Formal Methods in the Robin project: Specification and verification of the Nova microhypervisor

Hendrik Tews*

Radboud Universiteit Nijmegen, The Netherlands

http://www.cs.ru.nl/∼tews

June 5, 2007

The objective of the Robin project is to develop an open robust computing infrastructure. The Nova micro hypervisor is currently being developed as a basis for this robust infrastructure. One workpackage of Robin concentrates on the application of formal methods to this newly developed micro hypervisor. The goals within Robin are (1) to verify some properties of a selected hypervisor module and (2) to develop a formal specification for the hypervisor interface.

This paper presents our approach for the verification of Nova. I will especially discuss the challenges and some solutions of operating-system kernel verification.

1 Introduction

The aim of the Robin project is to solve the following sort of dilemma: PDA's are used for browsing the world-wide-web *and* for storing private data. For web-browsing one wants to install latest software however, this opens the door wide for every attacker.

The solution that is currently developed in the Robin project is depicted in Figure 1. It is usually called the *Nizza security architecture*. A micro-hypervisor exploits the virtualization support in the new x86 CPU's to provide a basis for running dedicated applications and possibly multiple copies of legacy operating systems (like Linux and Windows). Pieces of software can be completely encapsulated such that they can only communicate to the outside through special channels. With such an architecture one can

*The author has been supported by the European Union through PASR grant 104600.

