Compositional Layered Specification and Verification of a Hypervisor OS Kernel

Zhong Shao

Yale University

December 9, 2013

Joint work with Ronghui Gu, Tahina Ramananandro, Newman Wu, Haozhong Zhang, Yu Guo, and Jeremie Koenig
A Tale of Cosmology

A well-known scientist once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise." The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever," said the old lady. "But it’s tortoises all the way down!"

What about the Cyber World?

Might the old lady actually be right?

The dependency graph in the seL4 kernel
Credit: Klein et al CACM 53(6), June 2010
The CertiKOS Project

Research tasks & key innovations:

• new **clean-slate extensible OS kernels** that “crash-proof” the entire system
• new **prog. languages & logics** for writing certified kernel plug-ins
• new **formal methods** for automating proofs & specs
CertiKOS Port on HACMS UGV

<table>
<thead>
<tr>
<th>USB</th>
<th>Ethernet/NIC</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usb stick</td>
<td>Motor Encoder</td>
<td>Serial console</td>
</tr>
<tr>
<td>Usb kbd</td>
<td>Frame Grabber</td>
<td>Moog Pan/Tilt</td>
</tr>
<tr>
<td>Xbox Controller</td>
<td>Automotive Radar</td>
<td></td>
</tr>
<tr>
<td>Frame Grabber</td>
<td>IMU</td>
<td></td>
</tr>
<tr>
<td>Automotive Radar</td>
<td>Ping &amp; ssh</td>
<td></td>
</tr>
<tr>
<td>IMU</td>
<td>Turret Pan/Tilt (Galil)</td>
<td></td>
</tr>
<tr>
<td>Ping &amp; ssh</td>
<td>Paintball</td>
<td></td>
</tr>
<tr>
<td>Turret Pan/Tilt (Galil)</td>
<td>GPS</td>
<td></td>
</tr>
<tr>
<td>Paintball</td>
<td>Serial console</td>
<td></td>
</tr>
<tr>
<td>GPS</td>
<td>Moog Pan/Tilt</td>
<td></td>
</tr>
</tbody>
</table>

status ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔ ✔
Decompose ROS apps to run on multiple VMs, based on their security-, computation-, & communication needs.
This Talk: How to Certify?

We focus on mCertiKOS (3 kloc C & asm, can boot Linux)

<table>
<thead>
<tr>
<th>User-space</th>
<th>User-space Virtual Machine Manager</th>
<th>Virtual Device 1</th>
<th>...</th>
<th>Virtual Device N</th>
<th>Certified App</th>
<th>Uncertified App</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap</td>
<td>Trap Handlers (interrupts, exceptions, system call handlers)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtualization</td>
<td>AMD SVM Abstraction (primitives for VMCB &amp; NPT)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process &amp; Thread</td>
<td>Process &amp; Thread Management &amp; IPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM</td>
<td>Memory Management (Physical Memory &amp; Virtual Memory Management)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drivers &amp; Preinit</td>
<td>Preinit</td>
<td>PIC Driver</td>
<td>Timer Driver</td>
<td>IDE Driver</td>
<td>SVM Driver</td>
<td></td>
</tr>
<tr>
<td>HW</td>
<td>CPU</td>
<td>Memory</td>
<td>PIC (i8259)</td>
<td>Timer (i8254)</td>
<td>IDE Controller</td>
<td></td>
</tr>
</tbody>
</table>

Certified Kernel
Main Challenges

• Complex interdependencies btw kernel components
  – mixed abstraction layers; subtle invariants

• The cost of verification is still prohibitive
  – seL4: 7500 lines C took 11 person years (an “one-off” effort)
  – 1200 lines C and 500 lines assembly still unverified
  – unclear how to “link” seL4 with other user/kernel code

• C and Asm provide limited support of abstraction
  – They are too low-level …

• C is also too high level
  – need to reason about realistic machine models
  – how to manage and multiplex HW? address translation?

