Microhypervisor Verification within the Robin Project (featuring a Nizza architecture demonstration)

Hendrik Tews
SoS group, Radboud University Nijmegen
http://www.cs.ru.nl/∼tews

Supported by the European Union through PASR grant 104600.

Objective: Create an open robust computing platform

Enjoy the latest bells and whistles of the internet.
Without having to worry about the security of online banking.

4 Partners:

- Technical University Dresden (Germany)

  Development/Implementation of the open robust computing infrastructure
- Radboud University Nijmegen

  Formal methods: specification and verification of some parts
- Secunet Security Networks AG (Germany)

  Case study
- ST Microelectronics (France)

  Port the platform to an embedded system (PDA)

Sponsored by the EU through PASR

Conflict between Security and Usebility

- mobile phone/PDA
  - mobile webbrowser
  - store personal data, used for monetary transactions
- PC at home
  - Internetbanking, private correspondence
  - Internet access console

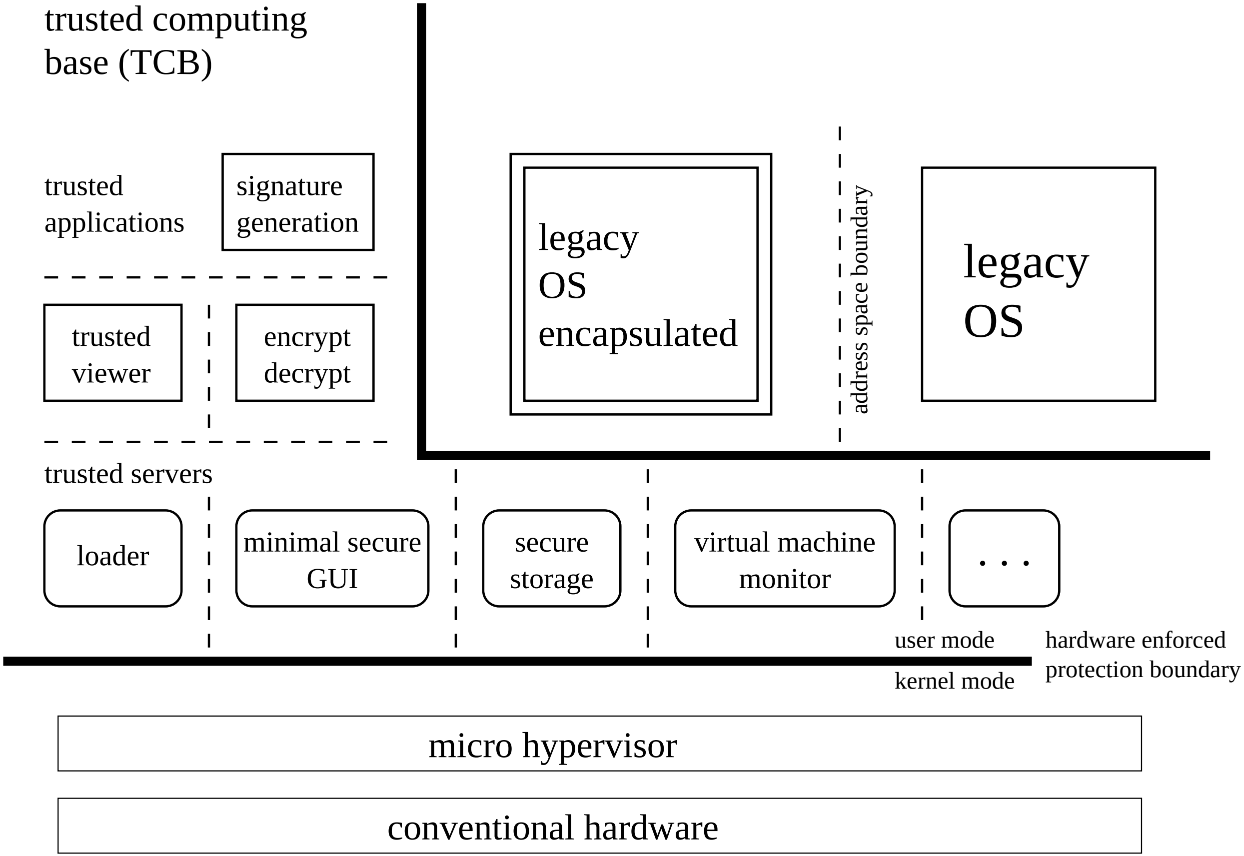
Security considerations:

- closed system
- minimal software

Usability considerations:

- supported OS with large application base (Windows, Linux)
- freely install/update software (from untrusted sources)

For private use: Disconnect from the internet or give up security.