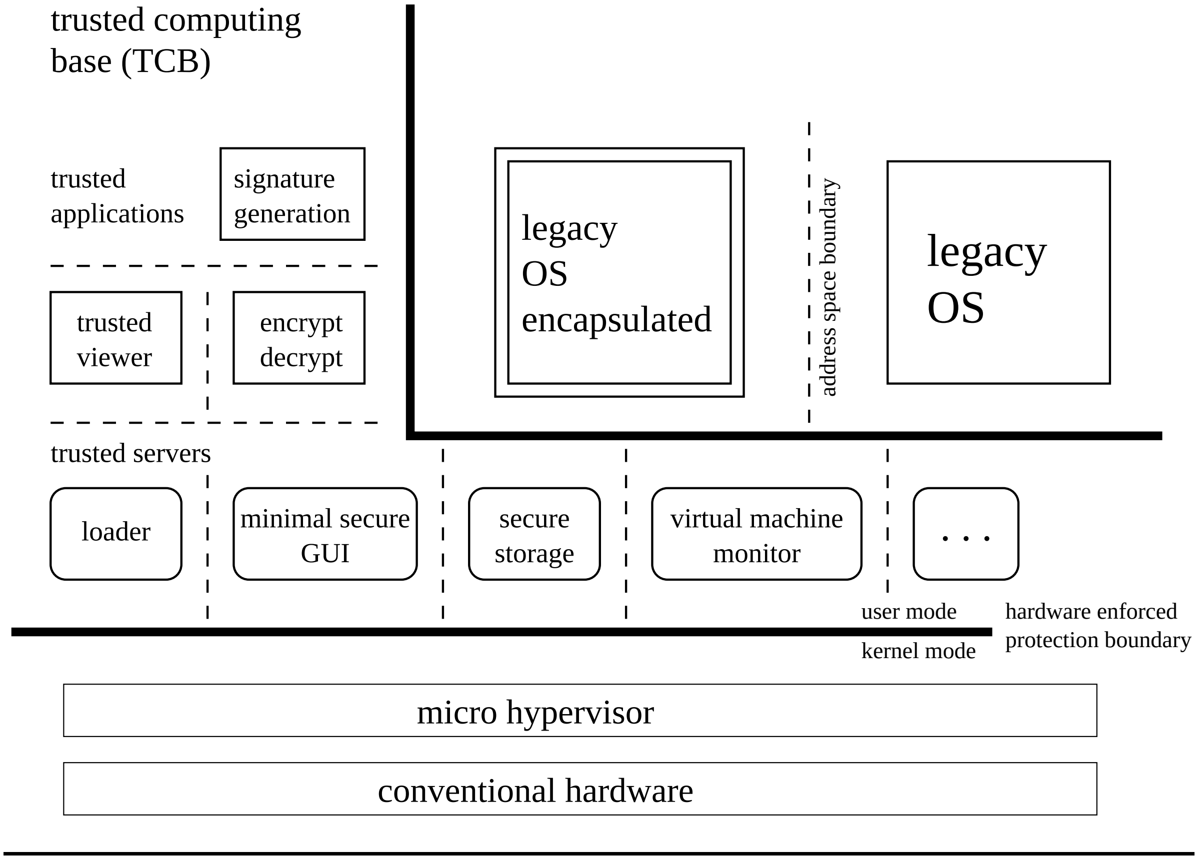


Figure 1: Nizza security architecture

browse the world-wide-web in one operating system (OS) instance, while, at the same time, one also writes a classified document in a different OS instance. The hypervisor makes sure that the browser OS cannot see the classified document, even if both OS instances have been taken over by an attacker. The Encapsulation ensures that the classified document can only be sent through one channel to an encryption module that runs directly on the hypervisor. Even if the installation media for the legacy OS was compromised, an attacker cannot get access to the classified document. For more information on the Nizza architecture see [Tew07, HHF$^+$05, FH05, FH03, HPHS04, Fes06].

The goal of work-package 4 of the Robin project is to develop a verification approach such that some properties of one selected hypervisor module can be mechanically proved. Further we plan to develop a formal specification of the hypervisor interface. For the specification the main challenges at the moment are non-technical. Our aim is to make the formal specification part of the hypervisor documentation. For that it must be based on simple set theory and the public parts of the specification must be free of fancy symbols such as $\in, \forall, \subseteq$.

In the remainder of the paper I concentrate on the verification goal. Section 2 describes the challenges, Section 3 the verification approach and Section 4 some goals of the verification.

2 Challenges of low-level system-software verification

A verification of an OS kernel, even if it is only a small hypervisor is a very challenging project for a number of reasons.

C++ source code Currently, there seems to be no convincing alternative to C/C++ for kernel programming, at least from the point of view of many OS-groups. Consequently the Robin hypervisor is written in C++. C++ programs are difficult to formalise for a number of reasons:

- The C++ standard [Int98] is relatively vague in order to permit conforming C++ implementations on the weirdest platforms. For instance the signed integral types are not required to contain negative numbers. Further, casts between different pointer types might change the pointer (to satisfy alignment requirements), except for the case where one casts to `void *` and back to the original pointer.

  Because of the vagueness of the C++ standard almost every program relies in some way on platform or compiler specific properties. Consequently, a formalisation of C++ program must incorporate some properties of the specific C++ implementation that is used to compile the program.

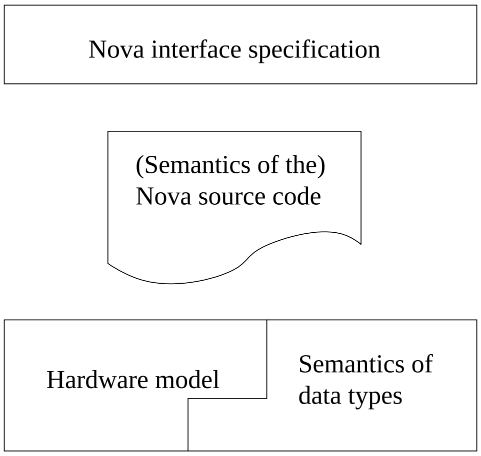
- The template mechanism of C++ alone is Turing complete [Vel]. This means the compiler can be forced to do arbitrary computations *at compile time*. A formalisations of C++ templates is accordingly difficult.

  The micro hypervisor will only use few templates. If they are getting too difficult we will work with the template instantiations instead.

- Type casts and `goto`-jumps are features that are traditionally not handled in textbooks on program semantics. However, it is impossible to write a micro hypervisor without typecasts and to avoid unduly performance penalties one needs some kind of unstructured jump such as `setjmp/longjmp` [lon] at a few places.

Embedded assembly code and direct hardware manipulations For operations that are not supported in C++ (mostly direct hardware manipulations) the hypervisor sources will contain some assembly code, mostly in the form of inlined assembly. Assembly code is needed at least for the following operations:

- Access to hardware registers, such as those from the APIC (Advanced Programmable Interrupt Controller), but also special CPU registers, such as CR3 (page directory base register), EFLAGS (the flags register), the global descriptor table, the interrupt descriptor table, the task segment register and the feature control registers CR0 and CR4.



The hardware model and the semantics of data types provide the basic operations and properties for the verification of the hypervisor. For hardware data types the hardware model relies on the semantics of data types. Technically

$$\Phi_{data\_types},\quad \Phi_{hardware} \ \vdash\ \varphi(\text{hypervisor})$$

where φ is one property from the hypervisor specification, such as *termination without runtime type errors*.

Figure 2: Robin verification approach.

- Embedding special instructions in the code, such as IRET (return from interrupt) and INVLPG (invalidate a TLB entry).
- Manipulating the stack frame to access and modify parameters of system calls or for programming non-local exists similar to `longjmp`.

