Overview

• How to build verifiably secure systems is an active area of research
• Two case studies
  – SeL4: How to build a provably secure microkernel
  – XMHF: Compositional verification of a hypervisor framework
• Challenges
• Objectives: Food for thought
The big picture

• Static analysis
  – Post-compilation analysis
    • Model checking is great, but cannot check complex properties (previous lectures)
    • Proof-based techniques can verify complex properties, but need manual effort (this lecture)
  – Compile time analysis
    • Type checking (future lectures)
      – Minimal annotation (type declarations)
      – Reasonable guarantees
      – Correct by construction
PART I: SeL4
SeL4 — a Microkernel

- Least privilege
- Small code base
  - 8700 Loc
Verification of a Microkernel

• Goal: Functional correctness of the C implementation
  – Verification in conjunction with development
Verification of a Microkernel

• Assurances: Implementation correctness
  – 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
    • Specify how SeL4 kernel should work
    • E.g. The global kernel memory containing kernel code and data is mapped in all address space
Verification of a Microkernel

- Assurances: Implementation correctness
  - 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
  - Algorithmic invariants
    - Specify how SeL4 kernel should work
    - E.g. The global kernel memory containing kernel code and data is mapped in all address space

Security?

- No buffer overflows
- No unchecked user arguments
Overview of Verification Techniques

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

\[
\text{schedule} \equiv \text{do} \\
\quad \text{threads} \leftarrow \text{all_active_tcbs}; \\
\quad \text{thread} \leftarrow \text{select threads}; \\
\quad \text{switch_to_thread thread} \\
\text{od OR switch_to_idle_thread}
\]
Haskell Implementation

```haskell
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
```
void setPriority(tcb_t *tptr, prio_t prio) {
    prio_t oldprio;
    if(thread_state_get_tcbQueued(tptr->tcbState)) {
        oldprio = tptr->tcbPriority;
        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 Proofs

• Definition of program C refines program A
  – Program, execution of programs

• How to prove refinement
Definitions – states, programs

• Programs change the machine states
  – Example: sorting program change an array of unsorted elements to a sorted one

• Machine states (memory)
  – External states are observable to all programs
  – Internal states are local to each program
Definitions – Programs

• Program = (Init, Step, Fin, Σ)
• execute(A, s, {e₁, … eₙ}, Σ) = (sₐ₁, … sₐᵢ)
Refinement (C refines A)
Refinement

- Process C refines process A if for all \( s, e_1, \ldots, e_n \),
  - \( \text{execute}(A, s, \{e_1, \ldots, e_n\}, \Sigma_A) = (s_{A1}, \ldots, s_{Ai}) \)
  - \( \text{execute}(C, s, \{e_1, \ldots, e_n\}, \Sigma_C) = (s_{C1}, \ldots, s_{Cj}) \)
  - \( (s_{C1}, \ldots, s_{Cj}) \subseteq (s_{A1}, \ldots, s_{Ai}) \)
Proof of Refinement

- Forward simulation
  - Examining one possible execution from s to s’
  - Given any execution from s to s’ in the concrete system, there exist a corresponding execution in the abstract system
Proof of Refinement

• Forward simulation
  – \( C = (\text{Init}_C, \text{Step}_C, \text{Fin}_C, \Sigma_C) \), \( A = (\text{Init}_A, \text{Step}_A, \text{Fin}_A, \Sigma_A) \)
  – There is a mapping \( f \) from \( \Sigma_C \) to \( \Sigma_A \) such that
    • (R1) \( f(\text{Init}_C) \subseteq \text{Init}_A \)
    • (R2) \( \text{external}(f(s_C)) = \text{external}(s_A) \)
    • (R3) if \( (s_C, s'_C) \in \text{Step}_C \) and \( f(s_C) = s_A \)
      then there exists \( s'_A \) such that
      \( s'_A = s_A \) or \( (s_A, s'_A) \in \text{Step}_A \) and \( f(s'_C) = s'_A \)

\[ \exists \quad S \quad \rightarrow \quad S_0 \quad \rightarrow \quad S_1 \quad \rightarrow \quad \ldots \quad \rightarrow \quad S' \]

\[ \forall \quad S \quad \rightarrow \quad S_0 \quad \rightarrow \quad S_1 \quad \rightarrow \quad \ldots \quad \rightarrow \quad S' \]
To Show: ∀ s s_f, C reaches an external state s_f from s ⇒ A can reach f(s_f) from f(s).

