Towards
Trustworthy Systems
Feature-Rich Embedded Systems

Are they trustworthy?
An exception 06 has occurred at 0028:C11B3ADC in VxD DiskTSD(03) + 00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
The Problem
Small Kernels

- Smaller, more trustworthy foundation
  - Hypervisor, microkernel, nano-kernel, virtual machine monitor, isolation kernel, partitioning kernel, exokernel…..
  - Fault isolation, fault identification, IP protection, modularity…..
  - High assurance components in presence of other components
Selected Scenarios

Router/Firewall, Clark-Wilson, etc…

Spatial Partitioning
MILS is about Evaluatable Separation and Connectivity: Physical and Logical
Trustworthy Microkernel?

- API?
  - What might be a secure API
- Correctness?
  - Code reviews?
  - Testing?
  - Static analysis?
    - Sound and unsound
- System level correctness?
Overview

• seL4 microkernel design – security perspective
  – some security background
  – application to microkernel

• seL4 microkernel correctness
  – Formal verification
  – Formal methods versus OS Hackers
  – Proof of correctness of seL4
BACKGROUND
Protection Mechanisms

• Protection state of system
  – Describes current settings, values of system relevant to protection

• Access control matrix
  – Describes protection state precisely
  – Matrix describing rights of subjects
  – State transitions change elements of matrix
## Description

- Subjects $S = \{ s_1, \ldots, s_n \}$
- Objects $O = \{ o_1, \ldots, o_m \}$
- Rights $R = \{ r_1, \ldots, r_k \}$
- Entries $A[s_i, o_j] \subseteq R$
- $A[s_i, o_j] = \{ r_x, \ldots, r_y \}$ means subject $s_i$ has rights $r_x, \ldots, r_y$ over object $o_j$

<table>
<thead>
<tr>
<th></th>
<th>$o_1$</th>
<th>$\ldots$</th>
<th>$o_m$</th>
<th>$s_1$</th>
<th>$\ldots$</th>
<th>$s_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ldots$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 1

- Processes $p, q$
- Files $f, g$
- Rights $r, w, x, a, o$

<table>
<thead>
<tr>
<th></th>
<th>$f$</th>
<th>$g$</th>
<th>$p$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>rwo</td>
<td>$r$</td>
<td>rwxo</td>
<td>$w$</td>
</tr>
<tr>
<td>$q$</td>
<td>$a$</td>
<td>ro</td>
<td>$r$</td>
<td>rwxo</td>
</tr>
</tbody>
</table>
Example 2

- Procedures *inc ctr*, *dec ctr*, *manage*
- Variable *counter*
- Rights +, –, *call*

<table>
<thead>
<tr>
<th></th>
<th>counter</th>
<th>inc_ctr</th>
<th>dec_ctr</th>
<th>manage</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>inc_ctr</em></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>dec_ctr</em></td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>manage</em></td>
<td></td>
<td><em>call</em></td>
<td><em>call</em></td>
<td><em>call</em></td>
</tr>
</tbody>
</table>
State Transitions

- Change the protection state of system
- \( \xi \rightarrow \xi' \) represents transition
  - \( \xi \rightarrow^\tau \xi' \): command \( \tau \) moves system from state \( \xi \) to \( \xi' \)
  - \( \xi \rightarrow^{*} \xi' \): a sequence of commands moves system from state \( \xi \) to \( \xi' \)
- Commands often called *transformation procedures*
Primitive Operations

- **create subject** \( s \); **create object** \( o \)
  - Creates new row, column in ACM; creates new column in ACM

- **destroy subject** \( s \); **destroy object** \( o \)
  - Deletes row, column from ACM; deletes column from ACM

- **enter** \( r \) **into** \( A[s, o] \)
  - Adds \( r \) rights for subject \( s \) over object \( o \)

- **delete** \( r \) **from** \( A[s, o] \)
  - Removes \( r \) rights from subject \( s \) over object \( o \)
Creating File