trusted computing
base (TCB)
trusted
applications
signature
generation
trusted
viewer
encrypt
decrypt
legacy
OS
encapsulated
address space boundary
legacy
OS
trusted servers
loader
minimal secure
GUI
secure
storage
virtual machine
monitor
. . .
user mode
kernel mode
hardware enforced
protection boundary
micro hypervisor
conventional hardware

- Use several OS instances in parallel (web-browser instance, editor instance)
- Every OS instance has only limited access and (typically) cannot access other OS instances
- reboot web-browser instance if contaminated to badly
- editor instance can only talk to the encryption module
- Even if attacker compromises installation media he cannot do anything
- data typed in the editor OS is completely secured,
- trusted viewer protects against trojan horses in the editor instance
- Even most of the hardware can be driven by encapsulated lagacy OS instances
- denial of service attacks are the only problem
  (but it requires an extraordinary attacker to deny service
  for more than a few hours)

Some history

1997 MkLinux: *Linux on the the OSF Mach3 microkernel*, too slow

1997 L4Linux, paravirtualized Linux: *The Performence on micro-kernel-based Systems* only 5% performance penalty

2003 XenoLinux: *Xen and the Art of Virtualization*

Comparison

L4, L4Linux	Xen
only Linux paravirtualisation, micro-hypervisor providing full virtualization underway	full virtualization
stand-alone application and OS guests	only OS guests
use case: many cooperating modules, RPC	several, mainly independent guest OS'es; no RPC
IPC latency heavily optimised	IPC throughput optimised
device drivers are separated by address space boundaries	Domain 0 controls all devices
sparse micro-kernel interface	rich hypervisor interface
lots of side channels	?

C++

- OS kernels are typically written in C or C++ enriched with assembly
- standard is very vague (are there negative numbers in signed int?)
- type system is not sound (even without typecasts)

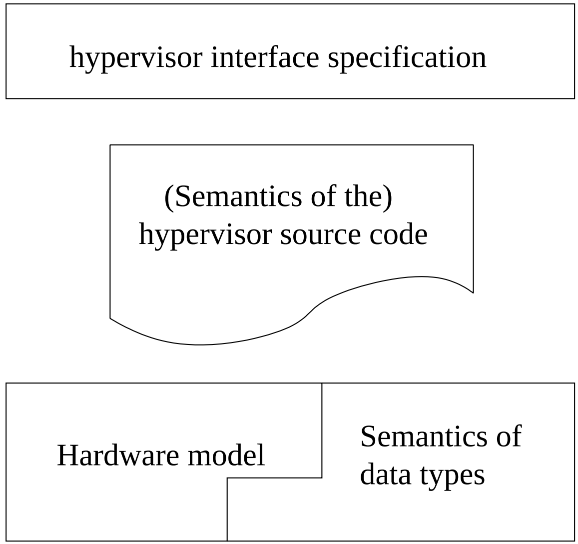
Specifics of kernel Verification

- need type casts (for memory management)
- has to deal with hardware registers that modify the behaviour of the CPU
  - CR3 (page directory base register
  - EFLAGS
  - global descriptor table, interrupt descriptor table
  - task segment
  - feature control register CR0, CR4
- need for assembly (IRET, INVLPG, . . .)
- strange programming environment
  - virtual memory, but the same piece of memory might be visible at different addresses
  - virtual memory mapping is manipulated by the kernel itself (even for kernel memory)
  - strange side effects (memory mapped devices)

Why not write the kernel in a real language (say Haskell) and verify that?

- For some reason, kernels written not in C/C++ only have limited impact.
- Because of memory allocation and hardware access one always has to escape to assembly or C (for kernel programming).
- The runtime system for a safe language is bigger than a whole C++ micro-hypervisor.
- C++ verification adds some additional research challenges.

- use an independent kernel (currently Nova)
- source code verification (of C++)
- develop denotational semantics for a subset of C++
- denotational semantics maps C++ into HOL
- denotational semantics is based on a hardware model and a semantics of C++ data types
- proof properties in the interactive theorem prover PVS

