Specifying and Verifying Hardware Support for Copy and Tamper-Resistant Software

David Lie, John Mitchell, Chandramohan Thekkath and Mark Horowitz

Computer Systems Lab
Stanford University
How Secure is Something?

• After building a system, how do you know it’s secure?
  – Try to reason about security
  – Think about existing attacks
  – Think, think, think…

• Computers never get tired of thinking:
  – Model checkers exhaustively check the state space of a state machine
  – Prove correctness for explored states

• Model checkers combine the advantages of formal methods with automated brute-force abilities of computers
Problems with Model Checking

• Computers aren’t very smart:
  – Need to model the system as a state-machine
  – A set of logical statements define:
    • State vector for the machine
    • Next State functions
    • Correctness properties

• Models must be abstracted:
  – Model Checkers can only check a finite state space
  – State space of models must be reduced
Checking Systems

• Previous work on automatically checking security.

• Previous work on formally verifying security:
Verifying Secure Processors

• Present a methodology for verifying security processors:
  – Show that an adversary executing on the system as the operating system cannot attack other users on the system

• Reduce complexity of the system:
  – Less logic means a simpler system
  – Remove actions that are not necessary for security

• Show liveness in the system:
  – Despite the restrictions imposed by security, the system is still usable and can guarantee forward progress
XOM

• **Our solution: eXecute Only Memory or “XOM”**
  – Programs in this memory can only be executed, they cannot be read or modified
  – Provides *isolation* between programs so that even the operating system cannot attack a user process

• 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 Provides Isolation

Secure XOM Machine

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

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

Ownership Tags

XOM Machine Isolation
Cryptography

• Cryptography is used to protect data and code when it is off-chip in memory or on disk:
  – Values in memory are encrypted and hashed with a MAC

• Operating system must be able to virtualize resources
  – Memory is encrypted so OS simply copies ciphertext and MAC’s
  – During an interrupt, OS asks hardware to encrypt registers first so that it can save the state
  – To restore, the OS presents the encrypted registers and the hardware restores them
Model Checking XOM

- XOM Model is a state machine:
  - State Vector
    - A set of all things on the chip that can hold state
    - Based on the Processor Hardware
  - Next-State Functions
    - A set of state transitions that the hardware can have
    - Derived from the instructions that can be executed on the processor
  - Invariants
    - Define the correct operation of the XOM processor
    - Two Goals: Prevent observation and modification
XOM Processor Hardware

- **Memory**: Data, Hash
- **Crypto Units**: Plain Text in Processor
- **Cache**: XOM Tags
- **Register File**: XOM Tags

Encrypted Text in Memory
Modeling XOM

- Defining the state space:
  - Hardware units are modeled as arrays of elements
  - Number of elements is scaled down

- 3 Registers:
  
  \[ r_i = \{ \text{Data}, \text{Tag}, \text{Key}, \text{Hash} \} \]

- 3 Cache Lines:
  
  \[ c_j = \{ \text{Data}, \text{Addr}, \text{Tag} \} \]

- 3 Memory Words:
  
  \[ m_k = \{ \text{Data}, \text{Hash}, \text{Key} \} \]
XOM User Instructions

- Instruction Available to the user:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Use</td>
<td>Reading a register</td>
</tr>
<tr>
<td>Register Define</td>
<td>Writing a register</td>
</tr>
<tr>
<td>Store</td>
<td>Store data to memory</td>
</tr>
<tr>
<td>Load</td>
<td>Load data from memory</td>
</tr>
</tbody>
</table>
XOM Kernel Instructions

- We assume an adversarial operating system
  - Operating system can execute user instructions and privileged kernel instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Save</td>
<td>Encrypt a user register for saving</td>
</tr>
<tr>
<td>Register Restore</td>
<td>Decrypt a user register for restore</td>
</tr>
<tr>
<td>Prefetch Cache</td>
<td>Move data from memory into the cache</td>
</tr>
<tr>
<td>Write Cache</td>
<td>Overwrite data in the caches</td>
</tr>
<tr>
<td>Flush Cache</td>
<td>Flush a cache line into memory</td>
</tr>
<tr>
<td>Trap</td>
<td>Interrupt User</td>
</tr>
<tr>
<td>Return from Trap</td>
<td>Return execution to User</td>
</tr>
</tbody>
</table>
State Transitions

