerwin Klein, Toby Murray, Peter Gammie, Thomas Sewell, Simon Winwood
NICTA and University of New South Wales, Sydney, Australia
firstname.lastname@nicta.com.au

Abstract

Strong, machine-checked security proofs of operating systems have been in the too hard basket long enough.

They will still be too hard for large mainstream operating systems, but even for systems designed from the ground up for security they have been counted as infeasible. There are high-level formal models, nice security properties, ways of architecting and engineering secure systems, but no implementation level proofs yet, not even with the recent verification of the seL4 microkernel.

This needs to change.

1 Introduction

For mainstream operating systems the community has given up on strong security. We can graft mandatory access control onto Linux, or try to harden Windows NT, but we are in no position to get to provable or even just assurable security of such OSes in the next 20 years.

We do have a number of high-level security models that work with nice, provable properties. There are ways to build and structure secure systems. We know that systems can be secured much better by reducing the trusted computing base (TCB). Microkernels with access control and separation kernels provide solid foundations for implementing such models. Some such systems have been built, demonstrated, and are deployed. But even with their reduced TCBs nobody has yet proved security down to the implementation level. They also have a legacy problem—users, even those with high security requirements, like to run the software they know, and such software usually expects to run on a mainstream OS.

Type 1 hypervisors have been proposed to deal with the legacy problem and to assume the role of separation kernels, but there are no implementation-level proofs for these yet either.

On the other hand there has been progress on the proof side for small kernels, such as our functional correctness verification of the seL4 kernel [10]. Extrapolating to the future, it is certainly possible to reduce the effort for this kind of verification. Let's assume that in the next 10 years there will be more and more small kernels with formally verified functional correctness down to the implementation. Such proofs do not automatically imply security [9]; we will revisit why precisely in Section 2. However, with these proofs available, we claim it is now feasible to reach one of the holy grails of security: We can now prove that not only kernels, but whole system implementations have strong security properties.

Feasible does not mean easy. There is a clash between the precise specification of a real system, and idealised, simplified, high-level security properties in the literature. Section 3 examines some of these challenges and our experience proving access control enforcement for seL4 [12]. Similar challenges are to be expected for other systems.

Assuming we can prove security properties of kernels, we are still not done yet. What we ultimately want to achieve is proof that whole systems enforce their security goals. Big legacy systems and big code bases may still be a problem, but there are ideas for solutions. We will still not get absolute security in the end (sorry, we never will), but we can get much stronger assurance for much larger systems than what is thought feasible today. In Section 4 we sketch how we plan to do so.

Our message is that security implementation proofs are doable and should be done. They will not make Linux secure, but in areas where we do care about security, they will be able to give us systems we can trust, that are usable, and that are not horribly expensive.

2 Functional Correctness

In this section, we revisit what functional correctness is precisely, why it does not necessarily imply security, and why it makes it feasible to prove security properties on the code level.

Functional correctness reasons about a specification and an implementation. It asserts that the implementation correctly implements the specification. For a microkernel, the implementation level should be the code, e.g. C or assembly. The specification is an abstract model of the kernel that is smaller, easier to understand, more concise, and written in a high-level language such as a rich formal logic.

The main implication of functional correctness is that in order to find out if a property is true about the kernel, one does not need to look at its complex, optimised implementation anymore, but instead can reason about the smaller specification. This does not work for all properties, but it does work for the large class of properties preserved by formal refinement, including everything that can be expressed by Hoare triples such as invariants about the states of the system.

For kernels written in C, functional correctness gives us additional beneficial side effects, because kernel execution must always be well defined. This eliminates the usual problems that plague C programs such as null pointer dereferences and buffer overflows. We do not need to consider low level code issues any more, and can instead focus on API and system security.

Proving functional correctness is the right first step and the basis for proving further high-level properties. It is also the point where language becomes confusing. For instance, the seL4 kernel is now verified, and it is tempting to assume it will never exhibit any unwanted behaviour if the proof assumptions hold. However, the proof says only that the code will follow the specification; it does not say that the specification enforces a specific security property (apart from those above that we get as free side effects).

What we have gained for security, apart from a precise specification and a proof that the kernel implements it, is the ability to state specific security properties concisely and prove them with much less effort. The reduction in effort is substantial. It is similar to working from a well defined API instead of re-implementing functionality from scratch each time. We estimate that proving e.g. integrity enforcement in seL4 would have taken us at least the same 25 person years that the functional seL4 verification did. Now, it took less than 10 person months [12]. Further high-level properties will be similar. The same is true for proving anything about user-level applications that make use of kernel calls.