• How to scale? How to support extensibility?
  – we want “semantically correct” certified plug-ins
Our Approach

- Break kernel interdependency by insisting on “formal” layered decomposition

- mCertiKOS is specified & verified in 37 abstraction layers
Our Main Results

A new compositional layered methodology that tackles all the challenges for building verified OS kernels

- Use **layered specs** to untangle kernel interdependencies
  - Each layer is an **assembly level** abstract machine
  - plus abstract state & primitives, defined in Coq w. formal semantics

- Use **contextual refinement** to compose abstraction layers

- Kernel code still written in C and verified at the source level
  - one ClightX(L) lang. for each abstraction layer L
  - new CompCertX compiler for transporting C-level proofs to Asm

- mCertiKOS specified & verified in Coq in **< 1 person year**
  - Aggressive layer decomposition offers great support to extensibility
Our Main Results (cont’d)

Layered Compositional Specification

- Spec for $L_0$ Primitives
- Asm Instructions
- $L_0$ Abs State + Mem + Regs + Devices

Refinement between Layers

- Layer $L_n$ code + machine
- Contextual Refinement

- Layer $L_2$ code + machine
- Contextual Refinement

- Layer $L_1$ code + machine
- Contextual Refinement

- Layer $L_0$ code + machine

Verification of Kernel Code

- $K_n$ Spec in $L_{n-1}$
- $K_n$ ClightX Code

New Certified Compilers

- CompCertX

Kernel ClightX Source Code:

- $K_1$, $K_2$, ..., $K_n$
Outline of this Talk

• Problem Definition & Main Results
• Overview of Our New Layered Approach
• Compositional Layered Specification
  – how to define an abstraction layer
  – case study: decomposing mCertiKOS
• Verification of Kernel Programs
• Implementation & Evaluation
Bottom and Top Layers

**top**

- CertiKOS
  - Registers
  - Memory
  - Abstract State
- ASM Instructions + System Calls

**bottom**

- X86
  - Registers
  - Memory
- ASM Instructions

CertiKOS Code

contextual refinement

P

P
Abstract State

CertiKOS

Registers
Memory
Abstract State

ASM Instructions + System Calls

Critical state
stored in the memory:
page table; TCBs
associated with invariants

Auxiliary state
introduced for specification:
init flag; ipt flag
associated with invariants
Final Theorem

\[ \forall P, \left[ P \otimes K_{Imp} \right]_{X86} \preceq \left[ P \right]_{CertiKOS} \]

\[ \forall P, \left[ P \otimes K_{n-1} \otimes K_1 \otimes K_0 \right]_0 (X86) \]

\[ \ldots \]

\[ \preceq \left[ P \otimes K_{n-1} \otimes K_{i+1} \otimes K_i \right]_i \]

\[ \preceq \left[ P \otimes K_{n-1} \otimes K_{i+1} \right]_{i+1} \]

\[ \ldots \]

\[ \preceq \left[ P \right]_n (CertiKOS) \]
Final Theorem

\[
\left[ \left[ P \otimes K_{n-1} \cdots \otimes K_{i+1} \otimes K_i \right] \right]_i \\
\leq \left[ \left[ P \otimes K_{n-1} \cdots \otimes K_{i+1} \right] \right]_{i+1}
\]

\[ P_{i+1} \]

\[
\left[ \left[ P_{i+1} \otimes K_i \right] \right]_i \\
\leq \left[ \left[ P_{i+1} \right] \right]_{i+1}
\]

layer_i  layer_{i+1}
What a layer does?

$\text{layer}_{i+1}$

$\text{abs-state}$  $\text{memory}$  $\text{primitives}$

$\text{layer}_i$

$\text{abs-state}$  $\text{memory}$  $\text{primitives}$
Example: Page Table

layer\textsubscript{i+1}

\begin{itemize}
  \item \textbf{Abstract state}
  \begin{itemize}
    \item Page Table: vaddr->(paddr, perm)
    \item Invariants: kernel’s page table is a direct map; user part of the pts are isolate;…
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{primitives}
  \begin{itemize}
    \item page_table_init;
    \item page_table_insert;
    \item page_table_rmv;
    \item page_table_read;
  \end{itemize}
\end{itemize}

layer\textsubscript{i}

\begin{itemize}
  \item \textbf{memory}
  \begin{itemize}
    \item uintptr_t page_dir[n_pd];
    \item uintptr_t page_table[n_pd][n_pt];
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{functions}
  \begin{itemize}
    \item int page_table_init();
    \item int page_table_insert();
    \item int page_table_rmv();
    \item int page_table_read();
  \end{itemize}
\end{itemize}
Key Strategy