Nonstandard program environment The hypervisor runs like usual programs in virtual memory. However, the hypervisor manipulates the virtual memory mapping itself. Some parts of the memory are visible multiple times at different virtual addresses. One can therefore have very subtle aliasing: A variable x at address a_1 can be changed by writing to the totally different address a_2.

The hardware manipulations that the hypervisor must perform bear the possibility of subtle errors. Certain bits in hardware data structures, such as the page directory entries, must be zero. A more subtle problem comes with the translation look-aside buffer (TLB). The TLB is a special kind of cache that caches page-directory traversals. As a cache the TLB is not transparent, which means, when changing a page-directory or page-table entry one must manually invalidate the TLB before using the new address mapping. Otherwise, depending on the execution history, the old mapping, still cached in the TLB, might be used.

3 A verification approach for a Micro Hypervisor

In this section I explain the approach that we are planning to use in the Robin project. The approach is depicted in Figure 2, it has already been worked out in the VFiasco project [HT05, HT]. Our approach heavily relies on the interactive theorem prover PVS [ORR+96]. The input language of PVS is higher-order logic enriched with predicate subtyping and some other forms of dependent types. For the verification we model the x86 hardware and the hypervisor inside PVS and use later the prover component of PVS to establish theorems about it.

Our verification approach uses source code verification. That is, we translate the C++ source code into a set of specific functions that are defined in the PVS input language. Source code verification also means that we do not directly verify the object code that will really be running. However, source code verification lets us profit from the relatively high abstraction level present in the source code (which is lost in object code). A connection to the real object code is vaguely planned for the far future.

In our approach the translation of the C++ code into PVS depends on the hardware model and the semantics of data types. Both, the hardware model and the semantics of data types are PVS specifications that are currently developed. As expected the semantics of data types deals with C++ data types in PVS. Our semantics of C++ data types exploits underspecification to make it possible to detect erroneous type casts and wrong implicit type conversions (like, for instance, reading data from a union with the wrong type), see [HT03].

The hardware model formalises an abstract model of the x86 hardware inside PVS. It provides physical memory, virtual memory with address translation via page directories, some kind of TLB and much more. The hardware model does not blindly model the real hardware. Instead the hardware is modelled in such a way that certain subtle programming errors yield a specific error state instead of doing nonsense (like the real CPU). For instance the attempt to interpret a string as a page directory entry yields an abnormal result value. This kind of error checking works even for the hardware initiated page directory traversals done during address translation.

In order to translate C++ into PVS we use a denotational semantics for (a subset of) C++. This denotational semantics has been developed partly already in the VFiasco project. It correctly treats type casts, goto jumps and all the other complications that I pointed out in the preceding section. One can view the hardware model and the semantics of data types as providing the basic building blocks of our denotational C++ semantics. In our design the three components (C++ semantics, hardware model and data types) are relatively independent from each other. It is therefore possible

- to add additional axioms to the data types, for instance, to model a compiler specific assumption about the size of some data types or the precise behaviour of some type casts.
- to add new operations to the hardware model
- to use different versions of the hardware model for different pieces of the hypervisor. The boot code of the hypervisor can be verified against physical memory and the hardware independent parts can be verified against a traditional, untyped memory model.
- to adopt the semantics for new C++ features or compiler specific C++ constructs.

Figure 3 depicts the data flow of our verification approach. A semantics compiler translates the C++ source code into its semantics in higher-order logic and outputs this