3 Access Control: Ideal and Real

In this section we show some of the challenges one should expect when proving high-level properties about real kernel APIs. As we shall see, there is a significant gap between traditional, idealised security properties and the complexities of real APIs. There is scope for developing realistic security properties that scale to real systems. However, our experience also indicates that proving security of kernel implementations is feasible.

We use as an example the proof [12] of a generic, high-level access control theorem about the seL4 kernel specification, and thereby the seL4 code. We believe our experience in this is not specific to seL4, but intrinsic to showing security about real implementations.

Correct access control is one of the most basic security properties any OS kernel should enforce, and it is the one that should be proved first, because it provides the basis for more complex, system-wide security properties as well as one of the main formal proof tools for composing user-level systems on top of the kernel.

Access control proofs have been completed before for various kernels [4, 13]. These proofs have analysed idealised kernel models removed in various degrees from the messy details that necessarily exist in practical kernel APIs. The focus of this section is how the story changes when the kernel model is rooted in reality by proof.

The seL4 kernel was designed to implement a capability-based [5] access control system, so we set out to prove that it does. The high-level enforcement property for access control in capability systems is simple and well known:

Authority. Each subject may read or affect the state of an object only if it can present a capability with the appropriate access rights to that object.

Propagation. Subjects may not inappropriately obtain new authority. In particular, one subject may share its capabilities with another only if that first subject transitively possesses Grant authority to the second, or if the two subjects share capability storage.

To prove that this is true, we first must define what each capability authorises, e.g. exactly which parts of which objects may be affected, and then show that only these allowed effects happen. This is also what developers intuitively reason about when they construct systems.

In ideal systems, this definition is obvious and easy to understand: a capability points to an object, it authorises only operations that can affect this object, and it does so only if the capability provides write access.

In real implementations, this definition is not always obvious. For instance, IPC in seL4 involves multiple objects. A thread sends to an endpoint, another thread listens on the same endpoint. To do so, the first thread presents an endpoint send capability, the second an endpoint receive capability. A simple enough mechanism, but it already requires us to say that the send capability authorises changes in four different objects: a) the endpoint, b) the thread state of the sender, c) the thread

state of the receiver, and d) the IPC buffer of the receiver which may be located in a frame. This is still not a problem: we generalise to a set of objects instead of one, and it may take some lookup operations to say exactly which ones. It can get more subtle: a frame capability in seL4 provides not only the authority to read and write to the frame according to its access rights, but it also confers the authority to unmap the frame from its current address space, for instance when the capability is being deleted as part of a larger revoke operation. Again, this is no conceptual problem, but now we are talking about a potentially large set of objects already (a whole address space), even though only very limited change is permitted (unmapping a particular frame). Clearly, the definition is no longer as clear-cut and intuitive as in the literature.

The normal reaction to such complications in high-level models is usually to say: without loss of generality, we can assume that the API can be simplified to another API that achieves the same effect, such as sending a message. In this new API there is now only one object affected again, keeping the definition of security simple. This is perfectly reasonable when the goal is to develop a security model and to present it.

If the model is rooted in the code, however, these kinds of simplifications do not work anymore. The intention is not to prove something about another, simpler API that has nice properties, but about the API that the code implements. You can prove further abstractions, but they have to be true abstractions of the API. You can also prove that some behaviour is equivalent to another formulation that may be nicer, but you cannot leave out a behaviour that the code has, just because it introduces complexities into the high-level property.

It is of course also possible to change the real API and thereby truly simplify the model, but usually there is a reason the API has this complication in the first place. In our example, the seL4 API is the product of many engineering, proof, and design trade-offs. As other kernels do, it contains necessary corner cases and subtle exceptions to an idealised notion of access control. Both complications mentioned above were known beforehand to the design and verification teams and were deemed worth the cost in the trade-off with other options.

In short, there is no place to hide: either simplify the real API or deal with more complexity in higher-level notions such as access control. This is not a fundamentally new trade-off, but it necessarily keeps both the designers and verifiers honest about the complexity in their system.

Even simply defining the high-level property and looking through the kernel API specification to see if it is enforced will throw up these issues. We did this for seL4, starting with the simple notion of access control from the literature and either refined and generalised it or changed the kernel as we discovered exceptions to it in the API. Sometimes the change can be an actual security fix (a right that should have been checked and was not), sometimes it will be an API simplification that makes things more consistent.