All layers follow two patterns

They are either “setter-getter” layers or “abstract function impl.” layers

This made layer definitions & layer refinement proofs much simpler and reusable.
Layer: Pattern

Hide concrete memory; replace it with Abstract State
Only the getter and setter primitives can access memory
Refinement Proof

memory_i

memory_i+1

memory injection

abs-state

0 1 2 3

0 1 2 3

22
Layer: Pattern ②

Memory does not change
New kernel code do not access memory directly!
Refinement Proof \(2\)

\[
\boxed{\mathcal{K}_i \ (asm) \supseteq \mathcal{S}_{K_i} \ (asm)}
\]

\[
\boxed{\mathcal{C}_{K_i} \ (asm) \supseteq \mathcal{S}_{K_i} \ (asm)}
\]
Outline of this Talk

• Problem Definition & Main Results

• Overview of Our New Layered Approach

• Compositional Layered Specification
  – how to define an abstraction layer
  – case study: decomposing mCertiKOS

• Verification of Kernel Programs

• Implementation & Evaluation
Layer Definition

- We formalize an x86 assembly language in Coq (built on top of CompCert Asm)
- 72 x86 instrs + kernel/user/host/guest + address translation + page fault + abs-state/primitives
- Address space & memory model
## Decompose mCertiKOS

<table>
<thead>
<tr>
<th>User-space</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>User-space Virtual Machine Manager</td>
<td>Virtual Device 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trap</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap Handlers</td>
<td></td>
</tr>
<tr>
<td>(interrupts, exceptions, system call handlers)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtualization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD SVM Abstraction</td>
<td></td>
</tr>
<tr>
<td>(primitives for VMCB &amp; NPT)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process &amp; Thread</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Process &amp; Thread Management &amp; IPC</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MM</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Management</td>
<td></td>
</tr>
<tr>
<td>(Physical Memory &amp; Virtual Memory Management)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Drivers &amp; Preinit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Preinit</td>
<td>PIC Driver</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Memory</td>
</tr>
</tbody>
</table>
Decomposing mCertiKOS

Based on the abstract machine provided by boot loader

Physical Memory and Virtual Memory Management (11 Layers)
1: MBoot Layer

<table>
<thead>
<tr>
<th>MBoot Layer</th>
<th>mi_get</th>
<th>bootloader</th>
<th>setcr3</th>
<th>pe/ikern/ihost_set</th>
</tr>
</thead>
<tbody>
<tr>
<td>(minfo, init, CR3, pe, ikern, ihost, ipt)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Abstract State** (*)
  - *minfo*: physical memory information table
  - *init*: (logical) whether bootloader/preinit has been called or not
  - *CR3*: abstract CR3 register (start address of page table)
  - *pe*: abstract CR0 register (paging is enabled or not)
  - *ikern*: whether it is in the kernel mode or not
  - *ihost*: whether it is in the host mode or not
  - *ipt*: (logical) whether it is using the kernel’s page table
1: MBoot Layer

<table>
<thead>
<tr>
<th>MBoot Layer</th>
<th>mi_get</th>
<th>bootloader</th>
<th>setcr3</th>
<th>pe/ikern/ihost_set</th>
</tr>
</thead>
<tbody>
<tr>
<td>(minfo, init, CR3, pe, ikern, ihost, ipt)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Primitives**
  - `mi_get`: read the `minfo`
  - `setcr3`: set the start address of the page table
  - `pe/ikern/ihost_set`: set the corresponding `abs-state bit`
  - `bootloader`: bootloader/preinit of mCertiKOS

- Initialization function is marked by **green**
2: MATIntro Layer

- Introduce the page allocation table
- Abstract State
  - `iflags`: `(pe, ikern, ihost, ipt)`
  - `AT`: page allocation table
  - `nps`: number of physical pages
- Primitive
  - `iflags_set`: a set of primitives that set the value of `iflags`
  - `at_get/set`: getter and setter for `AT`
  - `nps_get/set`: getter and setter for `nps`
Concrete data structures in C

```c
struct A {
    unsigned int isnorm;
    unsigned int allocated;
};
struct A AT_LOC[1048576];
```