Queue Q

(T, Q’) = select Q

Queue Q’

Thread T

• Q is a thread Queue
• Q’ is a thread Queue
• Q contains T and Q’
• T is None if Q is empty

Abstract spec: (Q = t_1, t_2..., t_n)
(T, Q’) = select Q

C implementation:
void select (*dlist Q, *dlist T) {
T = Q;
Q = Q->next;
T->next = null;
Q->prev = null;}

Simulation:
(R1) f(Init_C) ⊆ Init_A
(R2) external(f(s_C)) = external(s_A)
(R3) if (s_C, s’_C) ∈ Step_C and f(s_C) = s_A
then there exists s’_A such that
s’_A = s_A or (s_A, s’_A) ∈ Step_A and f(s’_C) = s’_A
**ToShow:** $\forall \ s \ s_f,$
C reaches an external state $s_f$ from $s$
$\implies$ A can reach $f(s_f)$ from $f(s)$.

**Simulation:**
(R1) $f(Init_C) \subseteq Init_A$
(R2) $external(f(s_C)) = external(s_A)$
(R3) if $(s_C, s'_C) \in Step_C$ and $f(s_C) = s_A$
then there exists $s'_A$ such that
$s'_A = s_A$ or $(s_A, s'_A) \in Step_A$ and $f(s'_C) = s'_A$

- Q is a thread Queue
- Q contains T and Q’
- T is None if Q is empty

**Abstract spec:** $(Q = t_1, t_2..., t_n)$
$(T, Q') = select \ Q$

**C implementation:**
void select (*dlist Q, *dlist T) {
T = Q;
Q = Q->next;
T->next = null;
Q->prev = null;}

f: (1) Queue Q’s elements are stored in a doubly-linked list Q
(2) Thread T is the element in the list node T
**ToShow:** $\forall \ s \ s_f,$

C reaches an external state $s_f$ from $s$

$\Rightarrow$ A can reach $f(s_f)$ from $f(s)$.

---

**Queue** $Q$

$Q'$ is a thread Queue

$Q$ contains $T$ and $Q'$

$T$ is None if $Q$ is empty

---

Abstract spec: $(Q = t_1, t_2, \ldots, t_n)$

$(T, Q') = \text{select } Q$

---

C implementation:

```c
void select (*dlist Q, *dlist T) {
    T = Q;
    Q = Q->next;
    T->next = null;
    Q->prev = null;
}
```

---

Simulation:

(R1) $f(\text{Init}_C) \subseteq \text{Init}_A$

(R2) $\text{external}(f(s_C)) = \text{external}(s_A)$

(R3) if $(s_C, s'_C) \in \text{Step}_C$ and $f(s_C) = s_A$

then there exists $s'_A$ such that $s'_A = s_A$ or $(s_A, s'_A) \in \text{Step}_A$ and $f(s'_C) = s'_A$

---

Core dump

Not a behavior allowed by the spec!
Verification of a Microkernel

Isabelle/HOL

\[ \text{MC refines ME, ME refines MA} \rightarrow \text{MC refines MA} \]
Verification of a Microkernel

M, refines ME, ME refines MA \rightarrow MC refines MA
Verification of a Microkernel

Abstract Specification

EXEC

High-Performance C Implementation

Isabelle/HOL

Automatic Translation

Refinement Proof

We only need to prove properties of the abstract model

$M_C$ refines $M_E$, $M_E$ refines $M_A \rightarrow M_C$ refines $M_A$
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)
Summary

• Verification of functional correctness of the implementation of a Microkernel
  – Refinement proofs

<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>
Verification of a Microkernel
Compositional security

- How to model the adversary?
- How to reason about a composed system?

\[ S_1 \psi_1 + S_2 \psi_2 \]

Applications
- SeL4 binary
- Hypervisor
- Hardware

Verifies that it has no buffer overrun

 Writes to SeL4 code region at Application’s request
PART II: XMHF
Virtual Machine Monitors (hypervisors)

• Provide hardware virtualization for multiple Oses
• Type 1 hypervisor
  – Directly sits on top of the hardware
Virtual Machine Monitors (hypervisors)