In our case, two API simplifications happened in places that were flagged as too complex already, but where looking through the specification uncovered violations of the authority and propagation properties above. These were true security violations, not just added complexity to the notion of security. One was a privilege escalation from read-only to read-write thread capabilities, the other was a state change authorised by a nominally read-only capability. The root cause of both cases was API evolution: the kernel designers initially envisioned systems where these access rights made sense, but later saw better ways to construct secure systems that did not need them. In systems constructed the new way, these read-only capabilities would never occur, and so less attention was spent on their consistent enforcement. The issues were not spotted earlier, because the rights had no impact in the new way of using the API. However, developers who had not read the specification carefully might assume a security mechanism that was not enforced. It would have been easy to add additional code checks to enforce the access rights consistently, but instead we decided to simplify the API to reflect the intended use case. Adapting the functional correctness proof to these API changes required little effort, less than one person week.

In summary, there is a significant gap between idealised security notions and the refinements and generalisations that are needed when they are applied to real kernel models. If more groups start reasoning about other kernel models, connected to reality by proof or other mechanisms, we will see increased demand for security definitions that scale to realistic features, but that are still usable for higher-level reasoning.

Our seL4 work has produced one such generalisation for access control, but the design space for such properties is large and we will certainly not have explored it fully. There is an opportunity for the whole field to make security properties more real, both in terms of mechanisms as well as formalisations. This does not require big functional correctness proofs. It only requires precise models of real APIs and the willingness not to simplify them for convenience, but only under realistic trade-offs when the code is changed in tandem.

4 Whole System Security

The previous section argued that while there is work to be done, we can soon expect generic code level security properties to be proved about small OS kernels. We now examine the prospects of proving security properties of whole systems. Again there is much work to be done,

but we think that this goal is now within reach.

How far can we push these properties? There is rich literature on security, and it is to be expected that the more mature, well developed properties lend themselves better to application.

Apart from access control, users are usually interested in information flow properties like secrecy and integrity. Label based security models that are close to access control models can be used to reason about both, and we expect a similar need for these to be refined and generalised when applied to real kernels, but that proving them for real implementations will still be feasible.

Noninterference [6], as a generic property between isolated subsystems, is a stronger information flow property, because it can encompass all storage channels observable in the kernel model. However, it has some specific technical challenges. In particular, many common noninterference properties are too weak, because they can hold for a specification but not for a functionally correct implementation of it [8], while others that require every functionally correct implementation to be secure are often too strong, when the insecure implementations never arise in practise [11]. This means that more work has to be invested, and more care taken, to specify and prove noninterference. We have made some initial headway in formulating it for seL4 and think it will still be feasible to prove at the code-level in controlled settings.

Building on information flow models, we can get to a generic separation kernel or hypervisor property for running isolated, untrusted guests side-by-side on the kernel. This is less general than the other properties above, but it is a popular and well defined use case. If we can prove noninterference, these others should follow easily.

What properties will still be too hard? We do not yet see a good formal handle on timing and time based covert channels for real implementations. While there are many analysis and mitigation techniques, we do not expect to be able to obtain the same level of formal proof for these in the near future as for storage channels.

In particular for information flow properties it will become much more relevant how detailed the base level machine model is. For functional correctness one can reason with fairly high-level machine abstractions. If the intention is to prove absence of storage channels via caches, buses, or device interactions, machine models would need to become much more concrete, and thereby larger and more complex. While some examples exist [3], we do not expect this to happen soon on a larger scale. It is more likely that we will have to live with hardware models that are too abstract to capture all such effects. Since a noninterference proof for instance will only be up to effects visible in the model, we will still need traditional assurance and risk mitigation techniques to address such channels.

Building Systems Assuming we have proved security properties about our kernel, what are the challenges in building and proving secure systems with them?

The kind of systems we expect to prove security properties about are systems in the spirit of MILS architectures [1]: security is built into the architecture of the system, component boundaries are enforced by the kernel, communication happens only via clearly defined channels, most components are untrusted, and only a few small trusted components exist.

The trusted components are those that have the authority to subvert the security policy—we will need to prove down to the code level of each such component that this authority is not misused. Untrusted components in contrast will be assumed to do anything in their power to subvert the system. We constrain their power via kernel mechanisms such as access control and can rely on this mechanism because we have proven it correct before. This means we do not have to prove anything about the code of untrusted components which makes it possible for them to consist of large legacy code bases, such as a whole Linux guest OS.