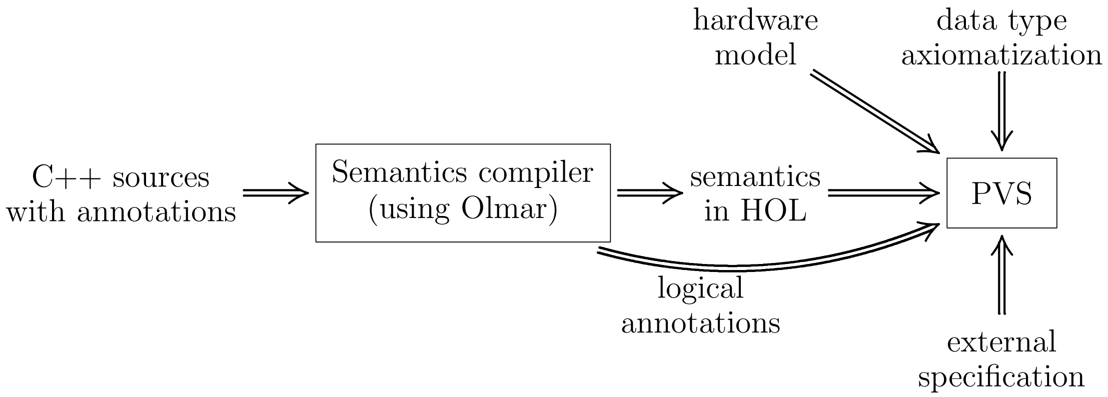


Figure 3: Approach for source code verification

semantics as PVS source code. There will be two kinds of annotations in the source code. The first kind influences the syntactic form of the output. It will for instance be possible to place the semantics of a block or a group of statements into a separate function, in order to make it possible to modularise the verification. The second kind of annotations contains specifications for the code similar to JML [BCC$^+$05]. The semantics compiler translates these specifications into PVS proof obligations.

The hardware model and the data type axiomatisation are directly developed in PVS. From the PVS point of view they provide declarations for all the function that appear in the output of the semantics compiler. With all the different pieces loaded in PVS one can start to prove hand-written, external specifications of the hypervisor.

The semantics compiler translates the sources of the program into semantic functions that precisely model the behaviour of the original source code. A semantic function is a state transformer of the following form[1]

$$\mathit{State} \quad\longrightarrow\quad \overset{\mathit{ok}:}{\mathit{State}} \uplus \overset{\mathit{pagefault}:}{\mathit{State} \times \mathit{Page\_fault\_info}} \uplus \overset{\mathit{hang}:}{\mathbf{1}} \uplus \overset{\mathit{fatal}:}{\mathbf{1}} \uplus \cdots$$

Here *State* is a set of machine states provided by the hardware model. Every state contains the contents of the physical memory and the contents of some important control registers (such as virtual address mapping or the stack pointer). The disjoint union on the right hand side describes the possible results of a state transformer. If no abnormal condition occurs it yields a successor state tagged with *ok*. If a page fault occurs it yields a successor state plus some additional information. A result tagged with *hang* means that the program did not terminate (for instance because of a nonterminating while loop or a page fault that keeps occurring at the same instruction). The result *fatal* is reserved for serious errors such as TLB inconsistency or reserved bit violations.

[1]I use $\uplus$ for disjoint union. The notion $\overset{\mathit{ok}:}{\mathit{State}}$ provides meaningful names for the injections. $\mathbf{1}$ is the unit or one-element set.

State transformers can be composed in the obvious way: If the first state transformer yields a result tagged with *ok* the result state is passed into the second state transformer. If any abnormal conditions occurs the second state transformer is skipped and the result of the composition is the abnormal result of the first state transformer.

The hardware model provides the basic state transformers for reading to and writing from memory and for reading and writing the control registers. The semantics compiler composes the basic state transformers from the hardware model to build the semantics of its input program.

Program verification proceeds by reasoning in PVS over a nontrivial state transformer that represents the semantics of some source code. This is mostly done by applying a start state to the state transformer and proving properties of the result (for instance that the result *is not* tagged *fatal*). Such a verification could equivalently be performed by computing the weakest precondition of the verification goal with respect to the program.

A slightly different view on the verification is as follows: The hardware model defines a state machine. The basic state transformers of the hardware model describe the actions of the state machine. The program is symbolically executed on top of the state machine. Properties are derived from the state changes that one observes.

4 Verification goals for the Robin Micro Hypervisor

The preceding section made very clear that the precision of the verification hinges on the hardware model. With precision of the verification I refer to the amount and kind of errors whose absence is proved with a successful verification.

It is our aim to make the hardware model precise enough to let it catch the following kinds of errors:

- *Common errors*, such as dereferencing a null-pointer, nonterminating loops, wrong results (wrt. the functional specification).
- Type errors. A type error occurs when one attempts to read an instance of some type from a memory location where nothing or an instance of a different type has been stored. The hardware model will be precise enough to catch type errors for user code (for instance resulting from wrong pointer or address calculations) *and* for implicit hardware memory accesses (for instance reading page directories)
- Virtual-memory aliasing errors. Virtual-memory aliasing occurs when the virtual memory of two distinct variables is mapped to the same (or overlapping) physical memory region. A virtual-memory aliasing error happens if one has virtual-memory aliasing for two variables that are used at the same time.
- TLB errors (accessing a linear address[2] for which the page directory or page table entry might be inconsistent with the translation look aside buffer)

[2] On the IA32 architecture virtual addresses, which are page-wise mapped to physical addresses, are called linear addresses. Virtual addresses in the sense of IA32 (i.e., the addresses that appear in the object code) are first subject to an address translation defined by the segment registers. This

- allocation errors (using the same memory for different variables at the same time)

Within Robin we are not targeting the following errors:

- any kind of hardware error
- errors that can only occur on systems with more than one logical processor (i.e., on systems with multiple CPUs or with active hyper-threading)

At the moment we have several candidates of proof obligations that we would like to verify for the hypervisor in the future.

