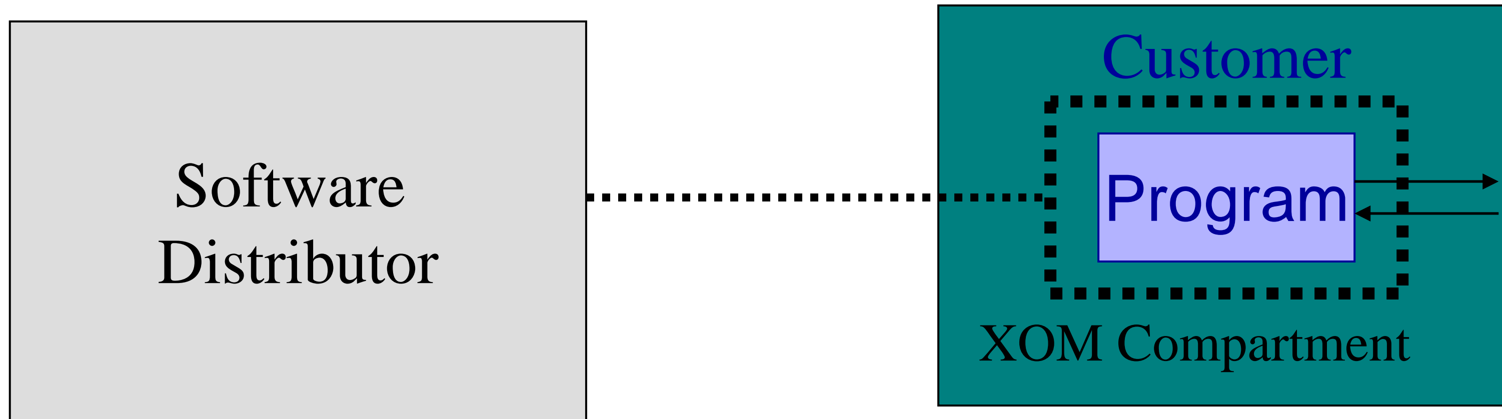
Architectural Support for Copy and Tamper Resistant Software