- Process $p$ creates file $f$ with $r$ and $w$ permission

```plaintext
command create_file(p, f)
    create object f;
    enter own into A[p, f];
    enter r into A[p, f];
    enter w into A[p, f];
end
```
Mono-Operational Commands

- Make process $p$ the owner of file $g$

  ```
  command make\cdot owner(p, g)
  enter own into A[p, g];
  end
  ```

- Mono-operational command
  - Single primitive operation in this command
Conditional Commands

- Let $p$ give $q$ $r$ rights over $f$, if $p$ owns $f$

  command grant \cdot read \cdot file \cdot 1(p, f, q)
  
  if own in A[p, f]
  
  then

  enter $r$ into A[q, f];

  end

- Mono-conditional command
  - Single condition in this command
Multiple Conditions

- Let $p$ give $q$ $r$ and $w$ rights over $f$, if $p$ owns $f$ and $p$ has $c$ rights over $q$
  
  command $grant\cdot read\cdot file\cdot 2(p, f, q)$
  
  if own in $A[p, f]$ and $c$ in $A[p, q]$
  then
    enter $r$ into $A[q, f]$;
    enter $w$ into $A[q, f]$;
  end
Copy Right

- Allows possessor to give rights to another
- Often attached to a right, so only applies to that right
  - \( r \) is read right that cannot be copied
  - \( rc \) is read right that can be copied
- Is copy flag copied when giving \( r \) rights?
  - Depends on model, instantiation of model
Own Right

- Usually allows possessor to change entries in ACM column
  - So owner of object can add, delete rights for others
  - May depend on what system allows
    - Can’t give rights to specific (set of) users
    - Can’t pass copy flag to specific (set of) users
Attenuation of Privilege

• Principle says you can’t give rights you do not possess
  –Restricts addition of rights within a system
  –Usually *ignored* for owner
    • Why? Owner gives herself rights, gives them to others, deletes her rights.
Key Points

• Access control matrix simplest abstraction mechanism for representing protection state
• Transitions alter protection state
• 6 primitive operations alter matrix
  – Transitions can be expressed as commands composed of these operations and, possibly, conditions
What Is “Secure”?

- Adding a generic right $r$ where there was not one is “leaking”
- If a system $S$, beginning in initial state $s_0$, cannot leak right $r$, it is safe with respect to the right $r$. 
Safety Question

• Does there exist an algorithm for determining whether a protection system $S$ with initial state $s_0$ is safe with respect to a generic right $r$?
  – Here, “safe” = “secure” for an abstract model
Mono-Operational Commands

• Answer: yes

• Sketch of proof:
  Consider minimal sequence of commands $c_1, \ldots, c_k$ to leak the right.
  – Can omit delete, destroy
  – Can merge all creates into one

Worst case: insert every right into every entry; with $s$ subjects and $o$ objects initially, and $n$ rights, upper bound is $k \leq n(s+1)(o+1)$
General Case?

• If we know that a specific case has no algorithm, then the general case has no algorithm
  – i.e. we have counter-example.

• How do we show a specific formulation of
  – access control matrix
  – transformation commands

yields an *undecidable* system?
General Case

- Sketch of proof: Reduce the safety problem to the halting problem.
  - i.e. show the ACM and transformation commands implement a Turing machine
Turing Machine Review

Turing Machine review:

- Infinite tape in one direction
- States $K$, symbols $M$; distinguished blank $b$
- Transition function $\delta(k, m) = (k', m', L)$ means in state $k$, symbol $m$ on tape location replaced by symbol $m'$, head moves to left one square, and enters state $k'$
- Halting state is $q_f$; TM halts when it enters this state
Mapping

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>...</td>
</tr>
</tbody>
</table>

- Head

Current state is \( k \)

<table>
<thead>
<tr>
<th>( s_1 )</th>
<th>( s_2 )</th>
<th>( s_3 )</th>
<th>( s_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>A</td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>( s_2 )</td>
<td>B</td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>( s_3 )</td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>( s_4 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After $\delta(k, C) = (k_1, X, R)$ where $k$ is the current state and $k_1$ the next state

