Formal Analysis of a Tiny Hypervisor

Jason Franklin
Overview

- **Goal:** Develop techniques to reason about assurance level of secure systems

- **Security mechanisms include:**
  - Memory protection and cryptographic protocols

- **Security properties include code or data integrity:**
  - Safety properties

- **Examples:** OSes, hypervisors, and VMMs
How is verification for security different?

- Limitations of related work in verification:
  - No adversary
  - Focus on checking correctness of security critical code rather than security (i.e., correctness in presence of adversary)
  - Correct system components do not imply secure system
Analysis of Secure Systems

- **Design level analysis**
  - Pros: Before implementation of possibly insecure design
  - Cons: May miss attacks in implementation
  - Techniques: Logics and *model checking*

- **Implementation level analysis**
  - Pros: Source code is closer to what is actually run
  - Cons: Much more complicated
  - Techniques: Software model checking
Hypervisor-Protected Systems

Traditional System Architecture

- App.
- App.
- OS
- Hardware

Hypervisor-Protected System Architecture

- App.
- App.
- Protected OS
- Hardware

Tiny security hypervisor provides additional layer of protection, if hypervisor is secure.
Claim: Hypervisors are easier to secure than traditional operating systems for two reasons:
- Hypervisors can be written in few lines of code
  - They’re small.
- Hypervisors expose a narrow interface
  - They’re simple.

Let’s test this hypothesis…
Case Study: SecVisor Hypervisor

- SecVisor security hypervisor (3K loc, 2 calls)
  - Only approved code executes with kernel privilege
Outline

- We model and analyze SecVisor’s design
- Find and repair vulnerabilities
- Verify security of repaired design
- Extend verification to arbitrarily large model
How SecVisor Works

Hardware State
- Mode = \{Kernel, User\}
- Program Counter (IP)
- Physical Memory
- Device Exclusion Vector

Memory Protection
- User Mode:
  - RWX
  - R
  - RW
- Kernel Mode:
  - RW
  - RX
  - RW

User Mem.
Kernel Code
Kernel Data
SecVisor and HW Environment
Operational Behavior

Write(0x00, “hello”);

0x01 = VA_to_HPA(0x00)

Write(0x01, “hello”);

00: RW 01
01: X 10
10: RW 00

00: RW 01
01: X 10
10: RW 00

Synch(KPT, SPT)

Phy. Mem.

“hello”
Modeling State in Murphi

hardware_platform_t: record
  phy_mem : phy_mem_t;
  mode: bit;
  IP: word;
  DEV: dev_t;
end;

mode = K;
IP = KC;

State 1

Hardware State
- Mode = \{Kernel, User\}
- Program Counter (IP)
- Physical Memory
- Device Exclusion Vector
pte_t: record -- Page Table Entry
    rw: bit; -- Read/Write Perm
    x: bit; -- Execute Perm
    PA: word -- Physical Address
end;

mode = K;
IP = KC;
pgtbl = [...]

State
Security Defined by Simulation

page_table_t: array[index_PTEs] of pte_t;
hw: hardware_platform_t;

-- Actual World
SPT: page_table_t; -- Shadow PT
KPT : page_table_t; -- Kernel PT

--Ideal World
sec_SPT: page_table_t; -- Shadow PT
rule “Kernel Exit”
  hw.mode = KERNEL_MODE ==> setIP(USER_MEM);
end;

rule “Kernel Entry”
  hw.mode = USER_MODE ==> secvisor_kernel_entry();
end;

State 1
  mode = K;
  IP = KC;

State 2
  mode = U;
  IP = UM;
procedure secvisor_init_kernel_mode();
begin
  -- Start in kernel mode
  hw.mode := KRNL_MODE;
  hw.IP := KRNL_CODE;

  -- Init. DEV to protect kernel code
  hw.DEV[KRNL_CODE] := 1;

  -- 1) Initialize actual model PTs
  SPT[USER_MEM].rw := RW;
  SPT[USER_MEM].x := NON_eXe;
  SPT[USER_MEM].PA := USER_MEM;
Modeling Transitions in Murphi

-- Kernel code permissions RX
SPT[KRNL_CODE].rw := R;
SPT[KRNL_CODE].x := eXe;
SPT[KRNL_CODE].PA := KRNL_CODE;

-- Kernel data permissions RW
SPT[KRNL_DATA].rw := RW;
SPT[KRNL_DATA].x := NON_eXe;
SPT[KRNL_DATA].PA := KRNL_DATA;
Modeling Transitions in Murphi

