A Formal Model of Memory Peculiarities for the Verification of Low-Level Operating-System Code

Hendrik Tews Tjark Weber Marcus Völp

3rd International Workshop on Systems Software Verification
February 25, 2008

Robin Project

A trustworthy and verified computing-base for OS virtualization

Nijmegen University — Dresden University
supported by European Union through PASR grant 104600

Kernel (or low level systems) programming is different

Special features

- direct hardware access
- can change the execution environment (eg. switching virtual memory off)
- must obey reserved bits

Special needs

- contain their own memory management, including the stack
- special control structures (`longjmp`, continuations)
- play tricks with the stack

Absurd situations

- Contents changes on read (eg. on memory mapped device registers)
- Spurious changes at completely different addresses
  (because of virtual memory aliases)

What is an error?

Pointer arithmetic without underlying array

- C++ standard: Pointers may only moved inside an array and to the first position after the array
- in kernel programming pointers are sometimes created out of thin air
- kernel memory management impossible with those *thin air pointers*

Explicit TLB invalidation on IA32: When necessary?

- Certainly necessary after restricting access rights in the page tables
- How about when granting additional rights (especially, making a read-only page writable)?
- Best practise is to rely on a page table walk even if the TLB contains an entry with insufficient rights
- Some IA32 manuals require to invalidate the TLB for *every page table change*

What is an error?

Copy arbitrary objects via/into unsigned char arrays

- explicitly legitimated (C++ standard 3.9.2)
- passing arbitrary objects to `printf` this way?
- implement `memcpy` via unsigned char seems ok

Legitimate versus erroneous code

- Distinction is subtle and varies
- Often even depends on the code base!
- A C++ semantics should be flexible
  - To validate/invalidate certain programming patterns easily
  - To adjust the validity depending on the program part

Memory in kernel programming environment

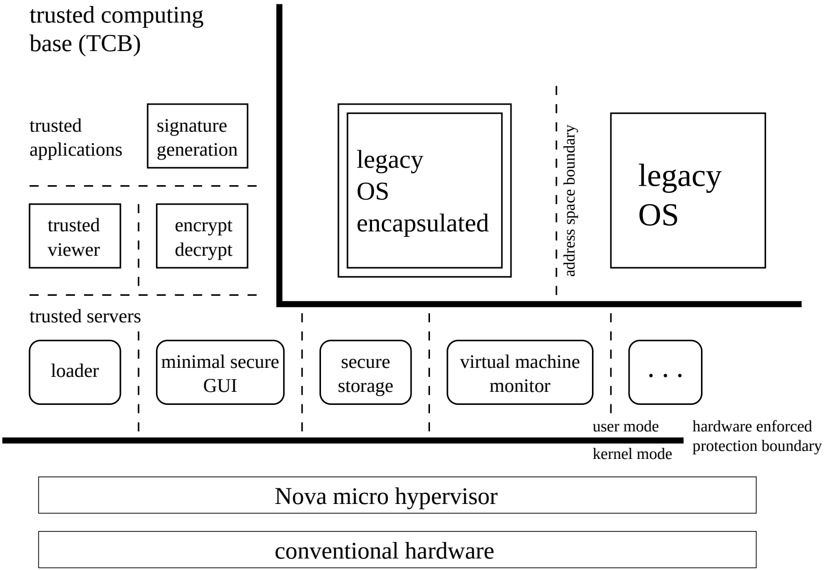
Same memory at different virtual addresses

- often necessary in the kernel, eg. for IPC data windows
- adds another aliasing problem: virtual-memory aliases
- raises subtle questions:
  - It is fine to modify a const via non-const access path (if you have one)
  - How about modifying a const via a virtual-memory alias?
- You cannot check well-typedness at the source- language level

Contains memory mapped devides

- special alignment requirements (8 or 16 bit access might be required)
- contents may change after a read
- reads or writes might change the execution environment
  (eg., acknowledging an interrupt at ACPI)

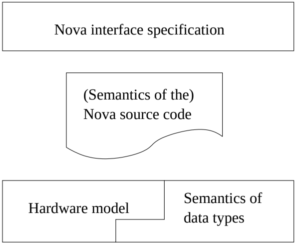
Robin — Build a trustworthy computing base



Targets & Goals

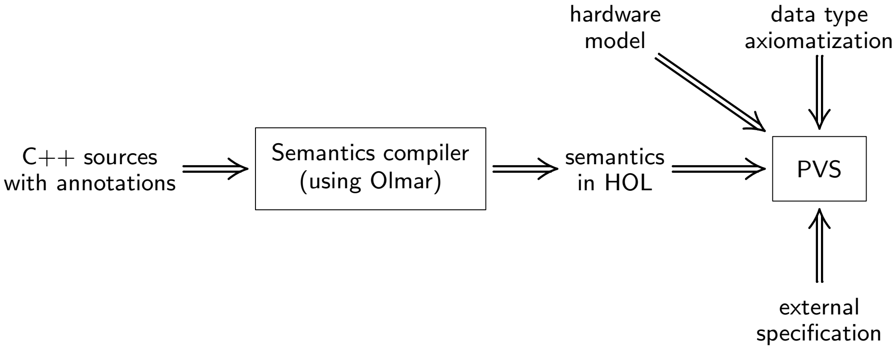
- apply source code verification to the Nova micro-hypervisor
- Nova is currently developed within Robin (in Dresden)
- take the challenge to work with a (simple) C++ code base

