Micro Hypervisor Verification: Possible Approaches and Relevant Properties

Hendrik Tews*

Radboud Universiteit Nijmegen, The Netherlands

http://www.cs.ru.nl/∼tews

April 2, 2007

The SoS-group at the Radboud University Nijmegen participates in the Robin project. The goal of this project is to develop a minimal trusted computing base for virtualizing (multiple instances of different) distrusted legacy operating systems in a secure way. With the resulting system it shall be possible to use, for instance, Word on Microsoft Windows for composing classified documents in a secure way. Thereby it is not necessary to trust Windows in any way, the Windows instance that is used could even have been compromised by an attacker already.

In this paper I give an short overview about the Nizza architecture that is further developed in the Robin project. I further elaborate on our task at Radboud University: To develop a verification approach for the underlying micro hypervisor for relevant security properties.

1 Introduction

This paper gives some information about the Robin project.[1] Robin is a European project with four partners: Technical University Dresden and Secunet Security Networks AG in Germany, ST Microelectronics in France and Radboud University Nijmegen in the Netherlands. The aim of the project is to develop a trustworthy platform for legacy operating-system virtualization, which I outline below. The task of the people at Dresden University is to develop a new trusted computing base (whose properties are described

[*] This work has been supported by the European Union through PASR grant 104600.

[1] See http://robin.tudos.org/

in Section 2), including a new micro-hypervisor operating system. The two industrial partners do case studies and port the Nizza architecture to their products. Our aim in Nijmegen is to develop and evaluate means and methods for the verification of system-level software, in particular for the newly developed micro hypervisor. In the course of the project we will develop methods that allow us to attempt the verification of some properties of the micro hypervisor. (Our aim *is not* to reduce the number of security bugs in the hypervisor. For that we would use techniques like extended static checks [ECCH00] or model checking.)

In this paper I explain the Nizza architecture for exploiting the power of legacy operating systems in a trustworthy way (Section 2). I describe then the specific difficulties that arise from the verification of systems-level code (Section 3). Finally I tell about our verification approach (Section 4) and some interesting properties that we would like to prove and some properties that we cannot yet (Section 5).

2 The Nizza architecture for confidentiality and integrity

We are facing more and more often situations where security considerations are in conflict with usability requirements. For instance:

- Mobile phones and PDAs store private data, which one usually wants to protect. Furthermore, these devices are used more and more for monetary transactions. For security reasons one should only run little carefully selected software on them.

  However, mobile phones and PDAs are also used as mobile web browsers. If possible one wants to run a full blown web browser on them to enjoy the latest bells and whistles of the web. With its numerous security holes any of the major web browsers basically opens the mobile phone or PDA for those who really want access. The problem becomes even worse when people want to download software, for instance games, from the internet.

- The secretary of an embassy has to work with classified data. However, she might also want to install skype to chat with friends at home over lunch break.

- PCs at home are used for many applications and lots of software from very different sources gets installed on them. The current discussion in Germany about online spying on suspected individuals demonstrates that it is currently far to easy to break into a PC. Online banking is obviously only made for people who do not care about their money.

One solution to solve the conflict between security and usability is to use two physically separated devices, a classified one for the data one wants to protect and an unclassified one to play with. There is of course the question what operating system to install on the classified system. With the currently available systems the classified system can only stay secure if it is not connected to the internet. This renders the classified system pretty much useless for private use, of course.

