

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.
 - D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. **A first step towards automated detection of buffer overrun vulnerabilities.**
 - K. Ashcraft and D. Engler. **Using programmer-written compiler extensions to catch security holes.**
- Previous work on formally verifying security:
 - J. Mitchell, M. Mitchell, and U. Stern. **Automated analysis of cryptographic protocols using Murphi**
 - S. Smith, R. Perez, S. Weingart, and V. Austel. **Validating a high-performance, programmable secure coprocessor.**



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

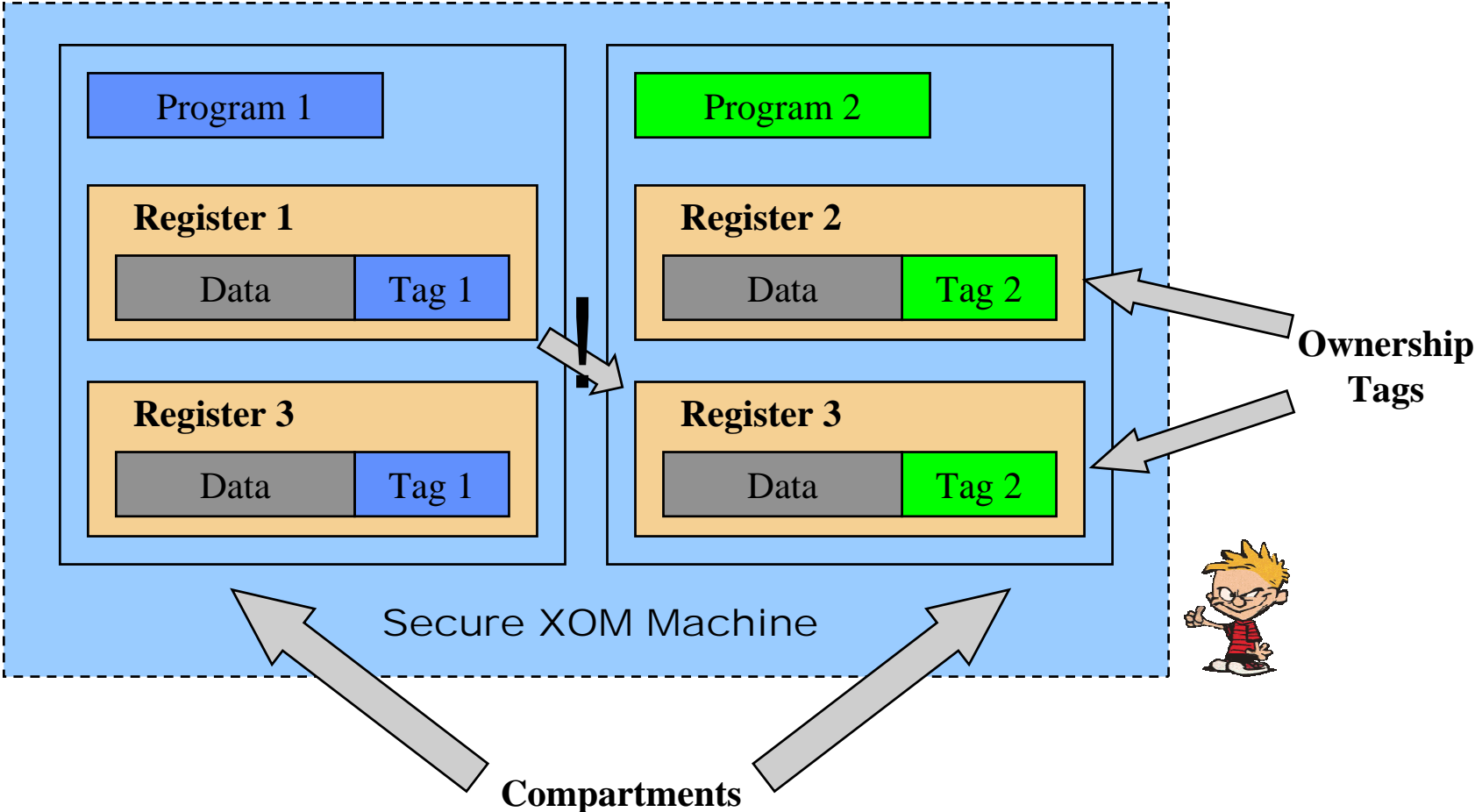


XOM

-
- **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