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>A</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>B</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td>X</td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
<td>D</td>
<td>$k_1$ end</td>
</tr>
</tbody>
</table>
\[ \delta(k, C) = (k_1, X, R) \] at intermediate becomes

**command** \( c_{k,c}(s_3, s_4) \)

**if** own in \( A[s_3, s_4] \) **and** \( k \) in \( A[s_3, s_3] \)
**and** \( C \) in \( A[s_3, s_3] \)

**then**
- delete \( k \) from \( A[s_3, s_3] \);
- delete \( C \) from \( A[s_3, s_3] \);
- enter \( X \) into \( A[s_3, s_3] \);
- enter \( k_1 \) into \( A[s_4, s_4] \);

**end**
After $\delta(k_1, D) = (k_2, Y, R)$ where $k_1$ is the current state and $k_2$ the next state

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>A</td>
<td>own</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>B</td>
<td>own</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td>X</td>
<td>own</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
<td>Y</td>
<td>own</td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$b k_2$ end</td>
</tr>
</tbody>
</table>
\[ \delta(k_1, D) = (k_2, Y, R) \] at end becomes

```plaintext
command crightmost_{k, c}(s_4, s_5)
if end in A[s_4, s_4] and k_1 in A[s_4, s_4]
    and D in A[s_4, s_4]
then
    delete end from A[s_4, s_4];
    create subject s_5;
    enter own into A[s_4, s_5];
    enter end into A[s_5, s_5];
    delete k_1 from A[s_4, s_4];
    delete D from A[s_4, s_4];
    enter Y into A[s_4, s_4];
    enter k_2 into A[s_5, s_5];
end
```
Rest of Proof

• Protection system exactly simulates a TM
  – Exactly 1 end right in ACM
  – 1 right in entries corresponds to state
  – Thus, at most 1 applicable command
• If TM enters state $q_f$, then right has leaked
• If safety question decidable, then represent TM as above and determine if $q_f$ leaks
  – Implies halting problem decidable
• Conclusion: safety question undecidable in general

Take-Grant Protection Model

- A specific (not generic) system
  - there are other models as well
  - Set of rules for directed graph transformations
    - take rule allows a subject to take rights of another subject (add a vertex)
    - grant rule allows a subject to grant own rights to another subject (add a vertex)
    - create rule allows every subject to create new objects (add a node)
    - remove rule allows a subject to remove rights it has over on another object.
take
grant
create
remove
Take-Grant Protection Model

- Safety decidable, and in time linear in the size of the system
- What this means is that, given the initial state of a system, we can determine if it is safe.
  - If a right can “leak”.

Key Points

- Safety problem undecidable
- Limiting “power” of systems can make problem decidable
- The set of protection commands that model a particular security policy affects whether safety of that model is decidable.
Returning to Kernel Design

Slides based on those by Dhammika Elkaduwe, for IIES’08
Small Kernel Approach

- Smaller, more trustworthy foundation
  - Hypervisor, microkernel, isolation kernel, ..... 
- Facilitate controlled integration and isolation
  - Isolate: fault isolation, diversity
  - Integrate: performance
Small Kernel Approach

- Smaller, more trustworthy foundation
  - Hypervisor, microkernel, isolation kernel, .....
- Facilitate controlled integration and isolation
  - Isolate: fault isolation, diversity
  - Integrate: performance

Microkernel should:
- Provide sufficient API
- Correct realisation of API
- Adhere to isolation/integration requirements of the system
seL4 Protection

- Uses segregated capabilities to kernel objects
- Protection model based on take-grant
Issue

- Kernel consumes resources
  - Machine cycles
  - Physical memory (kernel metadata)

Example:
- threads – thread control block,
- address space – page-tables
- bookkeeping to reclaim memory
Possible Approaches

How do we manage kernel metadata?

