Architectural Support for Copy and Tamper Resistant Software

David Lie, Chandu Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell and Mark Horowitz
Computer Systems Laboratory
Stanford University
XOM

• XOM: eXecute Only Memory
• Would like to support an environment where programs are protected from copying and tampering

- Prevent Hackers
- Protect Intellectual Property
- Combat Software Piracy
General Strategy

• Interfaces are suspect
  – Disk and memory are considered insecure

• On chip storage can be trusted

• How compartments are implemented
  – Data is always protected by some mechanism
  – With hardware tags when on chip
  – With crypto when off chip
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
  – Are much slower than symmetric ciphers

• Symmetric Ciphers (3DES, Blowfish)
  – Single key used for encryption and decryption
  – Pretty fast when implemented in hardware
  – Advanced Encryption Standard ciphers will be even faster
Software Distribution Method

Customer wishes to purchase software

Customer sends public key

Symmetric key is encrypted with public key

Customer receives encrypted code with encrypted key

Encrypt Program

Randomly Selected Symmetric Key

Program Code

Encrypted Code

Encrypted Code

 Encrypt Program
Loading Secure Code

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

Description:
- Encrypted Code is sent to the Secure XOM Machine.
- The Asymmetric Decryption Module decrypts the Encrypted Symmetric Key.
- The Symmetric Decryption Module then decrypts the Encrypted Code to produce Executable Code.
- The Private Key is used for decryption.
- The XOM Key Table is used for secure execution.
Managing Data

• Simple hardware rule: a tag check on every access

• But the model is too rigid, applications cannot pass out data for other applications to read

• Have a special compartment with a tag ID of zero called the “null” compartment

• Special instructions are required to move data to and from null to a program’s private compartment
  – Allows for communication with principals outside of compartment
A Simple XOM Machine

Main Memory

L2 Cache

L1 Cache

Instruction

Register

File

Datapath

XOM Tags

XOM

Protection Boundary
Two problems

• Memory is insecure
  – All sensitive program data must fit in registers
  – Too restrictive a programming model

• Programs can’t read or write data that doesn’t belong to them
  – But OS needs to do this when performing a context switch

• Use encryption to solve both problems
  – This was the same technique used to protect sensitive code
Supporting Main Memory

- `store_secure` instruction

Data | Tag | Data

To Insecure Main Memory

Encrypt Data before storing in memory

Check that the currently executing XOM ID matches the tag

Currently executing XOM ID

XOM Key Table
Supporting Main Memory

- `load_secure` instruction

```
Data  Tag

Data

From Insecure Main Memory
```

- Decrypt Data
- Currently executing XOM ID
- Target register tag is set to XOM ID
- XOM Key Table
Supporting Interrupts

• *save_secure* instruction

![Diagram showing data flow and key table]

- Data
- Tag
- Encrypt Data
- To Insecure Main Memory
- Look up session key based on Tag
- XOM Key Table
- Currently executing XOM ID
Supporting Interrupts

- `restore_secure` instruction

![Diagram]

- Data Tag
- Data
- From Insecure Main Memory
- Decrypt Data
- Executing Program indicates which XOM ID to use
- Currently executing XOM ID
- XOM Key Table

Target register tag is set to XOM ID
Spoofing Attacks

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

Data → Data → False Data → Junk

Encrypt and store to memory
Adversary swaps data
Decrypts to Junk but alters program behavior
Spoofing Prevention

- Solution is to add an integrity hash to the encryption
  - In cryptography terms, this is called a Message Authentication Code (MAC)

Data → Data → False Data

Encrypt and store to memory with added hash

Adversary swaps data

Hash does not match data so exception is thrown
Splicing Attacks and Replay Attacks

• Splicing Attacks
  – Attacker moves valid data from one location to another location
  – Add position dependent hash:
    • Virtual Address for secure load/stores
    • Register number for secure save/restores

• Replay Attacks
  – Attack records and reuses old register and memory values
  – Add a regenerative key to Key Table that is used for save/restores
  – Use protected registers to protect memory values
Required Hardware

- Private Memory
- L2 Cache
- XOM Tags
- L1 Instruction Cache
- Register File
- Datapath
- Protection Boundary
- Private Key
- L1 Data Cache
- Micro-code or Virtual Machine for control

Diagram:
- Main Memory
- L2 Cache
- L1 Cache miss trap
- Decode
- Path to write decrypted instructions into L1 Instruction Cache
- XOM Tags
- Protection Boundary
Performance Issues

• Performance hit is going to come from the cryptographic operations
  – XOM start-up
  – Instruction load path from memory
  – Data loads and stores to and from memory
  – Register saves and restores to and from memory

• The accesses to memory occur the most often

• We want to speed up the symmetric and hashing operations, as well as optimize access to memory
Additional Hardware

• Cache decrypted data
  – Add tags to caches

• Speed up symmetric operations
  – Add special symmetric cryptography hardware

• Speed up hash calculation
  – Select a fast hash calculation such as 128 bit CRC
Caching Values

- Caching reduces the number of cryptographic operations
- The size 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
Full XOM Machine

- Main Memory
  - Symmetric Accelerators
  - L2 Cache
  - L1 Instruction Cache
  - L1 Data Cache
  - Decode
    - Register File
  - Datapath
  - Private Memory
    - Private Key
  - Protection Boundary

- XOM_tags
  - XOM Tags
  - XOM Tags
  - XOM Tags
Summary

• Show how to implement compartments with architectural support
• Trust only the processor and assume memory and OS are insecure
• Use:
  – Data tagging on-chip
  – Crypto off-chip
• Required hardware is modest
  – Private Memory and Key
  – XOM Tags on registers
• Additional hardware can be added to improve performance
  – Symmetric hardware
  – XOM Tags in caches