Abstract state defined in Coq

```coq
(** Allocation table*)
Inductive ATType: Type :=
| ATKern
| ATResv
| ATNorm.

Inductive ATInfo :=
| ATValid (b: bool) (t: ATType)
| ATUndef.

Definition ATable := ZMap.t ATInfo
```
3: MATOp Layer – 4: MAT Layer

- Initialize the allocation table and provide primitives to manipulate the allocation table
- Abstract State
  - minfo and nps are hidden
- Primitive
  - meminit: initialize AT and nps from minfo
  - palloc: allocate a page in High Memory
  - pfree: free a page
## 5: MPTIntro Layer

<table>
<thead>
<tr>
<th>MPTIntro Layer</th>
<th>(init, iflags, AT, PT, ptp)</th>
<th>alloc/free</th>
<th>get/set/rmv_PTE</th>
<th>setPDE</th>
<th>meminit</th>
<th>setPT</th>
<th>iflags_set</th>
</tr>
</thead>
</table>

- **Introduce the two-level page table pool**
- **Abstract State**
  - **PT**: the current page table index
  - **ptp**: page table pool (64 page tables)
- **Primitive**
  - **setPDE**: setter for the first level page table entry
  - **get/set/rmv_PTE**: getter and setter for the second level page table entry
## 5: MPTIntro Layer

<table>
<thead>
<tr>
<th>MPTIntro Layer</th>
<th>palloc/free</th>
<th>get/set/ rmv_PTE</th>
<th>setPDE</th>
<th>meminit</th>
<th>setPT</th>
<th>iflags_set</th>
</tr>
</thead>
</table>

### Concrete data structures in C

```c
struct PTStruct {
    char * pdir[1024];
    unsigned int pt[1024][1024];
};

struct PTStruct PTPool[64];
```

### Abstract state defined in Coq

Inductive PTPerm: Type :=

- PTP | PTU | PTK (b: bool).

Inductive PTInfo:=

- PTValid (v: block) (p: PTPerm) | PTUnPresent | PTUndef.

Definition PTE := ZMap.t PTInfo.

Inductive PDTInfo :=

- PDTValid (pte: PTE) | PDTUndef.

Definition PTable := ZMap.t PDTInfo.

Definition PTPool := ZMap.t PTable.
6: MPTOp Layer – 8: MPTKern Layer

- Initialize the page table (pt) pool
- Primitive
  - PT_inst/read/rmv: insert/read/remove a map to/from a pt
  - PTinitComm: initialize the High Memory part of all the pts
  - PTinitKern: initialize the Low Memory part of the kernel’s pt (with the index 0)
9: MPTInit Layer

- Enable the paging mechanism
- Primitive
  - PT_init:
    - Initialize the kernel’s pt (call PTInitKern)
    - Set the start address of kernel’s pt to CR3 (call set_CR3)
    - Enable paging (call pe_set)
10: MPTBit Layer – 11: MPTNew Layer

- Introduce the bit map for page table pool
- Abstract State
  - `pbit`: bit map for `ptp`
- Primitive
  - `get/set_bit`: getter and setter for the bit map
  - `PT_new/free`: allocate/free a `pt` from `ptp`
  - `PT_resv`: allocate a page and insert a map into `pt`
  - `pmap_init`: enable paging and reserve the 0-th bit in `pbit`
Decomposing mCertiKOS (cont’d)

Current Target:
Single-Core CertiKOS
Thread and Process Management (14 Layers)
12: PKCtx Layer – 13: PKCtxNew Layer

- Introduce the kernel context pool
- Abstract State
  - kctxp: kernel context pool (using ptp as bit map)
- Primitive
  - kctx_switch: kernel context switch (written in assembly)
  - kctx_new: allocate a pt and kernel context (kctx) from ptp
  - mm.prim: primitives provided by memory management
• Introduce and initialize the thread control blocks (tcb) pool
• Abstract State
  – **Ltcp**: low-level tcb pool (using **ptp** as bit map)
  – **mm.abs**: \textit{abs} provided by memory management
• Primitive
  – **Ltcp\_get/set/init**: getter and setter for **Ltcp**
  – **kctx\_free**: free a pt, kctx and tcb from **ptp**
  – **tcbinit**: enable paging and initialize **ptp** and **Ltcp**
16: PTDQintro Layer – 17: PTDQInit Layer