One immediate question is what kind of useful and usable systems can be built in this fashion. We have published elsewhere a small case study on an seL4-based network router [2] that we have built in this style and of which we have analysed its security on a high-level. Our experience indicates that such devices with a clear purpose and a clear security goal can be readily architected with a very small TCB (less than 1,500 lines of code apart from the kernel), and at the same time contain legacy code on the order of a million lines of code or more. Hardware support like Intel's IO MMUs make it possible to even remove devices and user-level drivers from the TCB. Another promising application area are multilevel secure terminals. With a small TCB, these could provide a full desktop experience and still have strong isolation properties. In fact, a whole new design space opens up if the kernel is actually trustworthy [7].

The high-level security analysis of such systems proceeds by formally describing the security architecture, the behaviour of trusted components, and the security goal. Small systems such as the network router example can be analysed automatically on this level [2], in seL4 mainly by observing the possible flow of capabilities in the system. The challenge is to connect this analysis to the code level such that it remains valid.

The first technical challenge is that, while a system may be small on a conceptual whiteboard level with tens of objects, the implementation will already contain thousands of objects to reason about, even before large legacy

components have started up. After legacy components are started, our network router example contains tens of thousands of objects. For the purposes of the analysis, it should be possible to aggregate large sets of these objects into one object without discarding their security impact. While reasoning about large numbers of objects presents challenges to theorem proving tools, we think the process of aggregation and proving that the aggregation is correct could be mostly, if not fully, automated.

The second challenge is to prove that the code of trusted components indeed implements the behaviour that was assumed in the security analysis. This is similar to code verification for kernels, but reasons from the user's point of view, rather than the kernel's, and must deal with concurrency in the system. This verification step will likely be conducted manually.

The third challenge is to provide a correct initial policy setup for the kernel. This setup needs to provably enforce the architecture that is the basis for security. We think this step can be fully automated, based on a textual policy specification. A code generator can produce the program or data structure that constructs the initial setup and it can also produce a proof that the resulting initial system state corresponds to the policy specification.

Finally, we will need to compose the parts and their proofs into a full system such that the proofs fit together. The correctness theorems of the kernel and its security mechanisms are the formal basis for this composition.

All of these taken together mean that, while there are still a number of challenges to overcome, it is feasible to address whole system security on the code level with formal proof. If the tools are constructed and the generic proofs done, the verification effort for producing a new system in this style can be largely automated. Ideally, only the behaviour of the trusted components needs to be verified manually and even that effort could be reduced if higher-level programming languages and runtime systems can be employed.

5 Conclusion

There is a gap between high-level idealised security properties and the properties that hold of real kernels. Since we now know that we can produce kernel models that are connected to code by proof, we can and should make statements about abstract security properties on real systems, all the way down to the code level.

The value of such proofs is not just as a tool in constructing secure systems. It is to know when you are done, and to be able to provide evidence to others that you have achieved the goal.

Proof will only give us strong assurance up to the detail visible in the base level model, but this is a huge step forward over what we have now. Traditional techniques and risk mitigation for covert channels can then be employed in a much more focused way and can be informed by the precise assumptions made in the proof.

For truly security-critical systems we should start demanding full proofs of security, down to the implementation level. These proofs are not only necessary but are now feasible at reasonable cost.

Acknowledgements

We thank June Andronick, Kevin Elphinstone, David Greenaway, and Gernot Heiser for valuable feedback on drafts of this paper.

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.

References

[1] J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.

[2] 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, *5th SSV*, Vancouver, Canada, Oct 2010. USENIX.

[3] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together—formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):411–430, 2006.

[4] 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.

[5] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[6] 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.

[7] G. Heiser, L. Ryzhyk, M. von Tessin, and A. Budzynowski. What if you could actually *Trust* your kernel? In *13th HotOS*, Napa, CA, USA, May 2011.

[8] J. Jacob. On the derivation of secure components. In *IEEE Symp. Security & Privacy*, pages 242–247, 1989.

[9] G. Klein. Correct OS kernel? proof? done! *USENIX ;login:*, 34(6):28–34, Dec 2009.

[10] 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.

[11] T. Murray and G. Lowe. On refinement-closed security properties and nondeterministic compositions. In *8th AVoCS*, volume 250 of *ENTCS*, pages 49–68, Glasgow, UK, 2009.

[12] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. sel4 enforces integrity, 2011. Submitted.

[13] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *IEEE Symp. Security & Privacy*, pages 166–181, Washington, DC, USA, May 2000.