seL4: Formal Verification of an OS Kernel

Limin Jia

Foundations of Security and Privacy
Nov. 19th, 2009
Foundations of Security and Privacy

- Secure communication
  - cryptography
  - protocol analysis

- Secure software
  - type-safety (high-level language)
  - certifying compilation (low-level assembly, machine code)

- Authentication and Access control

- Privacy

- Operating system
  - software
  - Network
seL4: Formal Verification of an OS Kernel

- Microkernel
- Verification techniques
- Protection model (capability model)
Microkernel – Least Privilege

Monolithic Kernel based Operating System

- Application
  - System Call
- VFS
- IPC, File System
- Scheduler, Virtual Memory
- Device Drivers, Dispatcher, ...

Hardware

Microkernel based Operating System

- Application IPC
- UNIX Server
- Device Driver
- File Server
- Basic IPC, Virtual Memory, Scheduling

Hardware

user mode

kernel mode
Verification of a Microkernel

- Goal: Functional correctness of the C implementation
  - Verification in conjunction with development

- Assumptions (TCB)
  - C compiler, assembly code, hardware
    - Detailed machine states are not verified
      (registers, cache consistency)
Verification of a Microkernel

- Assurances
  - Implementation correctness
    - As long as assumptions are satisfied, the kernel never crashes and satisfies invariants been proven
- Invariants
  - Low-level memory invariants
    - No null-pointer dereferencing, object separation
  - Typing invariants
    - Object reference points to object of the right type
  - Data structure invariants
    - Linked list
  - Algorithmic invariants
    - The global kernel memory containing kernel code and data is mapped in all address space.
Overview of Verification Techniques

- Use Hoare Logic to verify invariants
- Refinement proofs establish the correspondence between high-level and low-level
Abstract Specification

schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
  od OR switch_to_idle_thread
Haskell Implementation

```
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
      ...
    chooseThread = do
      r <- findM chooseThread' (reverse [minBound .. maxBound])
      when (r == Nothing) $ switchToIdleThread
    chooseThread' prio = do
      q <- getQueue prio
      liftM isJust $ findM chooseThread’’ q
    chooseThread’’ thread = do
      runnable <- isRunnable thread
      if not runnable then do
        tcbSchedDequeue thread
        return False
      else do
        switchToThread thread
        return True
```
C Implementation

```c
void setPriority(tcb_t *tptr, prio_t prio) {
    prio_t oldprio;
    oldprio = tptr->tcbPriority;
    if(thread_state_get_tcbQueued(tptr->tcbState)) {
        ksReadyQueues[oldprio] =
        tcbSchedDequeue(tptr, ksReadyQueues[oldprio]);
        if(isRunnable(tptr)) {
            ksReadyQueues[prio] =
            tcbSchedEnqueue(tptr, ksReadyQueues[prio]);
        }
    } else {
        thread_state_ptr_set_tcbQueued(&tptr->tcbState,
                                        false);
    }
    tptr->tcbPriority = prio;
}
```
Verification of a Microkernel

Isabelle/HOL

- Abstract Specification
- Executable Specification
- High-Performance C Implementation