- Introduce and initialize the thread queue (td) pool
- Abstract State
  - Ltqp: low-level td pool (using ptp as bit map)
- Primitive
  - Ltdq_get/set: getter and setter for Ltdqp
  - Len/de/rm_queue: enqueue, dequeue and remove a thread from the low-level thread queue
  - tdqinit: enable paging and initialize ptp, Ltcbp and Ltp
18: PAbQueue Layer

<table>
<thead>
<tr>
<th>PAbQueue Layer</th>
<th>htdqinit</th>
<th>Htcb_get/set</th>
<th>Hen/de/rm_queue</th>
<th>kctx_switch/new/free</th>
<th>mm.prim</th>
</tr>
</thead>
<tbody>
<tr>
<td>(mm.abs, kctxp, Htcbp, Htqp)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Introduce the high-level (abstract) tcb and td
- Abstract State
  - Htcbp: high-level tcb (defined as Coq inductive type) pool
  - Htqp: high-level td (defined as Coq list) pool
- Primitive
  - Htcb_get/set: getter and setter for Htcb
  - Hen/de/rm_queue: enqueue, dequeue and remove a thread from the high-level thread queue
  - htdqinit: enable paging and initialize ptp, Htcbp and Htdp
18: PAbQueue Layer

Low-level tcb and td defined in C

typedef enum {
    TD_READY, TD_RUN, TD_SLEEP, TD_DEAD
} td_state;

struct tcb {
    td_state tstate;
    struct tcb *prev, * next;
};

struct tdqueue {
    struct tcb *head, *tail;
};

static struct tcb tcb_pool[num_proc];
static struct tdqueue ready_queue;
static struct tdqueue sleep_queue [num_chan];

High-level tcb and td defined in Coq

Inductive td_state :=
| TD_READY | TD_RUN | TD_SLEEP | TD_DEAD.

Definition tcb := td_state.
Definition tdqueue := List Z.

tcb_pool : Z -> option tcb.
ready_queue: tdqueue.
sleep_queues : Z-> option tdqueue.
19: PCID Layer

- Introduce the current thread id
- Abstract State
  - cid: current thread id
- Primitive
  - cid_get/set: getter and setter for cid
## 20: PSched Layer - 21: PThread Layer

<table>
<thead>
<tr>
<th>PSched Layer</th>
<th>(mm.abs, kctxp, Htcbp, Htqp, cid)</th>
<th>schedinit</th>
<th>cid_get</th>
<th>Htcb_set</th>
<th>thread_sched/kill/spawn/kill/wakeup</th>
<th>mm.prim</th>
</tr>
</thead>
<tbody>
<tr>
<td>PThread Layer</td>
<td>(mm.abs, kctxp, Htcbp, Htqp, cid)</td>
<td>schedinit</td>
<td>cid_get</td>
<td>Htcb_set</td>
<td>thread_sleep/yield/spawn/kill/wakeup</td>
<td>mm.prim</td>
</tr>
</tbody>
</table>

- **Introduce the primitives for thread management**
- **Primitive**
  - `thread_sched`: thread scheduler (non-preemptive)
  - `thread_spawn/kill`: spawn/kill a thread. Including allocate/free the corresponding pt and tcb, and modify the Htdp
  - `thread_wakeup`: wakeup a sleeping thread
  - `thread_sleep`: sleep for a resource (such as a channel)
  - `thread_yield`: yield to the first ready thread
  - `schedinit`: enable paging, initialize ptp, Ltcbp, Ltpd, and cid
## 22: PIPCIntro Layer

**PIPCIntro Layer**

| (mm.abs, kctxp, Htcbp, Htqp, cid, chanp) | schedinit | get/set_chan | thread.prim | mm.prim |