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

Reference: http://www-vlsi.stanford.edu/~lie/Papers/xom-oakland-2002.pdf

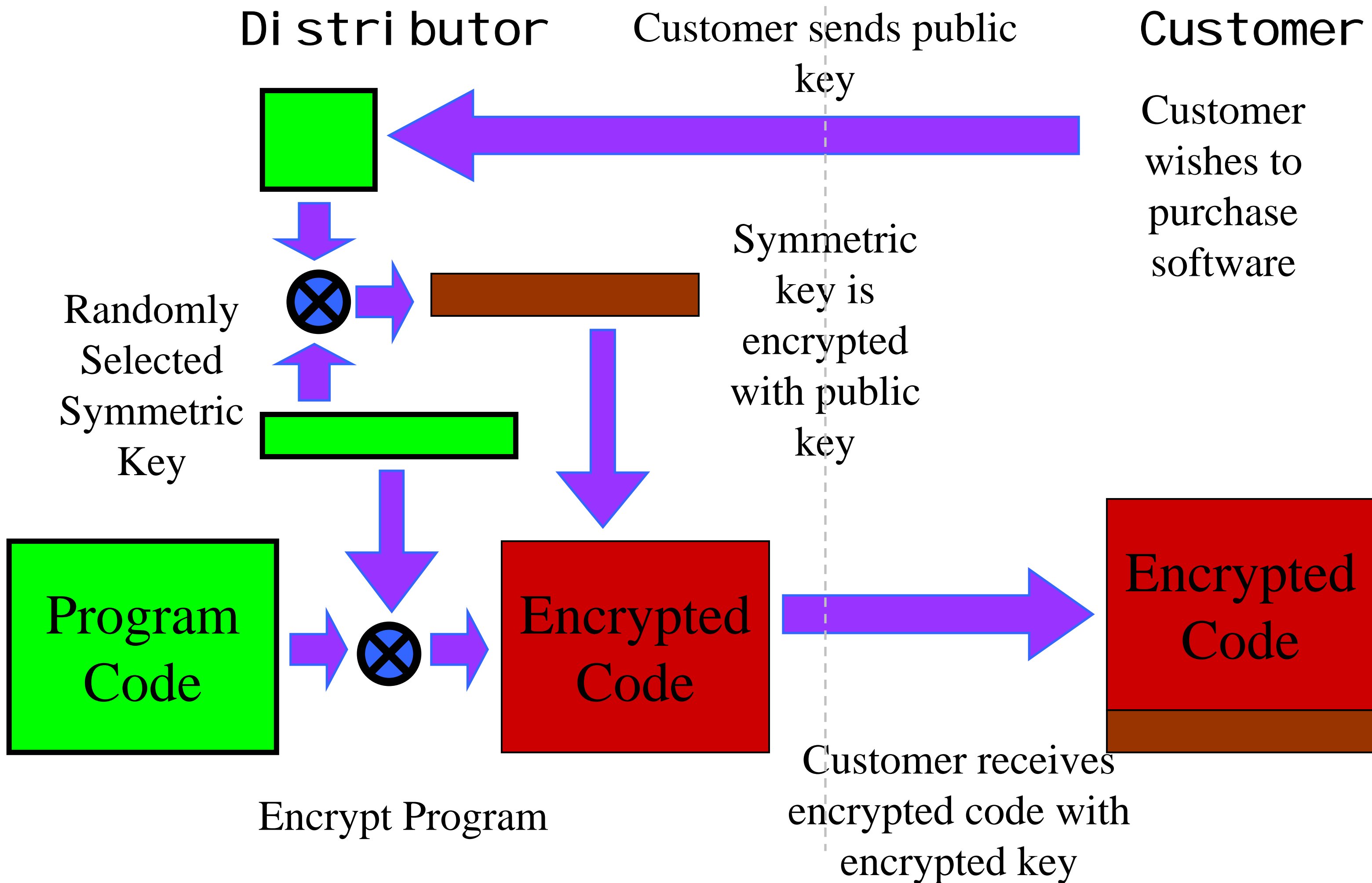
XOM

- XOM: eXecute Only Memory
- Protect programs from copying and tampering

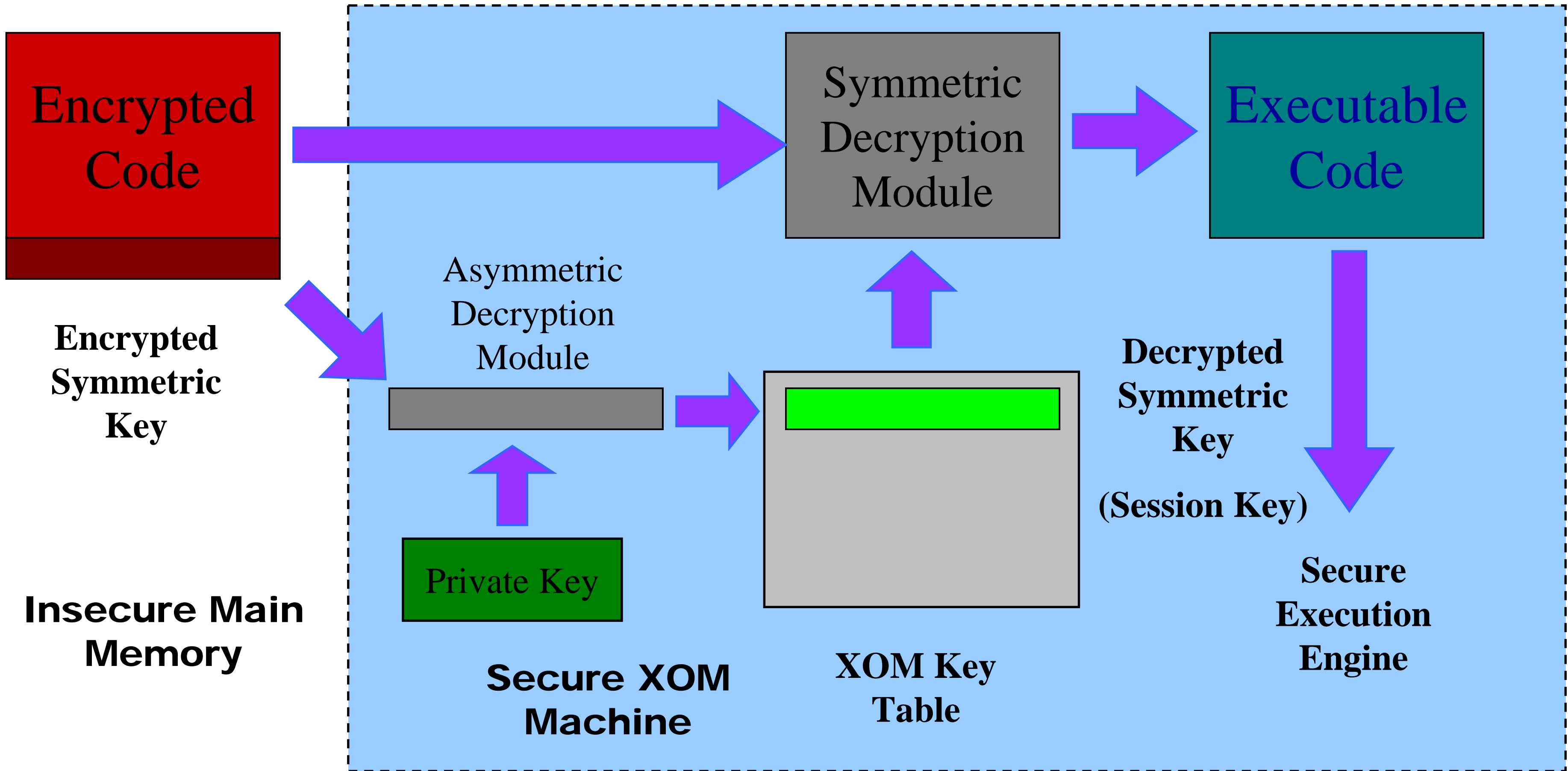


- Prevent software theft
  - Mark software for specific machine
- Guarantee pay-per-use
  - Program contains usage meter
  - Program sends payment to distributor
  - Cannot proceed until receipt is received
- Fraud detection in embedded apps
  - Cell phone id, etc.