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

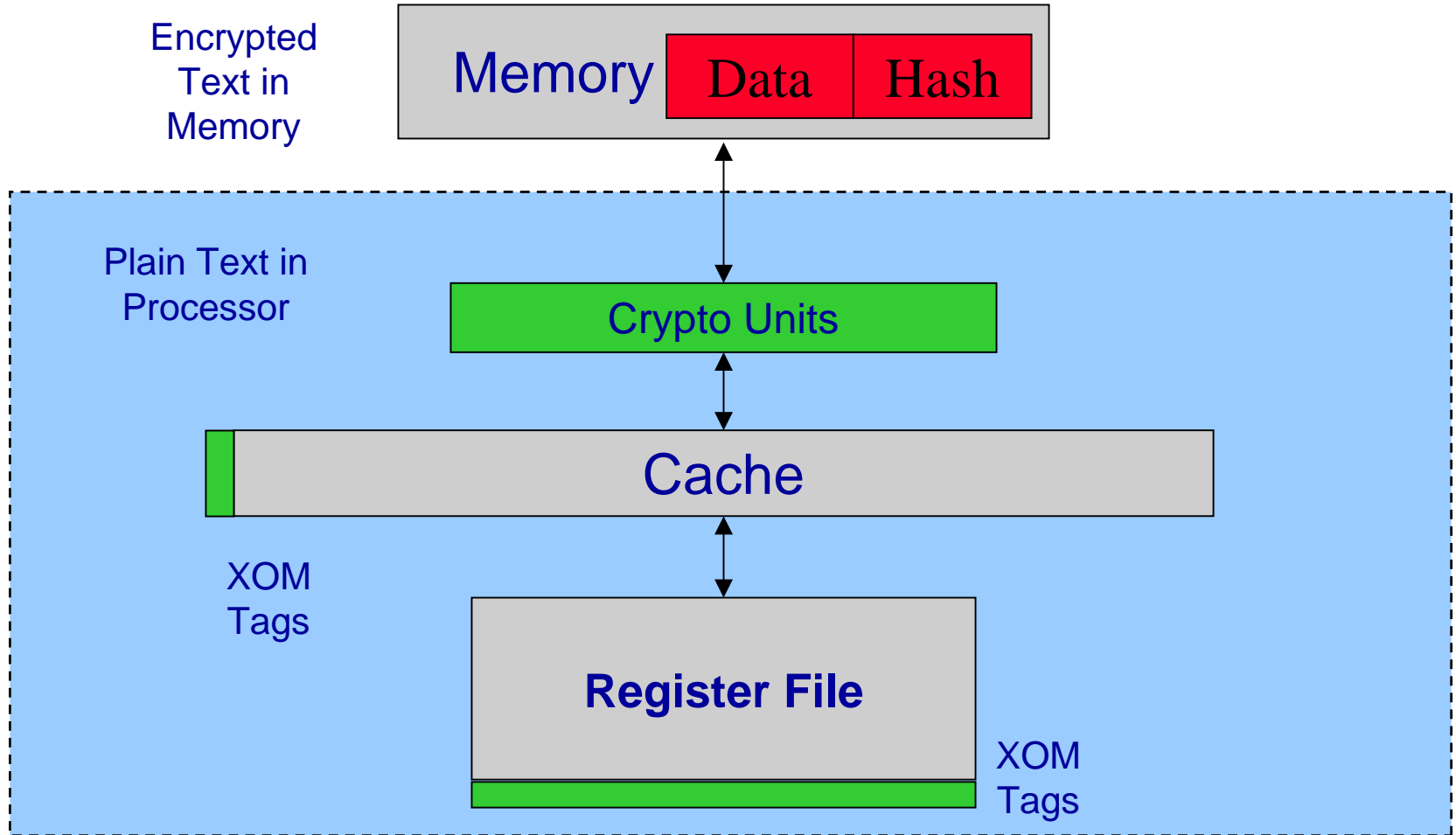


Model Checking XOM

- 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



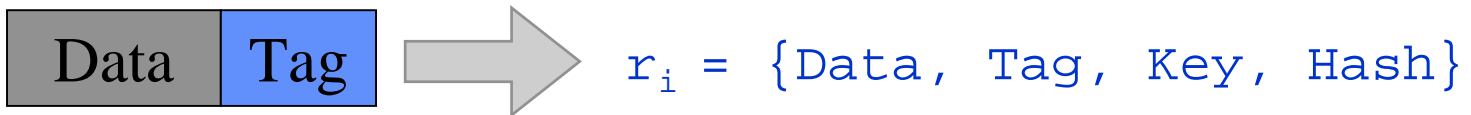
XOM Processor Hardware



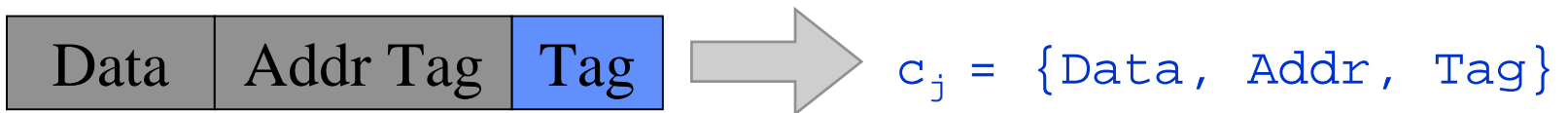
Modeling XOM

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

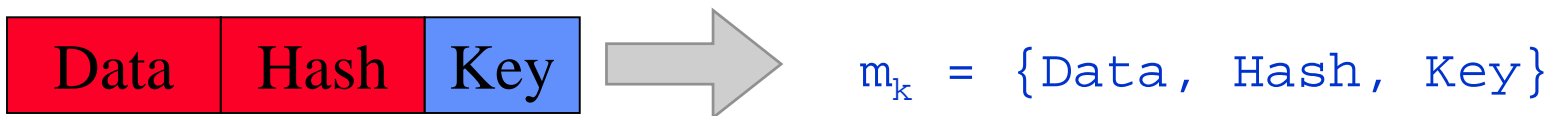
- 3 Registers:



- 3 Cache Lines:



- 3 Memory Words:



XOM User Instructions

- Instruction Available to the user:

Instruction	Description
Register Use	Reading a register
Register Define	Writing a register
Store	Store data to memory
Load	Load data from memory



XOM Kernel Instructions

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

Instruction	Description