- Introduce the inter process communication channel pool
- Abstract State
  - *chanp*: channel pool for inter process communication
- Primitive
  - *get/set_chan*: getter and setter for *chanp*
  - *thread.prim*: primitives provided by thread management
23: PIPC Layer

- Initialize the inter process communication channel pool
- Primitive
  - `send_chan`: send the message to a channel
  - `check_chan`: check whether its channel is full or not
  - `recv_chan`: receive the message from its own channel, and wakeup the first thread sleeping on the channel
  - `procinit`: enable paging and initialize `ptp`, `Ltcp`, `Ltcp`, `cid` and `chanp`
24: PUCtx Layer – 25: PProc Layer

- Introduce the user process context (uctx) pool
- Abstract State
  - `uctxp`: user process context pool (using `ptp` as bit map)
  - `thread/proc.abs`: `abs` provided by thread/proc management
- Primitive
  - `get/set/save/restore_uctx`: getter and setter for `uctxp`
  - `proc_create`: create a user process and initialize the uctx
  - `proc_start/exit`: start/exit a user process
Decomposing mCertiKOS (cont’d)

<table>
<thead>
<tr>
<th>Trap</th>
<th>Trap Handlers (interrupts, exceptions, system call handlers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtualization</td>
<td>AMD SVM Abstraction (primitives for VMCS &amp; NPT)</td>
</tr>
<tr>
<td>Process &amp; Thread</td>
<td>Process &amp; Thread Management &amp; IPC</td>
</tr>
<tr>
<td>MM</td>
<td>Memory Management (Physical Memory &amp; Virtual Memory Management)</td>
</tr>
<tr>
<td>Drivers &amp; Peripheral</td>
<td>Present</td>
</tr>
<tr>
<td>HW</td>
<td>CPU</td>
</tr>
</tbody>
</table>

Virtualization Support (9 Layers)
26: VNPTIntro Layer – 27: VNPTInit Layer

- Introduce and initialize the (guest) nested page table
- Abstract State
  - npt: nested page table (for guest mode)
- Primitive
  - set_NPDE: setter for the first-level npt entry
  - set_NPTE: setter for the second-level npt entry
  - nptinit: initialize the process management (PM) and the npt
  - proc.prim: primitives provided by PM
28: VSVMSwitch Layer – 29: VSVMOp Layer

- Introduce context for host mode
- Abstract State
  - hctx: host context
- Primitive
  - restore/save_hctx: restore and save the host context
  - switch_to_guest: save hctx and switch to guest mode
  - switch_from_guest: restore hctx and switch to host mode
30: VVMCBI Intro Layer – 31: VMCBI Init Layer

- Introduce and initialize the virtual machine control block
- Abstract State
  - **vmcb**: virtual machine control block
- Primitive
  - **vmcb_read/write**: getter and setter for **vmcb**
  - **vmcbinit**: initialize the PM, **npt** and **vmcb**
32: VSVMIntro Layer

- Introduce the virtual machine state
- Abstract State
  - `vmst`: virtual machine state, including `vmcb` and other registers (EBX, ECX, EDX, ESI, EDI, EBP)
- Primitive
  - `vm_st_read/write`: getter and setter for `vmst`
33: VMCBOp Layer – 34: VSVM Layer

- Introduce and initialize the virtual machine control block
- Primitive
  - `vm_run`: save/restore the hctx/vmst, run a virtual machine
  - `vm_exit`: exit a virtual machine, restore/save the hctx/vmst
  - `vmcb_check/clear/inject/set/get (13)`: 13 primitives to read and modify the vmcb
  - `svm_check/exit/sync/inject/set/get (16)`: 16 primitives to read and modify the vmst
Decomposing mCertiKOS (cont’d)

Current Target: Single-Core CertiKOS Syscall and Trap Handlers (3 Layers)

Syscall and Trap Handlers (3 Layers)
• Wrap the trap handler body with the argument getter and return value setter

• Abstract State
  – virt.abs: abs provided by Virtual Machine Module

• Primitive
  – get_arg/set_ret: getter and return value setter
  – sys_check/exit/sync/inject/set(chan) (17): 17 system calls
  – virt.prim: primitives provided by Virtual Machine Module
Top Layer: system calls and internal primitives of mCertikOS