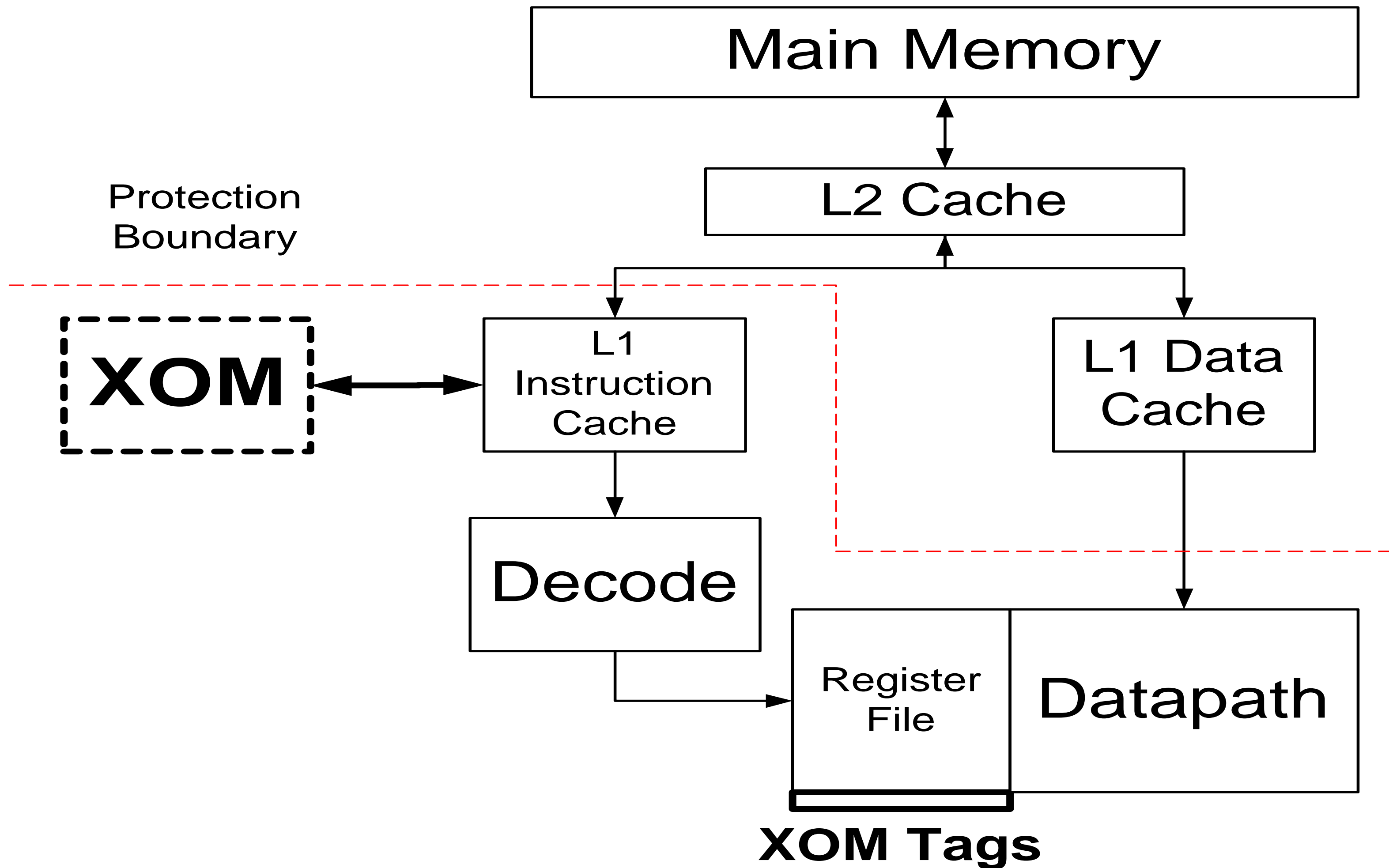
Software Distribution Method



Loading Secure Code



A Simple XOM Machine

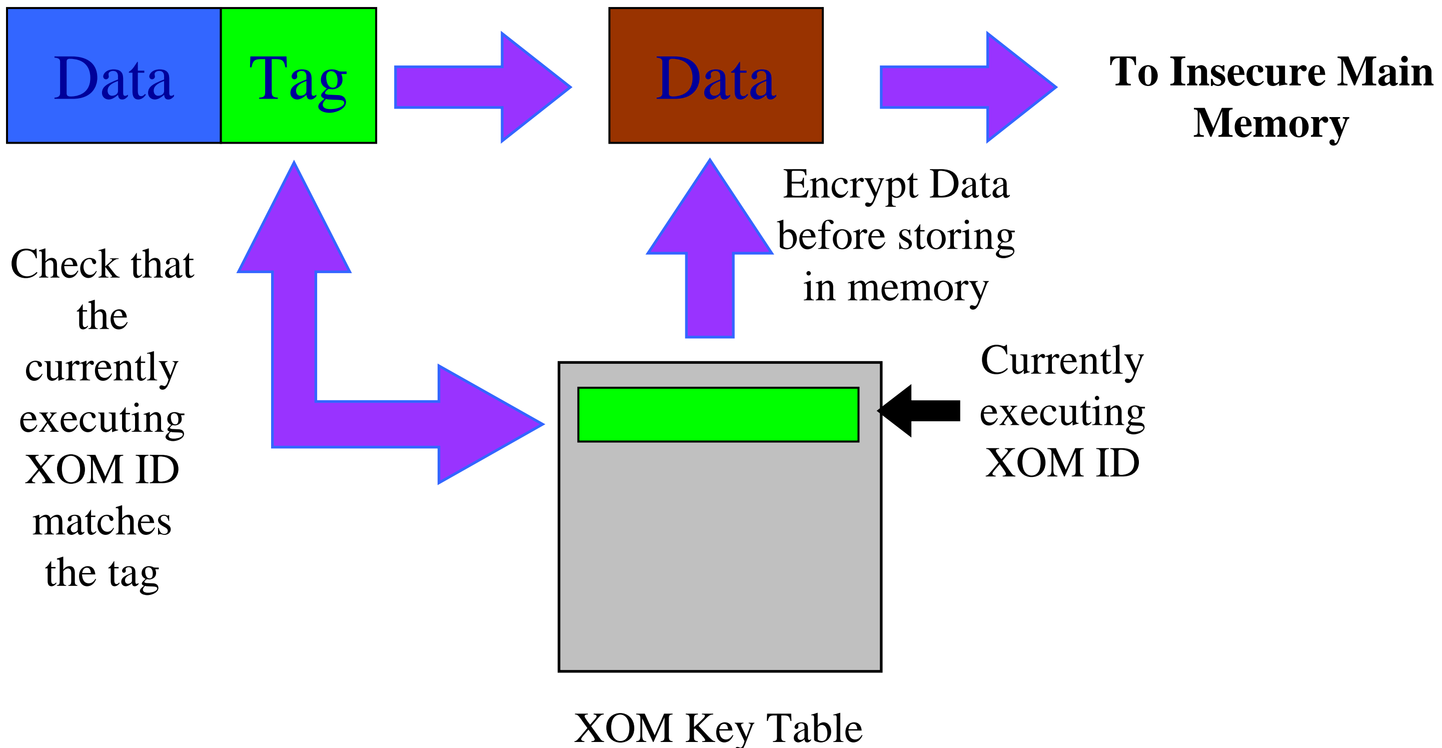


Two problems

- 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 doing a context switch
- Use encryption to solve both problems
  - Same technique used to protect sensitive code