Register Save	Encrypt a user register for saving
Register Restore	Decrypt a user register for restore
Prefetch Cache	Move data from memory into the cache
Write Cache	Overwrite data in the caches
Flush Cache	Flush a cache line into memory
Trap	Interrupt User
Return from Trap	Return execution to User



State Transitions

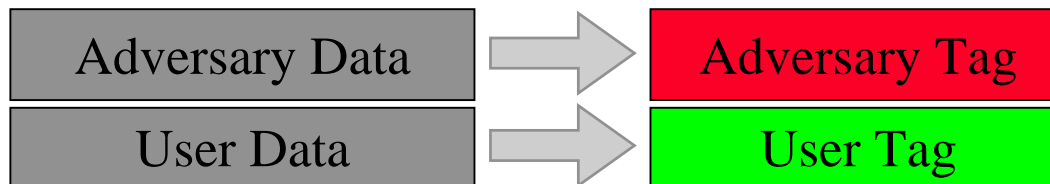
- State Transitions derived from instruction set
 - User has access to user level instructions
 - Adversary has access to kernel level instructions
- Example: Store $r_i \rightarrow m_j$
 - if $r_i.tag = user$ then
 - reset
 - else
 - if j is in cache then
 - $c_k = \{data = r_i.data, addr = j, tag = r_i.tag\}$
 - else pick a free c
 - $c_{free} = \{data = r_i.data, addr = j, tag = r_i.tag\}$