-- Init ideal model SPT
-- Set user memory perms to RW
sec_SPT[USER_MEM].rw := RW;
sec_SPT[USER_MEM].x := NON_eXe;
sec_SPT[USER_MEM].PA := USER_MEM;

...
-- User to kernel mode entry handler
procedure secvisor_kernel_entry();
begin

-- Kernel mode
hw.mode = KRNL_MODE;
hw.IP := KRNL_CODE;

-- A) Set actual model PTs
-- 1. User memory permission RW ...
-- B) Set ideal model SPT
...

procedure secvisor_spt_synchronize();
begin
SPT[KRNL_CODE].PA := KPT[KRNL_CODE].PA;
SPT[KRNL_DATA].PA := KPT[KRNL_DATA].PA;
SPT[USER_MEM].PA := KPT[USER_MEM].PA;
end;
rule "Attacker writes through MMU"
true ==> write();
end;

procedure write();
begin
for i : index_PTEs do
  -- If SPT entry is writable
  if (SPT[i].rw = RW) then
    -- Modify page integrity bit
    hw.phy_mem[SPT[i].PA] := 1;
  endif
endfor
end;
ruleset va: index_PTEs do
ruleset pa: page_index do
ruleset rw: bit do
ruleset x: bit do
  rule "Modify KPT entries"
    true ==> begin
      KPT[va].rw := rw;
      KPT[va].x := x;
      KPT[va].PA := pa;

      secvisor_synchronize();
    end;
endruleset; endruleset; endruleset; endruleset;
Specify security property as invariant (holds in all states):
- Let property $P = (\text{Mode} = K \implies IP = KC)$
- $hw.\text{mode} = \text{KRNL\_MODE} \implies \text{sec\_SPT[sec\_SPT[hw.IP].PA].x} = \text{SPT[SPT[hw.IP].PA].x}$

Verify (Model $M$, State $S$, Property $P$)
Check property in state $S$
If property does not hold, then print counter-example and exit
If (more neighbors exist)
Verify ($M$, Neighbors($S$), $P$)
else return VERIFIED

State 1
mode = K;
IP = KC;

Attacker Step

State 2
mode = K;
IP = UM;
Initial verification failed:

- Counterexamples identified two vulnerabilities:
  - Writable virtual alias attack
  - Approved page remapping attack

Diagram:
- Adv → W, A -> KC
- Synch(KPT, SPT)
- Write (A, mal.code)
- Mal. Code
- Phy. Mem
Successful Verification

- After adding checks in synchronization code, verification succeeded
  - No property violations found in small models

<table>
<thead>
<tr>
<th>Model Size</th>
<th>States</th>
<th>Transitions</th>
<th>Time</th>
<th>Memory</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 PTEs, 3 Mem.</td>
<td>~55,000</td>
<td>~2,000,000</td>
<td>2.52 Sec.</td>
<td>8MB</td>
<td>Success</td>
</tr>
<tr>
<td>4 PTEs, 3 Mem.</td>
<td>~1,700,000</td>
<td>~88,000,000</td>
<td>5.75 Mins.</td>
<td>256MB</td>
<td>Success</td>
</tr>
</tbody>
</table>
Limitations of Small Models

- We ran out of memory after 3 PTEs and 3 memory pages
  - Memory requirements grow exponentially (state explosion)

- Do attacks exist when system has many more PTEs and memory pages?

- Murphi can’t check realistically sized machine models
  - E.g., $2^{20}$ memory pages and $2^{20}$ PTEs in both KPT and SPT

- Even if it could, what if attacker has $2^{20} + 1$?
To infinity and beyond! (sorta)

- We prove SecVisor is secure with an arbitrarily large but finite number of PTEs and physical memory pages
- Let’s look at SecVisor from different perspective
Translation to Small Model

- Translate functional model to parameterized model
  - Mem. pages and PTEs become duplicated homogeneous processes
- Prove template model is reducible to small model
**Small World Theorem:** If security property $P$ is violated in process model with arbitrarily large (but finite) number of processes then it will be violated in small model.

Implication: It is sufficient to model check only small model (completeness)

---

**Functional Model**

**Process Model**

**Small Model**

---

**Successful Verification**

**Simulation Theorem**

---

**Successful Verification**

**Successful Verification**

---

**Translate**

**Reduce to**

---

**Process Template**
Conclusion

- We employed model checking to increase assurance level of SecVisor hypervisor

- Found and repaired vulnerabilities in SecVisor’s design and implementation

- Verified repaired design model up to 3 PTEs

- Extended verification result to large model
  - Using small world theorem and simulation
References