Architectural Support for Copy and Tamper-Resistant Software

David Lie

Computer Systems Laboratory
Stanford University
Resistance is not Futile

• Tamper-Resistant software can be used to
  – Prevent Software Piracy
  – Protect Intellectual Property
  – Prevent tampering by Viruses, Hackers
  – Can ensure that program performs certain actions

• Other initiatives starting:
  – Trusted Computing Platform Alliance (TCPA)
  – Microsoft Next-Generation Secure Computing Base (Palladium)
  – Intel LaGrande
Why Tamper-Resistance?

• Example: Electronic Online Voting

Internet  User

Voting Client

No!

Voting Client

Operating System

Vote Counter

Yes!
XOM

- **Our solution: eXecute Only Memory or “XOM”**
  - Programs in this memory can only be executed, they cannot be read or modified
  - Program authentication
  - Hide secrets in the program

- XOM combines cryptographic and architectural techniques
  - Access Control tags are fast but not necessarily secure
    - Only used on the trusted hardware of the processor
  - Cryptography is slow but offers more guarantees
    - Used to protect data that has to be stored off the processor

- XOM defends against attacks on memory
Compartments

- Compartments control access to data
  - Prevent adversary from reading data or code
  - Prevent adversary from modifying data or code

- Compartments provide a way of thinking about how data is handled
Where to Implement Compartments

- User Level security is hard
  - Relies on software obfuscation
  - Barak et. al. CRYPTO 2001

- Operating system can’t be trusted
  - OS can be open source
  - OS can be hacked or hijacked

- Hardware has some good security properties
  - Hardware is hard to observe
  - Hardware is difficult to alter
How to Implement Compartments

• Each compartment has a XOM ID
  – Programs are assigned a XOM ID, which indicates what compartment they are in
  – Data from their operations is tagged with their XOM ID
  – On-chip storage for data and tags is immutable

• Off-chip storage, memory and disk are insecure
  – Cannot by protected by tags
  – Cryptographic ciphers and hashes are used
Crypto Review

• Symmetric Ciphers (Rijndael, 3DES)
  – Single key used for encryption and decryption
  – Pretty fast when implemented in hardware

![Diagram showing the process of symmetric ciphers]

• How do we securely distribute the keys?
Crypto Review

- Asymmetric Ciphers or public-key ciphers (RSA, El Gamal)
  - Pairs of keys
  - Public key and is used to encrypt data
  - Private key and is used to decrypt data

- Public-key ciphers are very slow
  - Typically use hybrid systems with both ciphers together
Outline

1. Introduction
2. XOM Hardware
   i. Distributing and Loading Code
   ii. Executing Code
   iii. Supporting Memory
   iv. Hardware Simulator
3. Operating System Support
4. Attack Models
5. Conclusion
Computer Hardware

Bus to Memory

L2 Cache

L1 Instruction Cache

Private Key

Key Table

Fetch and Decode

L1 Data Cache

Register File

Data Path
Software Distribution Method

Distributor

Randomly Selected Symmetric Key

Program Code

Encrypt Program

Customer sends public key

Symmetric key is encrypted with public key

Encrypted Code

Customer receives encrypted code with encrypted key

Encrypted Code

Customer wishes to purchase software
Loading Secure Code

- Encrypted Code
- Encrypted Symmetric Key
- Insecure Main Memory
- Secure XOM Machine
- Asymmetric Decryption Module
- Decrypted Symmetric Key (Compartment Key)
- Private Key
- XOM Key Table
- Symmetric Decryption Module
- Executable Code

Secure Execution Engine
Supporting Execution

Bus to Memory

L2 Cache

- L1 Instruction Cache
- L1 Data Cache

Private Key

Key Table

Fetch and Decode

Register File

Data Path

XOM Tags
Secure Execution Engine

Program 1
Register 1
Data  Tag 1
Register 3
Data  Tag 1

Program 2
Register 2
Data  Tag 2
Register 3
Data  Tag 2

Ownership Tags

Secure XOM Machine
Compartments
How to Share Data

- Compartments are too restrictive, programs cannot share data

- Have a special compartment with a XOM ID of zero called the “null” compartment
  - Data in this compartment is not protected by any mechanism