- Provide hardware virtualization for multiple Oses
- Type 2 hypervisor
  - Runs on top of an OS
Applications of hypervisors

• Run multiple Oses
• Security applications
  – Provide separation
  – Reliable resource allocation

- Code integrity
- Data secrecy
- Resource accounting
Motivation for building XMHF

• Hypervisor-based security architectures are becoming popular

• An extensible verifiable hypervisor framework
  – Extensibility
    • A core module provides key hypervisor functionalities
    • Hypervisor-based security applications are extended modules
  – Verifiability
    • Modular design facilitates compositional reasoning
Architecture of XMHF

- Type-1 (bare-metal)
- Core is shared
- Extensible through hypapps
  - Register event handlers to implement additional functionalities
Memory integrity property

- Hypervisor’s memory (code and data) is only written to by hypervisor’s code
- Memory protected by
  - Hardware virtualization
  - Nested Hardware Page Table
  - Set hypervisor memory region to be read only
- **What could go wrong:**
  - Transfer control to guest without starting VM
  - Forget to protect HPT
  - HPT is not set up correctly
How to prove memory integrity?

- Attacker model
- Hypervisor model
- Hardware model
Modeling adversary

• Interface-confined adversary
  – Adversary is arbitrary code confined to a set of well-defined interfaces
  – E.g. sandboxed code has access to limited APIs

• Our adversaries
  – Guest OS
    • confined by the hardware protection mechanisms (Guest mode, NPT)
  – Device DMA
    • MMU
Reasoning about adversary

• Interface-confined adversary
  – Adversary is arbitrary code confined to a set of well-defined interfaces
  – E.g. sandboxed code has access to limited APIs

• Only need to examine the effects of the interfaces
  – If all the interfaces that the guest OS has access to cannot write to hypervisor memory, then guest OS cannot modify hypervisor memory
Proof sketch

I. When guest runs
   I. CPU mode is GUEST
   II. NPT marks hypervisor mem is read only
   III. NPT itself is read only

II. When control transfers to HOST
   I. Jump to a code location stored in a designated memory region

III. When hypervisor runs
   I. Maintain the same NPT permissions
   II. Does not modify it’s own code
   III. Only transfers to guest using vmresume

<table>
<thead>
<tr>
<th>Event</th>
<th>Event handler</th>
<th>Event handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>VM exit</td>
<td>Guest runs</td>
</tr>
<tr>
<td>Guest</td>
<td>Hypervisor runs</td>
<td>Guest runs</td>
</tr>
<tr>
<td>runs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VM exit</td>
<td></td>
<td>Guest runs</td>
</tr>
<tr>
<td>resume VM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DMA

Guest operating system

Hypervisor core

Platform devices
Rely-guarantee reasoning

1. $\varphi$ holds initially

2. If $\varphi$ holds before adversary code executes, then $\psi_A$ holds when it makes a transition

3. $\psi_1, \psi_2, \psi_A$ implies $\varphi$

Goal: $\varphi$ holds always
Overview

Properties

Implementation

Manual audit

Verification (automated)

Hypervisor model

Adversary model

Assumptions on hardware

Proof that the hypervisor satisfies memory integrity, interacting with untrusted guest OS

A novel program logic
Verification local properties of XMHF core

• Memory integrity of XMHF core
  – 6018 SLoC, largely automated verification
• Automated verification = 5208 lines of C
  – CBMC
  – Less than 2GB memory and 80 seconds
• Manual audits = 422 lines of C + 388 asm. instructions
  – Likely to remain unchanged as development proceeds
Summary

• eXtensible Modular Hypervisor Framework
  – Extensibility
  – Verifiability
  – Good performance

• Designed a sound program logic (type system) to construct the top-level proof

• What’s missing
  – Refinement proofs to link the model to implementation
  – Hardware modeling
Take away

• Verification of real software
  – Microkernel
  – Hypervisor
• Simulation (refinement) techniques
• Rely-guarantee reasoning
• Interface-confined adversaries
• Compositional reasoning is key (challenging)
The big picture

• Static analysis
  – Post-compilation analysis
    • Model checking is great, but cannot check complex properties (previous lectures)
    • Proof-based techniques can verify complex properties, but need manual effort (this lecture)
  – Compile time analysis
    • Type checking (future lectures)
      – Minimal annotation (type declarations)
      – Reasonable guarantees
      – Correct by construction