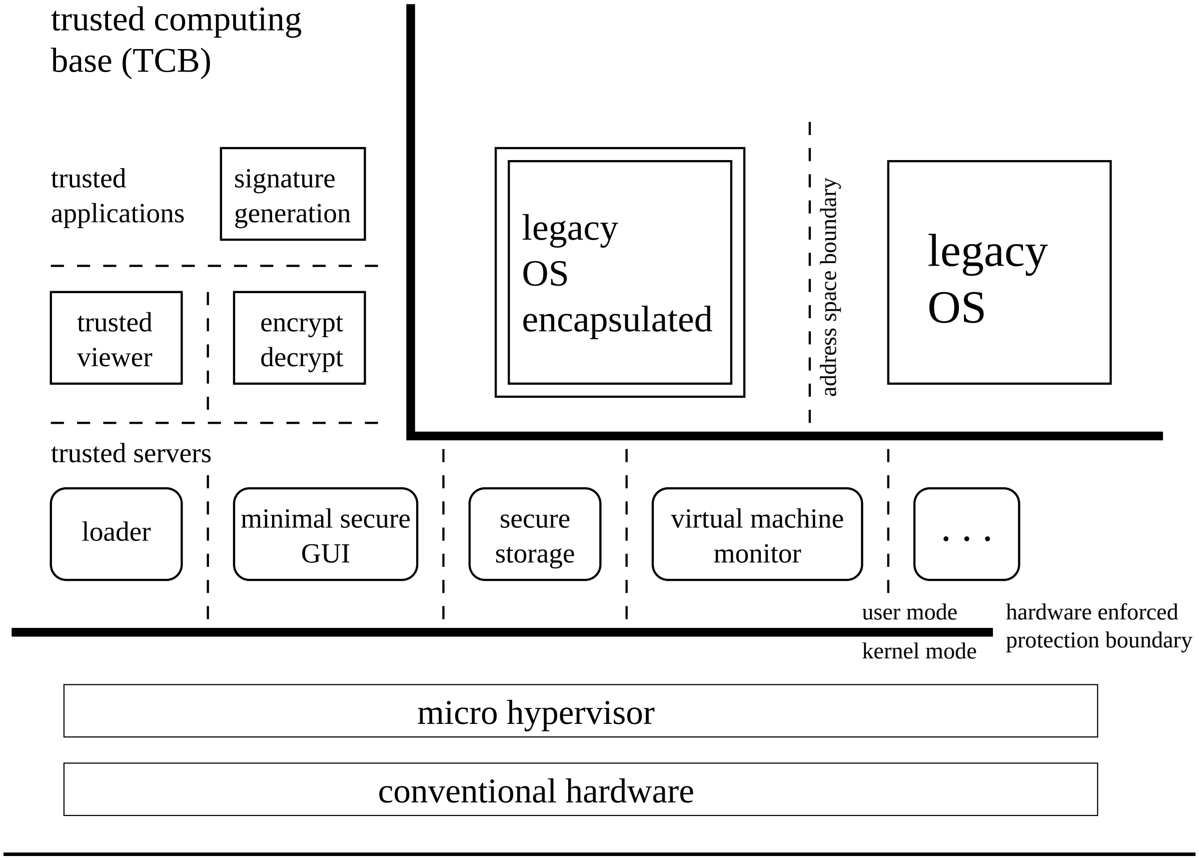


Figure 1: Nizza security architecture

The Robin project aims to provide a better solution for the just described security-usability dilemma. The basic ideas are captured in the Nizza architecture [HHF$^+$05], see Figure 1, that I am going to elaborate on in the following. I emphasis that all the following material about Nizza is my reflections of the original papers [HHF$^+$05, FH05, FH03, HPHS04, Fes06].

In the slightly simplified version of the Nizza architecture shown here the software on the system is categorised into four kinds:

1. The micro kernel or micro hypervisor (simply referred to as hypervisor in the following); the only piece of software that can fully control the hardware.

2. Trusted servers; which yield functionality that has been exported from the kernel into user space.

3. Trusted applications; which are small application programs that run directly on the hypervisor. (The use case that I describe below will require three trusted applications. In principle also non-trusted applications can run directly on the hypervisor.)

4. Untrusted legacy software; which is typically running inside a legacy guest operating system.

The Nizza architecture is able to provide confidentiality (only authorised people have access to sensitive data) and integrity (unauthorised changes of the sensitive data are easy to detect). As discussed here, the Nizza architecture cannot ensure availability (services can be used whenever necessary), through it requires an attacker of extraordinary strength to break the system for more than a few hours.

The hypervisor, the trusted servers and the trusted applications belong to the trusted computing base (TCB). The TCB is that part of the code on which the provided security hinges. In the Nizza architecture important services (such as mass storage from a hard disk) can be provided by untrusted components without comprising security. Because of the modular approach the TCB can vary from application to application (a trusted server that is not necessary for one application need not be trusted, thus it is outside of the TCB)

Let me now explain how the Nizza architecture works. The base consists of a micro kernel or micro hypervisor together with a few trusted servers. The hypervisor is the only piece of software that runs in the most privileged kernel mode of the CPU. The trusted servers provide functionality that is essential for the whole system, but which does not necessarily need to be included in the kernel.

On top of this minimal trusted computing base one can run stand-alone applications in parallel with (several instances of) legacy operating systems such as linux or windows. The new hypervisor that is currently developed within the Robin project supports the virtualization features of the new x86 CPUs. Legacy operating systems such as Linux or Windows can therefore run unmodified as fully virtualized guest operating systems. All the guest OS's, the trusted servers and all applications are separated from each other by an address-space boundary. If desired and permitted two such parties can communicate via interprocess communication (IPC) or set up a region of shared memory.

The system will also permit to enclose an application or a guest OS in a compartment such that it can communicate with the outside world only via dedicated channels. This feature makes it possible to ensure security with a minimal TCB while maintaining usability. This is best explained with an example.

Assume a user wants to work with sensitive data that should be readable only for a dedicated recipient and should further be concealed from any possible attacker. Assume further that for this work he does not want to give up his highly customised working environment and his favourite editor. I will refer to this working environment as *convenience software* because the hypothetical user could of course create and send the sensitive data without his much beloved 3D window manager. With Robin however, the user can use as much convenience software as he likes without compromising security.