- Special instructions provided for owners to move data to and from null compartment
  - Only owner can move to and from null compartment
Supporting Memory

17

Plain Text in Processor

Crypto Units

L2 Cache

L1 Instruction Cache

Fetch and Decode

Register File

Data Path

XOM Tags

Private Key

Key Table

Encrypted Text in Memory

XOM Tags
Supporting Memory

- **secure store** instruction

**Registers**

- Data
- Tag

**Caches**

- Data
- Tag

**Memory**

- Data

**Secure Store:**
- Tag is copied from register to cache

**Writeback:**
- Look up Tag, Encrypt and Hash

**XOM Key Table**
Supporting Memory

- **secure load** instruction

**Registers**

- **Data**
- **Tag**

**Caches**

- **Data**
- **Tag**

**Memory**

- **Data**

**Secure Load:**
Load data and tag from cache

**Memory Fetch:**
Look up currently XOM ID, Decrypt and verify Hash

Currently executing XOM ID

Set Tag in cache
Protection Granularity

- Caching reduces the number of cryptographic operations
- The granularity of each message is increased

- The granularity of ownership is increased
  - Need to add per word valid bits
  - Clear all valid bits when tag changes
XOM Hardware Simulator

- The SimOS Simulator:
  - Simulates hardware in enough detail to boot an unmodified operating system
  - Performance modeling processor, caches, memory and disk

- Processor Model: MIPS processor
  - Private Key and Key Table
  - Ciphers and hashes on memory bus
  - Tags in registers and caches
  - Additional instructions
    - Enter/Exit XOM
    - Move to/from NULL
    - Secure Load/Store
Outline

1. Introduction
2. XOM Hardware
3. Operating System Support
   i. OS Issues
   ii. OS Modifications
   iii. Overheads
4. Attack Models
5. Conclusion
Operating System Issue

• Traditional operating systems perform both resource management and protection for applications
  – Since XOM does not trust OS, protection is done in hardware
  – However, OS still has to manage resources
  – Must be able to store interrupted state and restore it later

• Compartments do not allow other programs to read data

• Solution is to encrypt data before allowing the Operating System to handle it
Supporting Interrupts

• *save register* instruction

![Diagram depicting the process of supporting interrupts.]

- Data
- Tag
- To Insecure Main Memory

- Encrypt Data
- Look up program key based on Tag
- Currently executing XOM ID

- XOM Key Table
Supporting Interrupts

- *register restore* instruction

![Diagram]

- Data
- Tag
- Target register tag is set to XOM ID
- Decrypt Data
- Operating System indicates which XOM ID to use
- Currently executing XOM ID
- XOM Key Table

From Insecure Main Memory
Operating System Support

- Modify the IRIX 6.5 Operating System to run on XOM processor
  - IRIX 6.5 is the most current operating system from SGI
  - Deployed on MIPS based SGI computers
  - Boot and run modified IRIX6.5 on our XOM simulator

- Main areas that need modification:
  - Need support for XOM key table
    - Loading/unloading, management
  - Resource management of secure data
    - Traps, Virtual Memory
  - Compatibility with original system
    - Fork, Signal Handling, Dynamic Linking
Operating System Overhead

- Slow down due to operating system overhead due to extra instructions and cache pollution:
  - Costs largely due additional cache misses, a result of larger code and data footprint larger registers
  - Avoid doing these instructions whenever you can, check for XOM compartment

<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>9K</td>
<td>11K</td>
</tr>
<tr>
<td>Signal Handling</td>
<td>65K</td>
<td>99K</td>
</tr>
<tr>
<td>Fork</td>
<td>702K</td>
<td>784K</td>
</tr>
</tbody>
</table>
Application Overhead