Verification environment



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

Dataflow


C++ sources
with annotations
Semantics compiler
(using Olmar)
semantics
in HOL
hardware
model
data type
axiomatization
PVS
external
specification

Goals of the formalisation

Use a single formalisation

- Do not use different semantics for clean and dirty code

Introduce looseness/underspecification for error checking

- Exploit freedom of the C++ standard
- The semantics is not based on a model of the IA32 hardware
- Instead, use a specification for which the real machine is a model
- specification contains other models, for instance type-checking ones
- errors lead to a prove goals
  - that might be true on some models (eg. the real machine) but not on all
  - those prove goal remain therefore unprovable
- some less common features (eg. virtual x86 mode)
  are hardcoded to abort any verification attempt

Verification results are directly applicable

- A successfully verified property holds on the real machine

Hardware Model

Technically: A loose model of IA32 hardware

- However, many aspects are totally specified, justifying “model”
- CPU registers and physical memory (with alignment and size checks)
- virtual memory (page table based address translation)
- integration of devices with side effects possible
- features are separeted into different layers of memory with a uniform interface (providing byte-wise read and write operations)

Features planned

- TLB
- cache policy checks
- segments and offsets
- connection between the high-level semantics and the object code

Features ignored

- other processor modes (virtual 8086, system management)
- multiple processors
- caches (they are transparent anyway for single CPU systems)

Semantics of data types

Semantics of all C++ data types

- de- and encoding of base types into lists of bytes
- suitable operations for compound types
- detached from memory — relies on the abstract memory layer interface

Type correctness through underspecification

- de- and encoding functions are completely underspecified
- decoding might fail on invalid object representation
- thus reading an integer from an string gives leaves the following proof obligation: The string data forms an valid object representation for integer
- might be true in some models, definetely wrong in others
- equivalent to an implicit type-tagging in some models

There would be a lot more to say ...

Plain Memory: Goals

- Different memory layers contain complicated formalisations
- Mostly needed for proving the absence of errors
- plain memory provides a shortcut through the complicated semantics of memory access on the abstract memory-layer interface
- optimised for save and clean code
- makes it possible to prove correctness with respect to several memory layers

Plain Memory: Specification

Formalises well-behaved memory

- Technically: an invariant (wrt. a set of state-transformers) parametrised by the set of read-only blessed and read-write blessed addresses
- reads and writes on blessed addresses succeed and behave as expected
- implies that there is no virtual memory aliasing in the write-blessed addresses (might well be the case in read-blessed addresses)
- should apply to all memory layers
- therefore formalised in terms of the general memory-layer interface

Usage

- on blessed memory one can ignore the page-table translation
- result of read does only depend on the last write
- plain memory provides shortcut lemmas for well-beaved code
- for some rare special operations one shortly has to leave plain memory and reestablish it later again

Plain Memory: Specification

Future enhances

- so far plain memory makes only guarantees about blessed addresses
  non-blessed content may change arbitrarily during blessed access
- need guarantee that certain content stays constant
  (outside of the read-blessed range)

Plain Memory: Technically

Interface

- byte-wise read and write functions
- *states*: a set of (memory) states
- *ro_addr*, *rw_addr*: read-only and read-write blessed addresses

Properties

- access to blessed memory does not fail in all *states*
- read access to $ro\_addr \cup rw\_addr$ leaves blessed addresses unchanged
- write access in *rw_addr* only changes the written address in right way
- the *states* with the above properties are invariant under all memory accesses to blessed addresses
- additional properties for side effects, see later

Plain Memory: Results

Physical memory is plain

- with all addresses read-write blessed

Virtual memory, consisting of *states, ro_addr* and *rw_addr*, is plain if

- some CPU control registers and the relevant page table entries stay constant
  Precisely: `CS`, `PTBR` and the page table entries for $ro\_addr \cup rw\_addr$
  are equal in all *states* (except for access bits)
- address translation succeeds for all blessed addresses
- relevant page table entries are disjoint from $ro\_addr \cup rw\_addr$
  (because of access bits)
- no virtual-memory aliases in *rw_addr*
- Note:
  - virtual aliases in *ro_addr* are permitted
  - irrelevant page-table entries can be blessed
  - could be generalised to have page-table entries with access-bit set in *ro_addr*