Automatic Translation

- Refinement Proof
- Semantic Modeling

Haskell Prototype

C Implementation
Refinement – Process

- Process = (Init, Step, Fin)

```
S
```

Init

```
S_0
```

Step

```
S_{01}
```

```
S_{02}
```

```
S_{011}
```

```
S_{012}
```

```
S'
```

```
S''
```

```
S_1
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```
Refinement

- Process A is refined by process C
  - if given the same external state and event sequence, the execution of C yields a subset of the external states yielded by executing A
Refinement

- Forward simulation
- Traces in a concrete implementation are mapped to a subset of the traces in the abstract model
Refinement Proofs in SeL4

- Hoare Logic (review)
  - Hoare Triple \{Pre\} code \{Post\}
  - Soundness of Hoare Triples
    - If \( S \) satisfies Pre
      and \((S, \text{code}) \rightarrow (S', \text{code}')\)
      then \( S' \) satisfies Post

- Corr \( R \) \( P \) \( P' \) \( c \) \( c' \) if
  - \{\( P \}\} \( c \) \{\( Q \}\)
  - \( P \sim_R \) \( P' \)
  - exists \( Q' \), \{\( P' \}\} \( c' \) \{\( Q' \}\) and \( Q \sim_R \) \( Q' \)
Verification of a Microkernel

\[ M_C \text{ refines } M_E, \quad M_E \text{ refines } M_A \implies M_C \text{ refines } M_A \]
seL4: Formal Verification of an OS Kernel

- Microkernel
- Verification techniques
- Protection model (capability model)
Protection Model

- Object-capability model

- Access to kernel resources (memory, interfaces) are protected by capabilities

- Capability is an unforgeable token of authority
  - E.g. file handle in UNIX system
  - Name of the resource + access rights \{R, W, C, G\}
  - Stored in protected data structures
**Capability Model**

**Datatype**  
right = R | W | G | C

**Record**  
capability = {resource: entity_id, rights: Set right}

tentity_id = Int

**Record**  
entity = {caps : Set capability}
Isolation of Authority

• Given a sane state $S$, a non-empty subsystem $ss$ in $S$, and a capability $c$ with a target identity $e$ in $S$, if the authority of the subsystem does not exceed $c$ in $S$, then it will not exceed $c$ in any future state of the system.
Capability Model

- Leak
  - $S \vdash e_x \rightarrow e_y$
    - $\{\text{resource} = e_y, \text{rights} = \{G\}\} \in e_x\text{.caps}$

- Subsystem
  - $e_x$ and $e_y$ are in the same subsystem if $S \vdash e_x \leftrightarrow^* e_y$

- Operations cannot connect disconnected entities.
Capability Model

- `create e n c_1 c_2`
  - Entity `e` creates a new entity using free memory provided by `e_1` and assigns the new cap for controlling access to `n` to `e_2`
Capability Model

- grant e c₁ c₂ R
  - Entity e gives a copy of the capability e₂ (restricted to the subset R) to the entity of c₁

\[ G \in c₁.\text{rights} \quad e₁ \quad e₂ \]

\[ e₁ \text{ may leak to } e₂ \text{ if } G \in c₂.\text{rights} \cap R, \text{ however, in this case } e₁ \text{ and } e₂ \text{ are already in the same subsystem} \]
Capability Model

- remove $e c_1 c_2$
Capability Model

- revoke e c
Protection Model

- Single step does not increase capabilities
- Isolation of authority between subsystems
Verification of a Microkernel

-isabelle/HOL-

isoaltion  Abstract Specification  Executable Specification  High-Performance C Implementation

Automatic Translation
Refinement Proof
Semantic Modeling
Haskell Prototype
C Implementation

Figure 2: The correspondence established by the refinement proof.

The property we are proving is functional correctness, meaning that if a security property is proved in Hoare logic, it holds for the kernel source code. In this paper, we concentrate on the general functional correctness property of the kernel.

The security of seL4's access control system is based on formal proof such as static analysis or model checking. Interactive theorem proving requires human interaction, whereas automated methods of proof are less constrained to specific properties or finite state spaces. Unlike more automated methods, interactive proof offers opportunities to micro-optimize the kernel, which is crucial for adequate microkernel performance.

Specifically, we use the theorem prover Isabelle/HOL required for adequate microkernel performance. The correspondence between a high-level abstract model and the refined operational model that is the main interest of the refinement proof is shown in the figure. The translation is manual and the proof requires human intervention.
### Summarize

- Verification of functional correctness of the implementation of a Microkernel
  - Hoare triples to verify invariants
  - Refinement proofs
- Capability model
  - Isolation of authority

<table>
<thead>
<tr>
<th></th>
<th>Haskell/C LOC</th>
<th>Isabelle LOC</th>
<th>Invariants</th>
<th>Proof LOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>abst.</td>
<td>—</td>
<td>4,900</td>
<td>~ 75</td>
<td>110,000</td>
</tr>
<tr>
<td>exec.</td>
<td>5,700</td>
<td>13,000</td>
<td>~ 80</td>
<td>55,000</td>
</tr>
<tr>
<td>impl.</td>
<td>8,700</td>
<td>15,000</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>