<table>
<thead>
<tr>
<th></th>
<th>Execution Overhead</th>
<th>Instruction Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPG Decode</td>
<td>3.0%</td>
<td>0.0004%</td>
</tr>
<tr>
<td>RSA</td>
<td>3.0%</td>
<td>0.0004%</td>
</tr>
</tbody>
</table>

- Application overhead due to:
  - Memory latency due to cryptography
  - Cache behavior

- Results are depend on application:
  - These applications did not miss heavily in the cache
  - Additional memory latency adds small overhead
Outline

1. Introduction
2. XOM Hardware
3. Operating System Support
4. Attack Models
   I. Formal Verification
   II. Spoofing Attacks
   III. Splicing Attacks
   IV. Replay Attacks
5. Conclusion
Model Checking

• Model Checker exhaustively explores the state space of the model, checking that every state satisfies invariants

• The state space must be kept small
  – Model is an abstraction of the real system
  – Model checkers cannot prove correctness, but are very useful in finding errors

• Use model checker to verify that given a malicious operating system, program code and data is safe from tampering and observation
Invariant 1

1. **Program data cannot be read by adversary**
   - XOM machine performs tag check on every access
   - Make sure that owner of data always matches the tag
Invariant 2

2. Adversary cannot modify the program without detection
   • Need a “pristine” model to compare XOM model against
   • Define a second, simpler model that adversary cannot affect
   • Make sure the state of the model is consistent with the state of the “pristine” model
Spoofing Attacks

- Tags are able to catch spoofed attacks because tag ID changes
- Encryption alone is not sufficient for memory
- Spoofing attack:
  - Adversary tries to substitute fake cipher text to alter behavior

Data → Encrypt and store to memory → False Data → Junk

Adversary swaps data → Decrypts to Junk but alters program behavior
Spoofing Prevention

- Solution is to add an integrity hash to the encryption
  - Adversary has to reverse encryption to fake the hash

For this reason, encrypted data is larger than unencrypted data
Splicing Attacks

- **Splicing attack:**
  - Adversary copies valid cipher text from another location replacing one value with another

Address 1

Data 1 → Data 1

Encryption and storage into memory

Decryption from memory, Data 2 has been replaced with Data 1

Address 2

Data 2 → Data 2

Data 1 → Data 1

- **Position dependent hash prevents this**
  - For secure load/store, values must be from the same virtual address
  - For secure restore/save values must be from the same register number
Register Replay Attacks

- **Replay Attack**
  - Adversary records previous valid values and reuses them
  - The OS records register values and replays them

- Key Table uses a register key (different from the compartment key)
- Old register key is revoked every time OS interrupts process
- Similarly, we can protect memory from replay attacks by keeping a hash in a register
What XOM Cannot Prevent

• XOM does not prevent denial of service
  – The Operating System controls resource management
  – So it can always prevent applications from running by denying resources

• XOM does not prevent frequency analysis of data
  – Attacker can observe cipher texts in memory

• XOM does not prevent adversary from getting an address trace
  – OS can use the TLB to get a trace
  – Application can stop this (Ostrovsky, Oblivious RAM)

• Other attacks are shown to be prevented with formal verification (model checker)
Verification Result

• Show that a malicious operating system can’t tamper with software

• Show that all actions in the XOM processor are required:
  – Removing any actions allows the adversary to break the model

• Show that a properly working operating system can guarantee forward progress:
  – By restraining the OS, show that XOM exceptions are never triggered
Conclusions

- The XOM model is a working system that separates protection from resource management
  - Data protection and access control is in hardware
  - Resource management is in Operating System
  - Complete working system
  - OS semantics are preserved

- Implementation requires
  - Required hardware is a secret private key and storage for key table
  - Modifications to the operating system, mainly in areas that deal with program data

- Performance impact can be made pretty small with hardware assist
Future Work

• More detailed analysis of applications:
  – XOM doesn’t stop programmers from writing security bugs into their programs
  – How do applications mitigate XOM OS and Hardware overhead

• Implementation alternatives
  – Virtual machine authenticated with secure boot
  – Use a secure coprocessor

• Alternative applications
  – Intrusion detection and monitoring
  – Strong forms of isolation for services