- Cache like behaviour [EROS, Cache kernel, HiStart..]
  - No predictability, limited RT applicability

- Static allocations
  - Works for static systems
  - Dynamic systems: overcommit or fail under heavy load

- Domain specific kernel modifications?
• **L4.Verified project:** 
  Formally verify the implementation correctness of the kernel

• **Properties:**
  – Isolation, information flow ...

• Formal refinement
  – Formally connect the properties with the kernel implementation

Mathematically proven properties

Abstract Model

Property preserving refinement

C Code

HW
• L4Verified project:
  Formally verify the implementation correctness of the kernel

• Properties:
  – Isolation, information flow ...

• Formal refinement
  – Formally connect the properties with the kernel implementation
• L4. Verified project:
  Formally verify the implementation correctness of the kernel

• Properties:
  – Isolation, information flow ...

• Formal refinement
  – Formally connect the properties with the kernel implementation
  – Modifications invalidate refinement
  – Verification is labour intensive
    • 10K C-lines = 100K proof lines (1st refinement)
    • Memory management is core functionality
Approach in a nutshell

- No implicit allocations within the kernel
  - No heap, no slab allocation etc..
- All abstractions are provided by first-class kernel objects
  - Threads – TCB object
  - Address space – Page table objects
- All objects are created upon explicit user request
No implicit allocations within the kernel

Physical memory is divided into untyped objects

Authority conferred via capabilities

Untyped capability is sufficient authority to allocate kernel objects

All abstractions are provided via first class kernel objects

Allocate on explicit user request

Creator gets the full authority

Distribute capabilities to allow other access the service
No implicit allocations within the kernel

Physical memory is divided into untyped objects

Authority conferred via capabilities

Untyped capability is sufficient authority to allocate kernel objects

All abstractions are provided via first class kernel objects

Allocate on explicit user request

Creator gets the full authority

Distribute capabilities to allow other access the service

- Kernel objects
  - Untyped
  - TCB (Thread Control Blocks)
  - Capability tables (CT)
  - Comm. ports ....
- No implicit allocations within the kernel
- Physical memory is divided into untyped objects
  - Authority conferred via capabilities
  - Untyped capability is sufficient authority to allocate kernel objects
- All abstractions are provided via first class kernel objects
- Allocate on explicit user request
  - Creator gets the full authority
  - Distribute capabilities to allow other access the service
• **Delegate** authority
  - Allow others to obtain services
  - **Delegate** resource management

• Memory management policy is completely in user-space

• **Isolation of physical memory** = **Isolation of authority (capabilities)**
  - Capability dissemination is controlled by a “Take-Grant” like protection model
Memory Management Model...

- De-allocation upon explicit user request
  - Call revoke on the Untyped capability
  - Memory can be reused

- Kernel tracks capability derivations
  - Recorded in capability derivation tree (CDT)
    - Need bookkeeping
      - Doubly-linked list through capabilities
      - Space allocated with capability tables
For allocation:

- The untyped capability should not have any CDT children
  - Guarantees that there are no previously allocated objects
- Size of the object(s) must be small or equal to untyped object
**Formal properties:**
- Formalised the protection model in Isabelle/HOL
  - Machine checked, abstract model of the kernel
- Formal, machine checked proof that mechanisms are sufficient for enforcing spatial partitioning
- Proof also identify the invariants the “supervisory OS” needs to enforce for isolation to hold
• **Formal properties:**
  - Formalised the protection model in Isabelle/HOL
    - Machine checked, abstract model of the kernel
  - Formal, machine checked proof that mechanisms are sufficient for enforcing spatial partitioning
  - Proof also identify the invariants the “supervisory OS” needs to enforce for isolation to hold

  • Can not share modifiable page/capability tables
  • Can not share thread control blocks
  • Can not have communication channels that allow capability propagation

IIES08/seL4
Evaluation ...