For working on the sensitive data the user starts a new copy of his favourite legacy operating system (referred to as editor OS in the following) in a fully encapsulated compartment with only one channel to communicate to the outside world. This channel will be connected to a dedicated secure encryption application. The user can now work on the sensitive data inside the new compartment. For inspiration he can browse the internet using a different instance of his favourite legacy OS that runs as a separate guest (referred to as browser OS). There is no danger, even if this browser OS instance gets compromised and is completely taken over by a remote attacker. The hypervisor and the secure GUI make sure that the attacker will not even become aware of the other OS instance with the sensitive data inside. All the attacker can see is that mouse and keyboard become inactive when the mouse gets close to certain areas on the screen.

When the work on the sensitive data is finished it is passed on to the secure encryption application. Once the data is encrypted it can be passed into the browser OS instance to be sent to the right receiver. The attacker can now only play a denial of service attack and delete the encrypted data. The encryption makes sure that he can not fake it.

Assume now that an attacker of sufficient strength has managed to compromise the installation media which was used for the users favourite legacy OS. Inside the editor OS the spyware of the attacker can now see the sensitive data, however it cannot do anything with it! There is only one channel to the outside and that is controlled by a program that is not under the control of the attacker. The spyware could play a denial of service attack or, a bit more sophisticated, it could replace the sensitive data of the user to let him sign and sent data of the attacker. The latter case can easily be detected if the user checks the data with a trusted viewer, which runs as secure application alongside the guest OS's. So even if the editor OS is compromised there is only the threat of a denial of service attack.

It is important to notice here that all the convenience software, that is used in the described scenario, is not part of the trusted computing base, that is, the user does not need to trust its correctness. Encapsulation and encryption makes it even possible to let the (possibly compromised) legacy OS drive parts of the hardware, for instance the hard disk.

I believe I convincingly argued that the system can ensure confidentiality and integrity. There are several ways to improve the availability of the system that I am not going to discuss in detail. Adding redundancy (for instance starting multiple legacy OS's of possibly different versions for the same purpose) would make the attackers life much harder. In order to make the system unavailable for a long period the attacker must either be able to compromise a wide variety of legacy OS's within seconds or he must compromise all the available installation media for those legacy systems. However, an attacker such strong would probably simply place a tank in front or the poor user to achieve his goals.

Users that need to ensure their availability under all circumstances (like police stations or embassies) must equip the system displayed in Figure 1 with an additionally trusted application that can be used as fall-back system. The fall-back system has to have sufficient independent communication facilities (like a dedicated telephone line). It will of course only provide most basic functionality and therefore be much simpler than the preferred convenience software.

Let me give some more background and details about some components of the Nizza architecture. Work on some components goes back to 1995. At the moment a fully working system is available on a demo CD [Fes06]. It uses the Fiasco micro kernel [HH01] as basis. A major limitation of Fiasco is that it supports neither hardware virtualization nor faithful software virtualization. Therefore, the system on the demo CD supports only para-virtualized or ported guest operating systems. There is currently only one guest available: L^{4}Linux, the linux port of the OS group at Dresden University to Fiasco.

As said, the new hypervisor developed in Robin will support a variety of guests through the virtualization features of recent CPUs. The hypervisor will be the lowest software component of the system and the only one that fully controls the CPU. The hypervisor stands in the tradition of Fiasco and the L4 micro kernel family. It only implements those services that are absolutely necessary: address or memory spaces (especially separation of address and memory spaces), process/task management, inter process communication (IPC) with delegation of resources (especially memory) and round robin scheduling with fixed priorities. There are no traditional device drivers in the hypervisor. It only contains drivers for the interrupt controller and the clock. The hypervisor runs in root mode level 0. All other software components of the system have less privileges. They run either in level 3 (root or non-root mode) or in non-root mode level 0 (for kernels of guest operating systems).

The minimalistic kernel design approach makes it necessary that some additional components are always needed. Those components are called *trusted servers* and they provide functionality that is traditionally provided in the kernel. One of these servers provides secure persistent storage. This storage is mainly needed for cryptographic keys, efficient mass storage can be provided by an encapsulated legacy driver outside of the TCB. The loader is responsible for loading new trusted servers. As you will realize by now the loader itself does not need a hard disk driver, it can rely on an untrusted driver and only check that the data it gets has not been tampered with.

The minimal secure GUI server is far simpler than a stripped down version of X or even a window manager. The minimal secure GUI must only provide the following services:

- multiplex several application windows onto the real screen,
- redirect input from keyboard and mouse to applications,
- separate applications from each other, such that one application cannot read the contents of another application window,
- unforgably identify application windows on user request to detect trojan horses

Feske and Helmuth showed that one can provide this functionality in less than 1,500 lines of code [FH05].