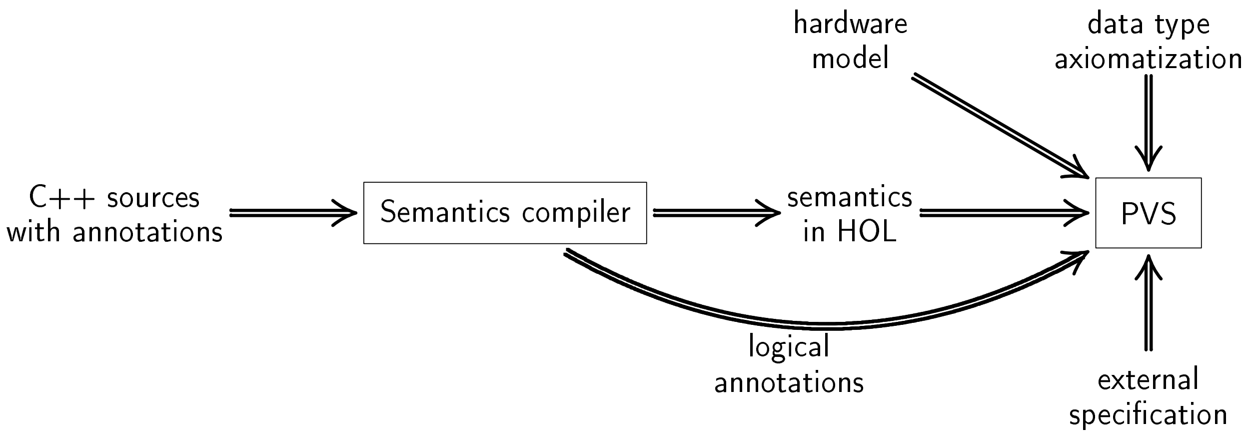


$$\Phi_{data\_types},\quad \Phi_{hardware} \quad \vdash \quad \varphi(\text{hypervisor})$$

- denotational semantics relies on state transformers

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

- specification for the hypervisor interface developed separately
- base specification in pseudo code (simple set theory with lots of syntactic sugar)



- PVS: an interactive theorem prover for higher-order logic
- semantics compiler translates C++ sources into PVS
- program semantics "*is evaluated*" on top of the hardware and the data type model
- verification goals are handwritten or included in the sources as special annotations

- strictly speaking the hardware model is an underspecified hardware specification such that IA32 is a model of it
- provide basic operations for program semantics (reading/writing typed variables in virtual memory)
- physical memory, paging, virtual memory
- TLB
- memory mapped devices
- registers
- provides system state and base operations for source-code semantics:
  - writing in memory
  - reading in memory (which might change the memory state: accessed bits, page faults)
  - special hardware operations: registers (CR3), bits in control registers, ...
- provides a hierarchy of memory interfaces: physical memory, virtual memory, VM with page fault handler
- relies on data type semantics for hardware data types (such as page directory entries)
- stricter check for nonsense/errors than the real hardware (e.g., fail when a string is encountered in the page directory)

- highly underspecified specification for each data type
- three levels: uninterpreted data, interpreted data, pod
- consistency proved with PVS theory refinement
- interface

```
size       : nat,
valid?     : [list[Byte], Address -> bool]
uidt       : Uninterpreted_data_type,
to_byte    : [Data, Address -> list[Byte]],
from_byte  : [list[Byte], Address -> lift[Data]]
```

- leaves the object representation of the data completely open
- the object representation might contain type tags (permitted by the C++ standard)
- only functions for conversion to and from the object representation
- converting from the object representation fails for invalid data
- result of interpreting a string as an integer cannot be determined (not even that the conversion does not crash)
- permitted casts must be given as axioms or additional assumptions
- therefore

Normal termination proves dynamic type correctness

Task

- common abstraction of the various memory interfaces for the majority of the code
- deals with virtual memory aliasing
  (two different virtual address regions are mapped to the same physical memory)
- provides shortcut lemmas for well-behaved variable access

Definition, technically

- invariant parameterised with a set of read-only and a set of read-write addresses with additional properties
- a set of system states that is invariant under all memory read and writes within these sets of addresses
- memory accesses within the address sets terminate normally (no page fault occurs infinitely often)
- only expected changes (no virtual memory aliasing)
- in summary

Memory as one would expect

Goals that we would like to attempt

- absence of the following hardware errors
  - reserved bit violations
  - accessing features not present in the model (such as physical address extension)
  - TLB inconsistency
  - unaligned access to memory mapped hardware devices (such as the Advanced Programmable Interrupt Controller)
- dynamic type correctness
  - absence of conventional type errors
  - TLB errors (missing INVLPG)
  - virtual memory aliasing
  - allocation errors (two variables overlap)
- only kernel code runs in kernel mode

Goals currently out of reach

- address space separation
- attacker does not get access to data in a different address space

Unchanged Object Code

- Goal: on every memory write produce a proof obligation:
  the kernel object code is not changed
- work around: prove separately that kernel object code remains unchanged

Connection between the object code and the semantics of the source code

- assume correct compiler(s) currently

- Nizza architecture solves conflict between security and usability
- verification of the underlying micro-hypervisor is tackled in the SoS group
- use denotational semantics of (a subset of) C++
  to prove simple correctness properties