Normal termination The hardware model contains a lot of checks (like for instance reserved bit conditions and TLB consistency) that enter an abnormal state if they are not fulfilled. In order to prove that the hypervisor does not contain this kind of errors it is therefore sufficient to prove normal termination.

Dynamic type correctness Because of the type casts and the internal memory management the type correctness of the hypervisor cannot be checked with a type system. Instead, type correctness has to be established during verification. The semantics of data types relies on underspecified functions for reading and writing data from and to memory. Axioms for these functions just cover the case of reading some data type from a memory location where data of the same data type has been written before. In case of a type error, where one tries to read from a memory location that contains garbage or data of a different type, non of these axioms apply. One can thus proof nothing, not even normal termination, about a program with a type error. In order to prove type correctness it is sufficient to prove normal termination of the hypervisor code. The consistency of the axioms will be shown using refinement of theories in PVS [OS01]. For more details see [HT03].

Only kernel code runs in kernel mode One of the most terrible programming errors of an operating system is to execute arbitrary user level code with kernel mode privileges. This can happen if the hypervisor forgets to reset the privilege level on return to user code. It can also happen if the user manages to exploit a buffer overflow inside the kernel. In order to prove that only kernel code runs in the highest privilege level one has to prove that nobody tampers with the return addresses on the stack and that all control path that leave the kernel reset the privilege level in the right way.

With security applications in mind it would also be very interesting to prove the following.

Address space separation If one address space (read process) has access to memory of another address space, then this memory has previously been explicitly mapped

translation yields a linear address which is further translated using the page directory. In practice segments are not actively used so that virtual address and linear address are identical.

from one of the involved address spaces to the other one. This property ensures that a legacy operating system cannot see the memory with the cryptographic keys of the encryption module, unless there is a very stupid programming error in the encryption module.

However, high level properties like this are currently not in our scope. We first have to be successful with more basic properties.

In principle one would like to prove that, whatever code is running inside one of those legacy OS's, it is impossible to break the encryption of the encryption module. However, this requires an attacker model which has not been considered yet in cryptography. Typical attacker models in cryptography are such that the attacker has full access to the messages on the internet and can additionally control some hosts there. What we need here, is an attacker that is able to execute arbitrary code on the CPU that runs the cryptographic engine. Because covert channels can only be minimised but never completely avoided, the attacker can additionally observe the internal state of the cryptographic engine at a very low bit rate. We are not aware of any work with such an attacker model.

5 Conclusion

This paper presents the approach that is followed in Nijmegen to verify some properties of the micro hypervisor, which is currently developed as basis of the Nizza architecture. I also discuss the special challenges of operating-system kernel verification and some interesting properties we would like to verify.

References

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[Fes06] N. Feske. TUD:OS Demo CD. Available at demo.tudos.org, March 2006.

[FH03] N. Feske and H. Härtig. Dope - a window server for real-time and embedded systems. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 74–77, Washington, DC, USA, 2003. IEEE Computer Society.

[FH05] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA*, pages 85–94. IEEE Computer Society, 2005.

[HHF+05] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing, San Jose, CA, USA, December 19-21, 2005*. IEEE, 2005.

[HPHS04] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, page 22, New York, NY, USA, 2004. ACM Press.

[HT] M. Hohmuth and H. Tews. The vfiasco project. Website www.vfiasco.org.

[HT03] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. Technical Report No. 187 Institut für Informatik Universität Freiburg.

[HT05] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programm Languages and Operating Systems*, Glasgow, 2005.

[Int98] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1998.

[lon] longjmp, siglongjmp — non-local jump to a saved stack context. Linux manual page.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, Berlin, 1996.

[OS01] S. Owre and N. Shankar. Theory interpretations in pvs. Technical Report SRI-CSL-01-01, Stanford Research Institute, Computer Science Labority, April 2001.

[Tew07] H. Tews. Micro hypervisor verification: Possible approaches and relevant properties. Accepted at the NLUUG Voorjaarsconferentie 2007, February 2007. Available from `www.cs.ru.nl/~tews/science.html`.

[Vel] T. L. Veldhuizen. C++ templates are turing complete. Available at citeseer.ist.psu.edu/581150.html.