- **Performance**
  - Used paravirtualised Linux as an example
  - Compared with L4/Wombat (Linux) for running LMBench

<table>
<thead>
<tr>
<th>Bench mark</th>
<th>L4 (μs)</th>
<th>seL4 (μs)</th>
<th>Gain(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>4570</td>
<td>3083</td>
<td>32.5</td>
</tr>
<tr>
<td>exec</td>
<td>5022</td>
<td>3440</td>
<td>31.5</td>
</tr>
<tr>
<td>shell</td>
<td>29729</td>
<td>19999</td>
<td>32.7</td>
</tr>
<tr>
<td>page faults</td>
<td>34</td>
<td>18.7</td>
<td>45.4</td>
</tr>
<tr>
<td>Null Syscall</td>
<td>3.4</td>
<td>2.9</td>
<td>11</td>
</tr>
<tr>
<td>ctx</td>
<td>10.7</td>
<td>9.3</td>
<td>7.6</td>
</tr>
</tbody>
</table>
Conclusion

• No implicit allocations within the kernel
  – Users explicitly allocate kernel objects
  – No heap, slab .. (no hidden bookkeeping)
  – Authority confinement guarantees control of kernel memory

• All kernel memory management policy is outside the kernel
  – Different isolation/integration configurations
  – Support diverse, co-existing policies
  – No modification to the kernel (remains verified)

• Hard guarantees on kernel memory consumption
  – Facilitate formal reasoning of physical memory consumption

• Improve performance by controlled delegation
  – Similar performance in other case
Our Goal -- A Formally Verified Kernel

- A formally specified API (abstract model)
  - Proven properties such as spatial partitioning.
- High-performance implementation in C
  - A formal refinement between model and implementation
- Guaranteed correct implementation!!!
Success creates problems….

- Verified kernel is rigid
  - Changes invalidate proofs
- Kernel
  - “Mechanism not policy”
  - Flexible distribution of authority
    - Capabilities (KeyKOS/EROS)
  - Address spaces
    - Memory objects implemented at user level (L4/Xen/Nemesis)
  - Partitioning to hold within the kernel
    - No implicit memory allocation in kernel (Cambridge CAP)
  - Event-based/Atomic (Fluke)
Kernel Developers
Versus
Formal Methods Practitioners

Abstract Model
C Code
HW
Bridging the gap

Modelling?

• Well defined semantics
• Readily formalisable
• Exposed implementation details
• Programming language
Our Prototyping Approach

- Model the kernel in detail
- Literate Haskell to model
  - Pure functional programming language
  - Embedded documentation
  - Close to Isabelle/HOL
- Executable
  - Supports running user-level code
Kernel Modelling in Haskell

Kernel is event-based (single stack) mostly atomic API.

• Kernels are big state machines with events as input
  – Imperative
  – Rely on side-effects all the time
    • P(s), makeRunnable(tcb)

• Kernels manipulate the low-level machine
  – Interrupts, TLBs, caches

• Preemption required
  – Kernels can’t always perform operations to completion
Kernel Code in a State Monad

- State monads are *units* of computation which consume and produce state
  - State transformers
- Kernel monad encapsulates a state transformer of the kernel and machine

- Monads can be bound together using the *bind* operator
  - Sequencing the computation
  - Connects the plumbing to pass the state along
Kernel Code in a Monad

type Kernel = StateT KernelState MachineMonad
callKernel :: Event -> Kernel ()
callKernel ev =
  handleEvent ev >>= (\x -> schedule >>=
    (\y -> activateThread))

---

```
type Kernel = StateT KernelState MachineMonad
callKernel :: Event -> Kernel ()
callKernel ev =
  handleEvent ev >>= (\x -> schedule >>=
    (\y -> activateThread))
```
Kernel Code in a Monad

type Kernel = StateT KernelState MachineMonad
callKernel :: Event -> Kernel ()
callKernel ev = do
  handleEvent ev
  schedule
  activateThread