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



if $r_i.data$ is user data then

check that: $r_i.tag = user$

else

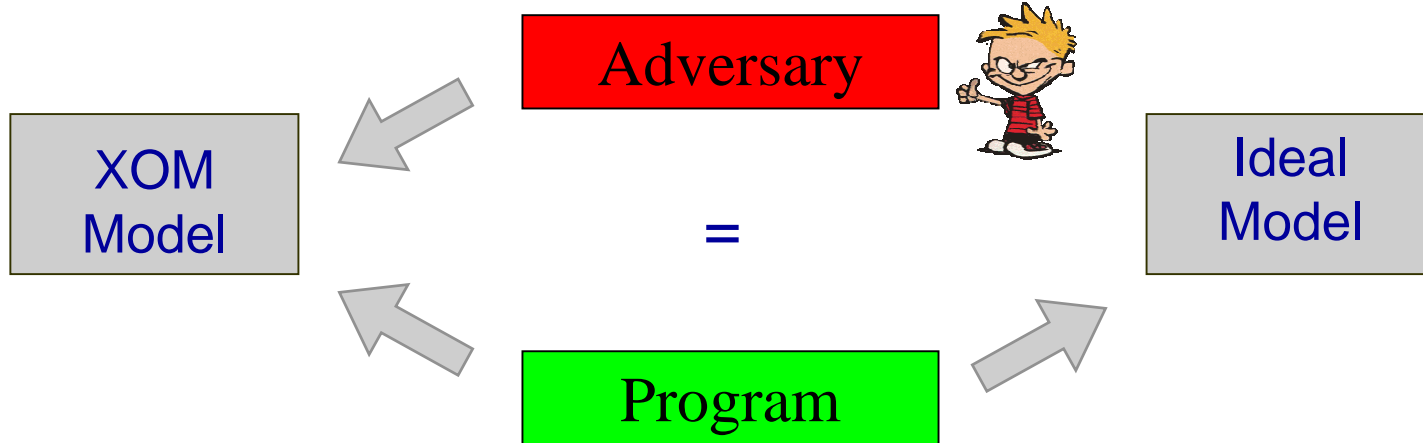
check that: $r_i.tag = adversary$



No Modification Invariant

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:

if `xom.mi.data = user data` then

check that: `ideal.mi.data = xom.mi.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

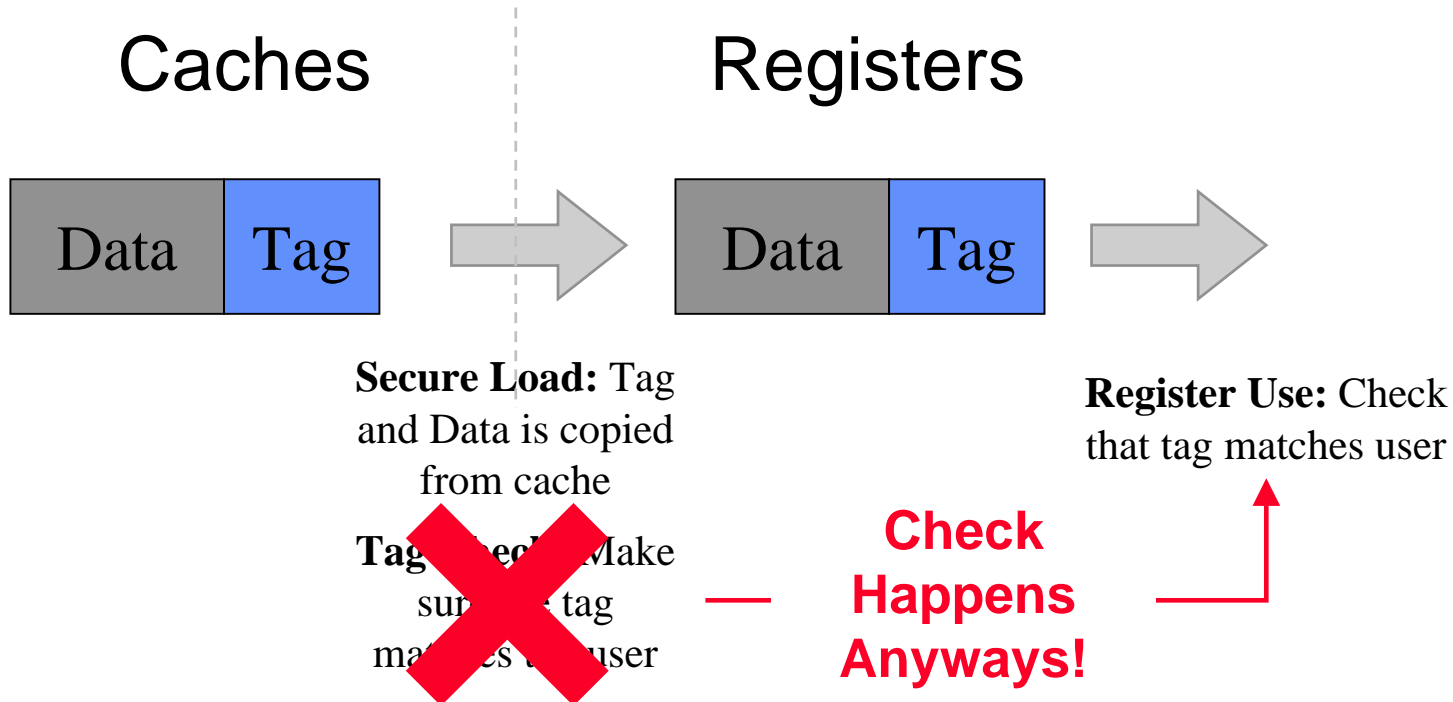
Principal	Action	\$	M	Hash
Program	Writes A into Cache	A	--	$H = \emptyset$
Machine	Flushes Cache	--	A	$H = h(A)$
Program	Writes B into Cache	B	A	$H = h(A)$
 Adversary	Invalidates Cache	--	A	$H = h(A)$
Program	Reads Memory	--	A	$H = h(A)$

- **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:



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

