Implementing an Untrusted Operating System on Trusted Hardware

David Lie
Chandramohan A. Thekkath
Mark Horowitz

University of Toronto, Microsoft Research, and Stanford University
Protection in Systems: Hardware Approach

- Many platforms exist where trust is placed in hardware:
  - Trust Computing Platform Alliance (TCPA)
  - Microsoft’s Palladium (NGSCB)
  - Intel LaGrande Architecture

- Advantages of hardware approach
  - Better tamper-resistance
  - Less dependent on OS correctness for security

- Trust in hardware means less trust (or no) in OS
  - We need to rethink operating system design
Rethinking the Role of the OS

• A traditional OS performs both resource management and protection for applications
  – But now applications and OS are mutually suspicious
  – But applications don’t trust OS to access their code and data
  – OS must be able to interrupt applications

• Use XOM (eXecute Only Memory) as an example platform
  – New operating system is called XOMOS

• XOMOS is UNIX-like, port of IRIX 6.5
  – Should support most standard UNIX applications
XOM Hardware Architecture

- XOM is implemented as a set of ISA extensions

- XOM Processor implements compartments for protection
  - Each compartment has a unique XOM ID
  - Architectural tags to control access to data on-chip
  - Cryptographic mechanisms protect data off-chip

- XOM Processor executes encrypted code
  - Combination of asymmetric/symmetric ciphers
  - Private key embedded on-chip hardware
Entering Secure Execution

- Applications Enter and Exit secure execution via instructions

**Main Memory**

- Unencrypted Code
- Encrypted Code (sym)
- Encrypted Symmetric Key

**XOM Processor**

- Asymmetric Decryption Module
- Private Key
- XOM Key Table
- Executable Code
- Symmetric Decryption Module

**Currently executing**

- XOM ID and Key
- Encrypted Symmetric Key
- Executable Code
- Unencrypted Code
- Encrypted Code (sym)
Implementing Compartments

- Problem: How does OS save registers during a trap or interrupt?
XOMOS Overview

- Implement XOM CPU in SimOS hardware simulator
  - MIPS-based architecture with XOM ISA extensions

- Ported the IRIX 6.5 Operating System to run on XOM processor

- Main areas that need modification:
  - Resource management of secure data
    - Saving Registers on Interrupt, Memory Management
  - Need support for XOM Key Table
    - Loading/Unloading, Management
  - Compatibility with original system
    - Fork, Signal Handling, Shared Libraries/System Calls
Saving Registers

- OS uses *save register* instruction

```
Data | Tag

Look up program key based on Tag

Hash | Data

Encrypt Data

Store encrypted data and hash to memory

OS XOM ID

XOM Key Table

Enlarge the exception frame to store hash and XOM ID
```
Protecting and Managing Memory

- Applications use *secure store* instruction

**Registers**

<table>
<thead>
<tr>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
</table>

**Caches**

Secure Store:
Tag is copied from register to cache

<table>
<thead>
<tr>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
</table>

**Memory**

Writeback:
Look up Tag, Encrypt and Hash

| Hash | Data |

Relocate data and hashes together

XOM Key Table
Replay Attacks and Fork

- An OS Fork must be differentiated from a Replay Attack
  - Replay Attack: OS duplicates registers and replays them
  - Fork: OS must duplicate register set and restore them

- Revoke and regenerate register keys on every interrupt
- This hardware defense breaks traditional fork code
Naïve Fork Implementation

- Parent and Child are exactly the same, have the same XOM ID
- Problem: Both threads use the same register key

Fork

Parent

Child

Register Key

XOM Processor
Fork Solution

- The OS must allocate a new XOM ID for the child process.
  - The child process has a separate XOM ID and register key.

Pre-fork:
Allocate a new XOM Key Table Entry

Fork:
Child uses one XOM ID, the Parent uses other

Child
XOM ID

Parent
XOM ID

Process
XOM ID

Process
XOM ID
XOM ID

Process
XOM ID
XOM Performance Simulation

• The SimOS Simulator:
  – Performance modeling processor, caches, memory and disk

• Simulation Parameters
  – Memory latency increases 10% due to encryption
    • Based on AES (Rijndael) cipher implementation
  – Public key decryption: 400,000 cycles
    • Once per process creation
Operating System Overhead

<table>
<thead>
<tr>
<th></th>
<th>Total Cycles</th>
<th>Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IRIX</td>
<td>XOM</td>
</tr>
<tr>
<td>System Call</td>
<td>5.7K</td>
<td>6.3K</td>
</tr>
<tr>
<td>Signal Handling</td>
<td>31K</td>
<td>40K</td>
</tr>
<tr>
<td>Fork</td>
<td>138K</td>
<td>126K</td>
</tr>
</tbody>
</table>

- OS overhead due to poor cache behavior:
  - More cache misses due to bigger code/data footprint
  - Should check to see if OS needs to do extra work
    - Work is only needed if application is in compartment
End-to-end Application Overhead

- Applications use Enter/Exit XOM to execute sections securely
  - How large to make secure sections?
  - Executing in compartment has memory access overhead
  - Entering/leaving compartment can cause cache misses

- Different compartment sizes: granularities
  - 3 levels: Coarse, Fine, Super-Fine
  - Applications are MPEG Decode and RSA Encryption
Application Overhead

- Turns out that granularity has little impact on performance overhead!
- Overheads for most applications turn out to be less than 5%
  - If cache behavior is similar
- Coarse application has smallest effect on cache behavior
Conclusions

• Possible to construct an Untrusted Operating System
  – Modest modifications
    • Port required 1900 lines of low level code
  – Modifications limited to low level interface to the hardware
    • Able to support most traditional OS mechanisms

• Performance hit is modest (< 5%)
  – Can run applications completely inside compartment
    • As long as cache behavior is unaffected
  – This is because of hardware acceleration