Imperative in “style”
Lowers barrier to entry for kernel developers
Kernel Monad

- Machine monad contains state to interface to simulator
- Kernel contains the state of physical memory
Machine Monad – Lowest Level of Model

- `getMemoryTop :: MachineMonad (PPtr ())`
- `getDeviceRegions :: MachineMonad [(PPtr (), Int)]`
- `loadWord :: PPtr Word -> MachineMonad Word`
- `storeWord :: PPtr Word -> Word -> MachineMonad ()`
- `insertMapping :: PPtr Word -> VPtr -> Int -> Bool -> MachineMonad ()`
- `flushCaches :: MachineMonad ()`
- `getActiveIRQ :: MachineMonad (Maybe IRQ)`
- `maskInterrupt :: Bool -> IRQ -> MachineMonad ()`
- `ackInterrupt :: IRQ -> MachineMonad ()`
- `waitForInterrupt :: MachineMonad IRQ`
- `configureTimer :: MachineMonad IRQ`
- `resetTimer :: MachineMonad ()`

- Foreign Function Interface (FFI)
- Approximate machine-level C functions
- Close to “real” as possible
  - Forces us to manage “hardware”
KernelState Monad

- Statically allocated global kernel data
  - Current thread
  - Scheduler queues
- Physical Memory
Kernel State Monad

- Physical memory model
  - Contents of dynamically-allocated memory
  - Typed data used by the kernel
    - Thread control blocks
    - Capability and page tables
    - ...
    - Indexed by physical memory address
  - Forces us to model memory management (30% of kernel)
  - Reduces the gap to C
    - Pointers, not Haskell’s heap
  - Still provides strongly typed pointers

![Diagram showing physical address space with CTE objects and TCB pointer.]
User-level Simulation

- User-level CPU simulator
  - M5 Alpha simulator
  - Locally-developed ARMv6 simulator
  - QEMU
- Executes compiled user-level binaries
- Sends events to the Haskell kernel
Prototyping Environment

Verification team identifying conceptual and verification issues.

Kernel team identifying conceptual and potential performance issues.

Exercisable Kernel API
+ User-level Instruction Simulator

Porting Iguana
Embedded OS
+ Other Stuff

L4.verified
seL4

Formal Verification of an OS Kernel

Gerwin Klein  Kevin Elphinstone  Gernot Heiser
June Andronick  David Cock  Philip Derrin
Dhammika Elkaduwe  Kai Engelhardt  Rafal Kolanski
Michael Norrish  Thomas Sewell  Harvey Tuch
Simon Winwood
1 microkernel
8,700 lines of C
0 bugs

qed

*conditions apply*
Small Kernels

Small trustworthy foundation

- hypervisor, microkernel, nano-kernel, virtual machine, separation kernel, exokernel ...

- High assurance components in presence of other components

seL4 API:
- IPC
- Threads
- VM
- IRQ
- Capabilities

Untrusted

Trusted

Legacy Apps

Sensitive App

Linux Server

Trusted Service

seL4

Hardware
The Proof
**Functional Correctness**

**What**

**Proof**

**How**

---

**Specification**

```
definition
  schedule :: unit x_read where
  schedule = do
    threads = all_activeTCDs;
    thread = select threads;
    switch_to_thread thread
    id
  block
  switch_to_idle_thread
```

```c
void
schedule(void) {
  switch ((word_t)ksSchedulerAction) {
    case (word_t)SchedulerAction_ResumeCurrentThread:
      break;

    case (word_t)SchedulerAction_ChooseNewThread:
      chooseThread();
      ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
      break;

    default: /* SwitchToThread */
      switchToThread(ksSchedulerAction);
      ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
      break;
  }
}
```
Assume correct:

- compiler + linker (wrt. C op-sem)
- assembly code (600 loc)
- hardware (ARMv6)
- cache and TLB management
- boot code (1,200 loc)
Implications

Execution always defined:
- no null pointer de-reference
- no buffer overflows
- no code injection
- no memory leaks/out of kernel memory
- no div by zero, no undefined shift
- no undefined execution
- no infinite loops/recursion