Primitives

- **sys_run**: system call to run a virtual machine
- **pagefault_handler**: get the linear address that page fault happens from CR2 register, allocate a page and insert a pt map, and re-execute the instruction causes page fault
Outline of this Talk

• Problem Definition & Main Results
• Overview of Our New Layered Approach
• Compositional Layered Specification
  – how to define an abstraction layer
  – case study: decomposing mCertiKOS
• Verification of Kernel Programs
• Implementation & Conclusions
Verification of ClightX Code

• We derive Hoare-style inference rules from the ClightX big step semantics

• We prove “termination-sensitive” refinement (i.e., total correctness)

• Two types of kernel functions
  – Abstracting kernel data structures (setter/getter)
  – Abstracting kernel function implementation
18: PAbQueue Layer

Low-level tcb and td defined in C

typedef enum {
    TD_READY, TD_RUN, TD_SLEEP, TD_DEAD
} td_state;

struct tcb {
    td_state tstate;
    struct tcb * prev, * next;
};

struct tdqueue {
    struct tcb *head, *tail;
};

static struct tcb tcb_pool[num_proc];
static struct tdqueue ready_queue;
static struct tdqueue sleep_queue [num_chan];

High-level tcb and td defined in Coq

Inductive td_state :=
|TD_READY |TD_RUN |TD_SLEEP |TD_DEAD.

Definition tcb := td_state.
Definition tdqueue := List Z.

tcb_pool : Z -> option tcb.
ready_queue: tdqueue.
sleep_queues : Z-> option tdqueue.
Thread_wakeup: C code

#define TSTATE_READY 0
#define num_chan 64
#define num_proc 64

void thread_wakeup(unsigned int chan_index)
{
    unsigned int proc_index;
    proc_index = dequeue(chan_index);
    if(proc_index != num_proc)
    {
        set_state(proc_index, TSTATE_READY);
        enqueue(num_chan, proc_index);
    }
}
Thread_wakeup: ClightX function body

Definition thread_wakeup_body : statement :=
  (Ssequence
    (Scall (Some tproc_index) (Evar dequeue (Tfunction (Tcons tint Tnil) tint))
      ((Etempvar tchan_index tint) :: nil))
    (Sifthenelse (Ebinop One (Etempvar tproc_index tint)
      (Econst_int (Int.repr num_proc) tint) tint)
      (Ssequence
        (Scall None (Evar set_state (Tfunction (Tcons tint (Tcons tint Tnil)) tvoid)))
        ((Etempvar tproc_index tint) :: (Econst_int (Int.repr TSTATE_READY) tint) :: nil))
        (Scall None (Evar enqueue (Tfunction (Tcons tint (Tcons tint Tnil)) tvoid))
          ((Econst_int (Int.repr num_chan) tint) :: (Etempvar tproc_index tint) :: nil))
        Sskip)).
Thread_wakeup: ClightX function definition

Definition bthread_wakeup := {|
    fn_return := tvoid;
    fn_params := ((tchan_index, tint) :: nil);
    fn_vars := nil;
    fn_temps := ((tproc_index, tint) :: nil);
    fn_body := thread_wakeup_body
|}. 
Thread_wakeup Specification

Def. thread_wakeup_spec (abd: AbData) (n: Z): option RData :=
let adt := ADT abd in
  match (ipt adt, pe adt, ihost adt) with
  | (true, true, true) =>
    if zle 0 n then
      if zlt n num_chan then
        match (ZMap.get n (abq adt)) with
          | AbQValid l =>
            let la := last l num_proc in
              if zeq la num_proc then
                  ..........................................................
Thread_wakeup: Final Theorem

Theorem thread_wakeup_correct:
forall m m' chan_index abd,
   PCURID.thread_wakeup_spec (Mem.get_abstract_data m)
   (Int.unsigned chan_index) = Some abd

   ->
   m' = Mem.put_abstract_data m abd

   ->
   bigstep_terminates ge bthread_wakeup (Vint chan_index::nil)
       m (Tcons tint Tnil) tvoid E0 (Vundef, m').
Outline of this Talk