The often emphasised possibility to rely on untrusted legacy drivers (or even whole legacy operating systems) for certain functionality has two important advantages:

- the functionality is outside of the TCB, thus reducing the TCB size, and
- one can rely for free on the good support of diverse and new hardware that is present in some legacy OS's.

The price to pay is, of course, that the system becomes vulnerable to denial of service attacks.

3 Challenges of low-level system-software verification

The task of the group in Nijmegen in the Robin project is to develop means and methods for a mechanical verification of the hypervisor, which forms the basis of the Nizza architecture. The reason for verification is clear: In the long run one wants mathematically sound proofs that the hypervisor fulfils its security promises. A verification of the 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. Consequently the hypervisor is written in C++. C++ programs are more difficult to formalise then, say, Haskell or Clean programs, 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.

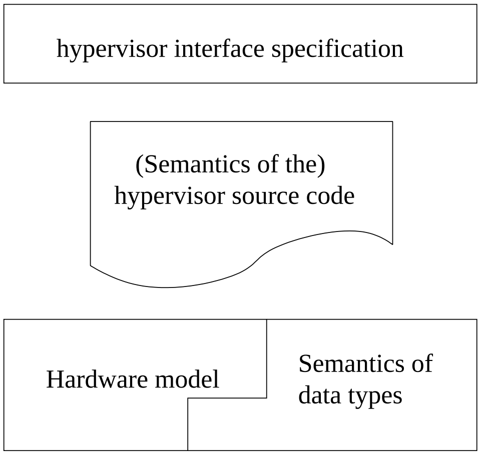
- The template mechanism of C++ alone is Turing complete [Vel]. This means the compiler can be forced to do arbitrary computations *at compile time.* A formalisations of C++ templates is accordingly difficult.

  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



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

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

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

Figure 2: Robin verification approach

(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.

- Embedding special instructions in the code, such as IRET (return from interrupt), 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.

4 A verification approach for a Micro Hypervisor

In this section I explain the approach that we are planning to use for the verification of the micro hypervisor. 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. Higher-order logic contains a complete lambda calculus. For the verification one therefore models the system at hand in a functional way inside PVS and later uses 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.

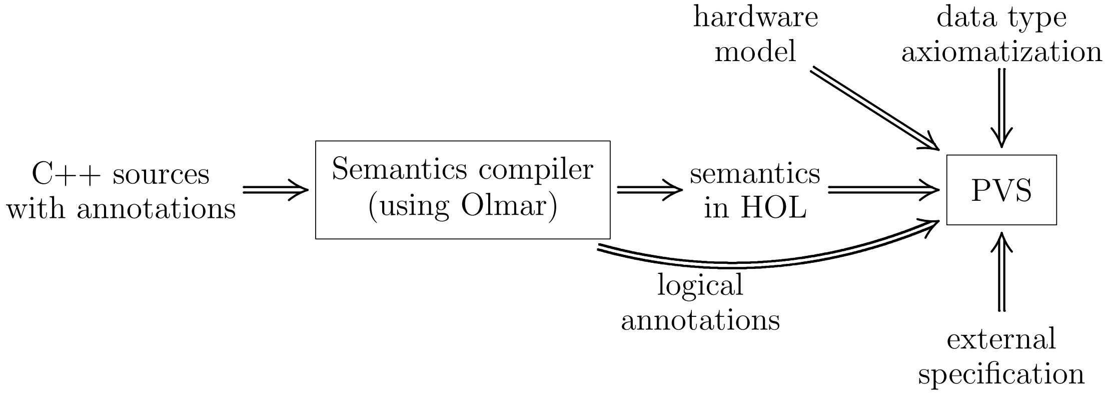


Figure 3: Approach for source code verification

- 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 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[2]

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

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

[2] The symbol $\uplus$ depicts disjoint union. Disjoint union unites two sets without identifying common elements. Formally it is defined as $A \uplus B \overset{\text{def}}{=} \{0\} \times A \cup \{1\} \times B$. The notion $\overset{\mathit{ok}:}{\mathit{State}}$ provides a meaningful name for the numerical tag that is attached to the elements of *State*. The symbol $\mathbf{1}$ stands for the one element set. It is the mathematical counterpart of the unit and void types found in many programming languages.

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.

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.

5 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 regions. 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[3] for which the page directory or page table entry might be inconsistent with the translation look aside buffer)
- 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 that one logical processors (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 is such that a type error introduces some kind of arbitrary state into the proof obligation. Therefore a type error yields an unprovable proof obligation. In order to prove type correctness it is therefore sufficient to prove normal termination for the hypervisor.

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

[3] 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 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.

6 Conclusion

This paper outlines the Nizza security architecture that is further developed within the Robin project in which Radboud University Nijmegen participates. The paper further 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.

[ECCH00] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 October 2000.

[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.

[HH01] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[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.

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