Not implied:
- “secure” (define secure)
- zero bugs from expectation to physical world
- covert channel analysis

© NICTA 2009
Proof Architecture

Isabelle

Proof

Specification

C Code
Proof Architecture

Access Control Spec

Specification

Confinement

Design

C Code

© NICTA 2009

void schedule(void) {
    switch ((word_t)ksSchedulerAction) {
    case (word_t)SchedulerAction_ResumeCurrentThread:
        break;

    case (word_t)SchedulerAction_ChOOSE_NEW_THREAD:
        chooseThread();
        ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
        break;

    default: /* SwitchToThread */
        switchToThread(ksSchedulerAction);
        ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
        break;
    }
}
States:
User, Kernel, Idle

Events:
Syscall, Exception, IRQ, VM Fault

System Model
Kernel Design for Verification
Design for Verification

Reducing Complexity

Hardware
- drivers outside kernel

Concurrency
- event based kernel
- limit preemption

Code
- derive from functional representation

```c
void schedule(void) {
    switch ((word_t)ksSchedulerAction) {
        case (word_t)SchedulerAction_ResumeCurrentThread:
            break;
        case (word_t)SchedulerAction_ChooseNewThread:
            ChooseThread();
            schedulerAction = SchedulerAction_ResumeCurrentThread;
            break;
    }
}
```
Everything from C standard

- **including:**
  - pointers, casts, pointer arithmetic
  - data types
  - structs, padding
  - pointers into structs
  - precise finite integer arithmetic

- **minus:**
  - goto, switch fall-through
  - reference to local variable
  - side-effects in expressions
  - function pointers (restricted)
  - unions

- **plus** compiler assumptions on:
  - data layout, encoding, endianness
Did you find any Bugs?

Bugs found

during testing: 16

during verification:
• in C: 160
• in design: ~150
• in spec: ~150
460 bugs

Effort

- Haskell design: 2 py
- First C impl.: 2 weeks
- Debugging/Testing: 2 months
- Kernel verification: 12 py
- Formal frameworks: 10 py
- Total: 25 py

Cost

- Common Criteria EAL6: $87M
- L4.verified: $6M
Summary

Formal proof all the way from spec to C.

- **200kloc** handwritten, machine-checked proof
- **~460** bugs (160 in C)
- Verification on **code, design, and spec**
- **Hard in the proof** → **Hard in the implementation**

Formal Code Verification up to 10kloc:

- It works.
- It’s feasible. (It’s fun, too.)
- It’s cheaper.
Werewolves beware - Silver Bullet!!!

Trustworthy software foundation – provably correct.

Next challenge is to complete the picture - **trustworthy systems!!!**

*conditions apply
Future research agenda - Our Approach

Leverage isolation guarantees of verified microkernel to achieve full-system guarantees while verifying only a few % of the code

Goal: system of >1M LOC, incorporating legacy components
Remaining Verification Issues:
  • information flow, temporal isolation, boot sequence, assembler
    - existing techniques known to provide some way out (to EAL7)
    - real solutions to information flow, temporal isolation are hard

Concurrency: need to verify multicore kernel
  • minimise shared data or minimise concurrency
  • technically and scientifically challenging

Device drivers
  • can be encapsulated safely
  • must be correct if shared

Security architecture:
  • minimise code and invariants that need verification
    - ideally generate architecture and associated proof
  • maximise automatic analysis

Overall system guarantee:
  • framework that allows composition of analysis & models
  • use different formalisms, map them into one common framework

Power management:
  • cross-cutting issue—interferes with info flow, temporal isolation
Summary

Achieved a software foundation of unprecedented trustworthiness – seL4
Challenge is leveraging seL4 to build trustworthy systems
  • evolving seL4
  • developing tools and techniques for system composition
  • domain-specific trustworthy components

Most likely demonstrator
  • Multi-level secure terminal

© NICTA 2009