• Problem Definition & Main Results

• Overview of Our New Layered Approach

• Compositional Layered Specification
  – how to define an abstraction layer
  – case study: decomposing mCertiKOS

• Verification of Kernel Programs

• Implementation & Evaluation
What we have done

The Complete Coq Specification & Verification of the mCertiKOS Hypervisor Kernel in x86 Assembly

37 layers
Definitions & Refinement Proofs & ClightX function verification

3000+ LOC C source code
CompCertX For ClightX(L)
## Development Cost: A Breakdown

<table>
<thead>
<tr>
<th>Development Type</th>
<th>Effort (pm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development of ClightX and CompCertX</td>
<td>10</td>
</tr>
<tr>
<td>Development of VCGen for ClightX</td>
<td>1.5</td>
</tr>
<tr>
<td>Verification of mm (Design: first 3 layers)</td>
<td>0.5</td>
</tr>
<tr>
<td>Verification of mm (Design: the rest 8)</td>
<td>0.5</td>
</tr>
<tr>
<td>Verification of mm (Refinement Proof: first 2)</td>
<td>1.2</td>
</tr>
<tr>
<td>Verification of mm (Refinement Proof: the rest)</td>
<td>1</td>
</tr>
<tr>
<td>Verification of mm (C verification)</td>
<td>2.5</td>
</tr>
<tr>
<td>Verification of proc (Design: 14 layers)</td>
<td>1</td>
</tr>
<tr>
<td>Verification of proc (Refinement Proof)</td>
<td>0.5</td>
</tr>
<tr>
<td>Verification of proc (C Verification)</td>
<td>1</td>
</tr>
<tr>
<td>Verification of virt (Design: 9 layers)</td>
<td>0.6</td>
</tr>
<tr>
<td>Verification of virt (Refinement Proof)</td>
<td>0.4</td>
</tr>
<tr>
<td>Verification of virt (C Verification)</td>
<td>0.3</td>
</tr>
<tr>
<td>Verification of trap (Design: 3 layers)</td>
<td>0.2</td>
</tr>
<tr>
<td>Verification of trap (Refinement Proof)</td>
<td>0.1</td>
</tr>
<tr>
<td>Verification of trap (C Verification)</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9.9 pm</strong></td>
</tr>
<tr>
<td><strong>VCG Dev</strong></td>
<td><strong>1.5 pm</strong></td>
</tr>
</tbody>
</table>
Average Effort & TCB

• **Current average effort**
  – Layer definition: 2 person days / layer
  – Refinement proofs: 1 person day
  – C code verification: 20 lines Coq per line of C

• **TCB: what are not verified?**
  – 300 lines of C (memcopy, varargs, …)
  – 170 assembly (switch btw ring 0/3 & guest/host)
  – bootloader & preinit & x86 assembler
Why it worked so well?

• The layered approach offers great tradeoffs:
  – Replace a monolithic spec w. extensible layered specs
    • Layered specs are really useful on their own
  – Break complex SW into simple & isolated components
    • Most C code will be synthesized from the Specs
  – Layer refinement proofs follow 2 patterns
    • Great for automation!

• Future certified SW development: *just write the layered specs!*
Related Work

• Dijkstra: “stepwise refinement”
  – Linearly ordered layers of abstract machines

• Hierarchical Development Methodology (HDM)
  – SRI PSOS (and then KSOS) and now “LAW”

• Correctness-by-Construction [Praxis 2002]

• The seL4 project [Klein et al 2008 – now]
  – first mechanized proofs for a real kernel
  – two layers (C -> Haskell-like Spec -> AbsSpec)
  – C-level abstract machine (no addr translation)
Conclusions

• **Compositional layered methodology is the way to go!**
  – It simplifies the design & specification
  – It dramatically improves the extensibility
  – It will be absolutely critical for verified SW

• **It needs to be pursued more aggressively**
  – *Abstract* as soon and as much as we can!

• **But it would only work if we enforce the abstraction formally & rigorously (e.g., Coq)**

• **Still need better PL and tool support**
  (we use Coq + CompCertX + ClightX + Assem)
Future: Evolving CertiKOS

Decomposition & Abstraction

Current Implementation

Compositional Specification & Verification

Developing New OS Kernels w. Certified Plugins