- State Transitions derived from instruction set
  - User has access to user level instructions
  - Adversary has access to kernel level instructions

- Example: Store $r_i \rightarrow m_j$

  \[
  \begin{align*}
  &\text{if } r_i.\text{tag} = \text{user} \text{ then} \\
  &\quad \text{reset} \\
  &\text{else} \\
  &\quad \text{if } j \text{ is in cache then} \\
  &\quad\quad c_k = \{\text{data} = r_i.\text{data}, \text{addr} = j, \text{tag} = r_i.\text{tag}\} \\
  &\quad\text{else pick a free } c \\
  &\quad\quad c_{\text{free}} = \{\text{data} = r_i.\text{data}, \text{addr} = j, \text{tag} = r_i.\text{tag}\}
  \end{align*}
  \]
No Observation Invariant

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

   \[
   \text{if } r_i\text{.data is user data then} \\
   \quad \text{check that: } r_i\text{.tag = user} \\
   \text{else} \\
   \quad \text{check that: } r_i\text{.tag = adversary}
   \]
2. Adversary cannot modify the program without detection
   • Adversary may modify state by copying or moving user data
   • Need a “ideal” correct model to check against

For Memory:
\[
\text{if } \text{xom.m}_i\text{.data} = \text{user data} \text{ then check that: } \text{ideal.m}_i\text{.data} = \text{xom.m}_i\text{.data}
\]
Checking for Correctness

- Model checker helped us find bugs and correct them
  - 2 old errors were found
  - 2 new errors were found and corrected

- Example:
  - Case where it’s possible to replay a memory location
  - This was due to the write to memory and hash of the memory location not being atomic
Memory Replay

- **Optimization 1**: Only update hash on cache write-back
  - On-chip cache is protected by tags

<table>
<thead>
<tr>
<th>Principal</th>
<th>Action</th>
<th>$</th>
<th>M</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Writes A into Cache</td>
<td>A</td>
<td>--</td>
<td>$H = \emptyset$</td>
</tr>
<tr>
<td>Machine</td>
<td>Flushes Cache</td>
<td>--</td>
<td>A</td>
<td>$H = h(A)$</td>
</tr>
<tr>
<td>Program</td>
<td>Writes B into Cache</td>
<td>B</td>
<td>A</td>
<td>$H = h(A)$</td>
</tr>
<tr>
<td>Adversary</td>
<td>Invalidates Cache</td>
<td>--</td>
<td>A</td>
<td>$H = h(A)$</td>
</tr>
<tr>
<td>Program</td>
<td>Reads Memory</td>
<td>--</td>
<td>A</td>
<td>$H = h(A)$</td>
</tr>
</tbody>
</table>

- **Fix by making write and hash calculation atomic!**
Reducing Complexity

- Fewer operations makes logic simpler
- Exhaustively remove actions from the next-state functions
  - If a removed action does not result in a violation of an invariant then the action is extraneous
  - Example:

Caches

Data Tag

Secure Load: Tag and Data is copied from cache

Tag Check: Make sure tag matches the user

Registers

Data Tag

Check Happens Anyways!

Register Use: Check that tag matches user
Liveness

- A weak form of forward progress guarantee:
  - At all times operating system or user can always execute an instruction
  - All instructions can be executed somewhere in the state space

- Constrain the operating system so that:
  1. Operating system always restores user state
  2. Operating system does not overwrite user data

- Check that within the state-space:
  1. User is never halted due to access violation
  2. User and operating system are able to execute every instruction
Conclusions

• Model Checkers are an effective tool for verifying security of processors
  – Hardware blocks define state vector
  – Instructions define next-state functions

• Can be used to verify:
  – **Tamper-resistance** by checking consistency between an “ideal” model and “actual” model
  – **Minimal Complexity** by checking that every action is necessary for correctness
  – **Liveness** by making the adversary cooperative and showing that both are always able to execute actions