Plain memory rewrite lemmas

```
pm : VAR Plain_Memory[State]
s : VAR State
dt1 : Var (interpreted_data_type?[Data1])
dt2 : Var (interpreted_data_type?[Data2])
addr1, addr2 : Var Address
data1 : Var Data1
data2 : Var Data2

plain_memory_read_write_other_q_data_expr : Lemma
  Forall (q : [State -> ExprResult[State,Data_q]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1,addr1, union(pm`ro_addr,pm`rw_addr)) And
    in_blessed_memory?(dt2,addr2,pm`rw_addr) And
    blocks_disjoint?(addr1,size(uidt(dt1)), addr2,size(uidt(dt2))) And
    pm_q_prop_read(pm,dt1,addr1)(q(s))
  Implies
    data((q ## write_data(pm,dt2)(addr2,data2) ## read_data(pm,dt1)(addr1))(s)) =
      data((q ## read_data(pm,dt1)(addr1))(s))
```

Proof of Concept: Array Search

Source Code

```
unsigned int a[N], value;
unsigned int *first = &a[0], *last = &a[N];
unsigned int *current;

current = first;
while(current < last) {
  if (*current == value) break;
  current++;
}
```

Semantics

```
e2s[State, Address, Semantics_void]
   (assign(pm, dt_pointer)(id(current), l2r(pm, dt_pointer)(id(first)))) ##
while(l2r(pm, dt_pointer)(id(current)) < l2r(pm, dt_pointer)(id(last)),
  if_else(
    l2r(pm, dt_uint)(deref(pm)(l2r(pm, dt_pointer)(id(current)))) == literal(value),
    break,
    skip) ##
  e2s[State, Semantics_pointer, Semantics_void](postinc(pm)(id(current)))),
```

Experiences

Rewrite system works, but

- plain memory simplification works from the right-hand side
- need to be combined with some left-to-right expansion, eg. for loops
- first version was highly inefficient (15 minutes for $N = 2$)
- pm_q_prop improves efficiency by reducing the number of subgoals

```
pm_q_prop_read(pm : Plain_Memory[State], dt, addr)
                                (res : ExprResult[State, Data_q]) : bool =
  OK?(res) And
  OK?(read_data(pm, dt)(addr)(state(res))) And
  pm`states(state(res))
```

One solution for several problems

Size and alignment restrictions on CPU registers

- AH has size 1 and is at offset 1 in EAX

Reserved bits

- defined value (may write directly)
- undefined value – must not be changed (always read before write)

Devices

- APIC, but also seriell devices or hard disks
- in reality they are independent and run in parallel

Importance

- needed for correctness
- not very often used

Approach

- avoid concurrency (at least until we verify a sata driver)
- model as a special memory layer
- state transitions are formalised as special *side-effect state transformers*
- the generic interface for multibyte memory access invokes the side-effect state transformers

Enrich the generic memory interface

memory_read_side_effect : [Address, list[Byte], bool −>
[State −> ExprResult[State, list[Byte]]]],
memory_write_side_effect : [Address, list[Byte], bool −>
[State −> ExprResult[State, list[Byte]]]]

- read side effects are invoked *after* normal memory access (can therefore modify the result in addition to the state)
- write side effects are invoked *before* normal memory access (can therefore check if reserved bits are not changed)

Side effect examples

Artificial demonstration example

- memory mapped random number generator
- `rnd_val`: read-only register with random value
- `seed, access_count` : internal state (real natural numbers)

State space of the random device memory layer

$$\text{Random\_device\_memory} \quad = \quad \text{underlying memory} \times \mathbb{N} \times \mathbb{N}$$

Access type and alignment restrictions (read and write side effect)

unaligned_access(a : Address, bl : list[Byte], cp : bool) :
    [Random_device_memory −> ExprResult[Random_device_memory, list[Byte]]] =
  If disjoint?(address_block(a, length(bl)), address_block(rnd_val, size(uidt(dt_uint))))
    OR
    (**NOT** cp **AND** length(bl) = size(uidt(dt_uint)) **AND** a = rnd_val)
  Then ok_result(bl)
  Else fatal_result **Endif**

Side effect examples

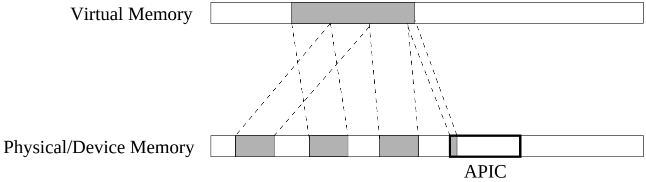
`rnd_val` is read only (write side effect)

```
write_rnd_dev(a : Address, bl : list[Byte], cp : bool) :
     [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
  If a = rnd_val
  Then fatal_result
  Else ok_result(bl) Endif
```

Count memory accesses (read and write side effect)

```
access_count(a : Address, bl : list[Byte], cp : bool)(s) :
                              ExprResult[Random_device_memory, list[Byte]] =
  OK(increase_access_count(s)(length(bl)), bl)
```

Devices and virtual memory



- by chance, last piece is mapped precisely to the first APIC register
- virtual memory layer splits side effects as well and sets the cp (cross-page) argument
- side effects can check cp and react

Conclusion

Plain Memory

- shortcut to the complicated semantics for well-behaved code
- concrete properties/behaviour of memory layers are abstracted away

Side effect transformers

- memory mapped devices, alignment, reserved bits
- side effect transformers modify memory access