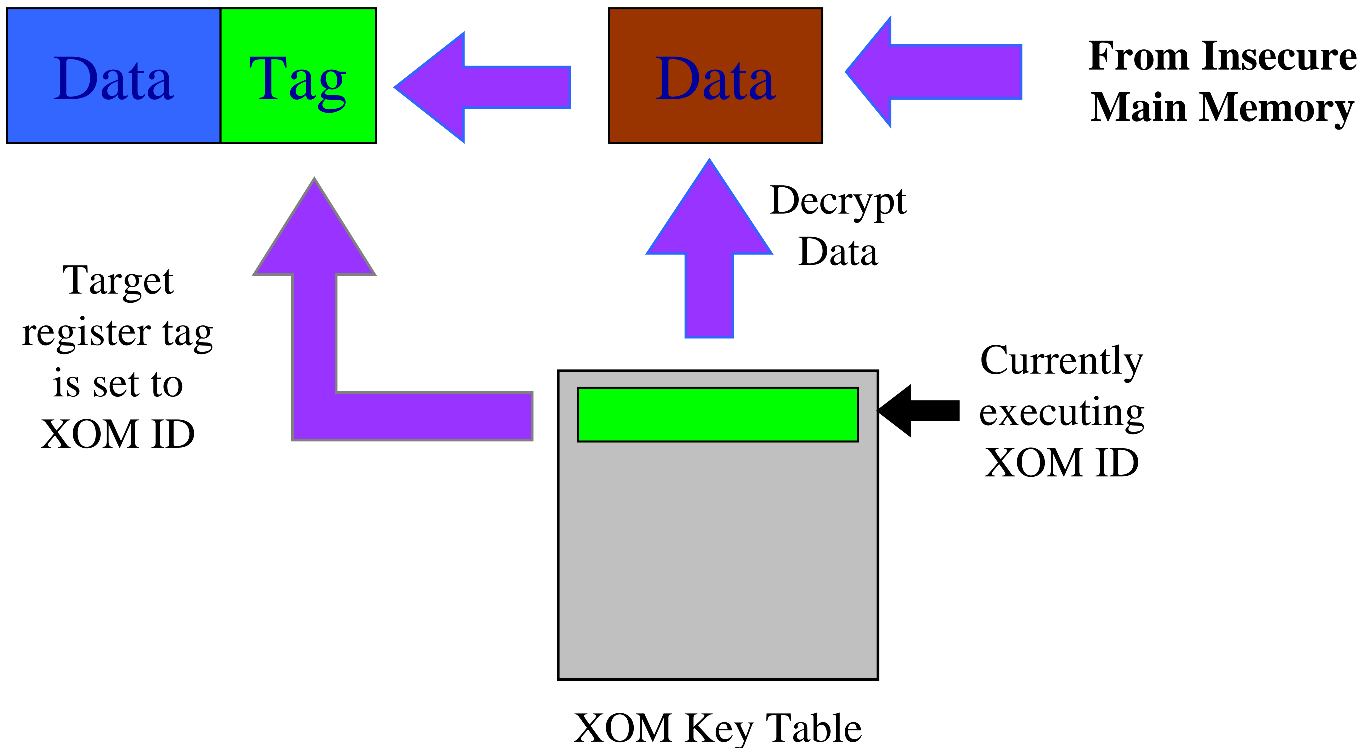
Supporting Main Memory

- *store_secure instruction*



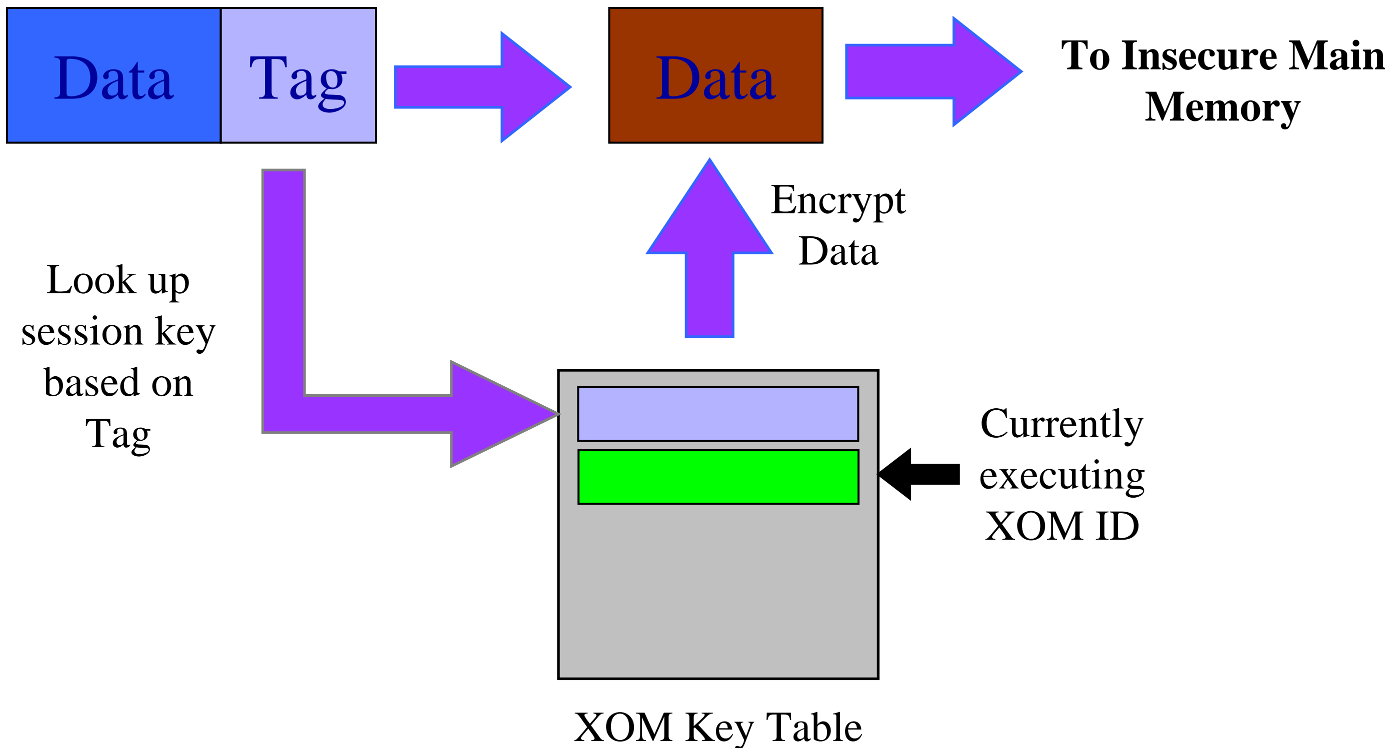
Supporting Main Memory

- *load_secure* instruction



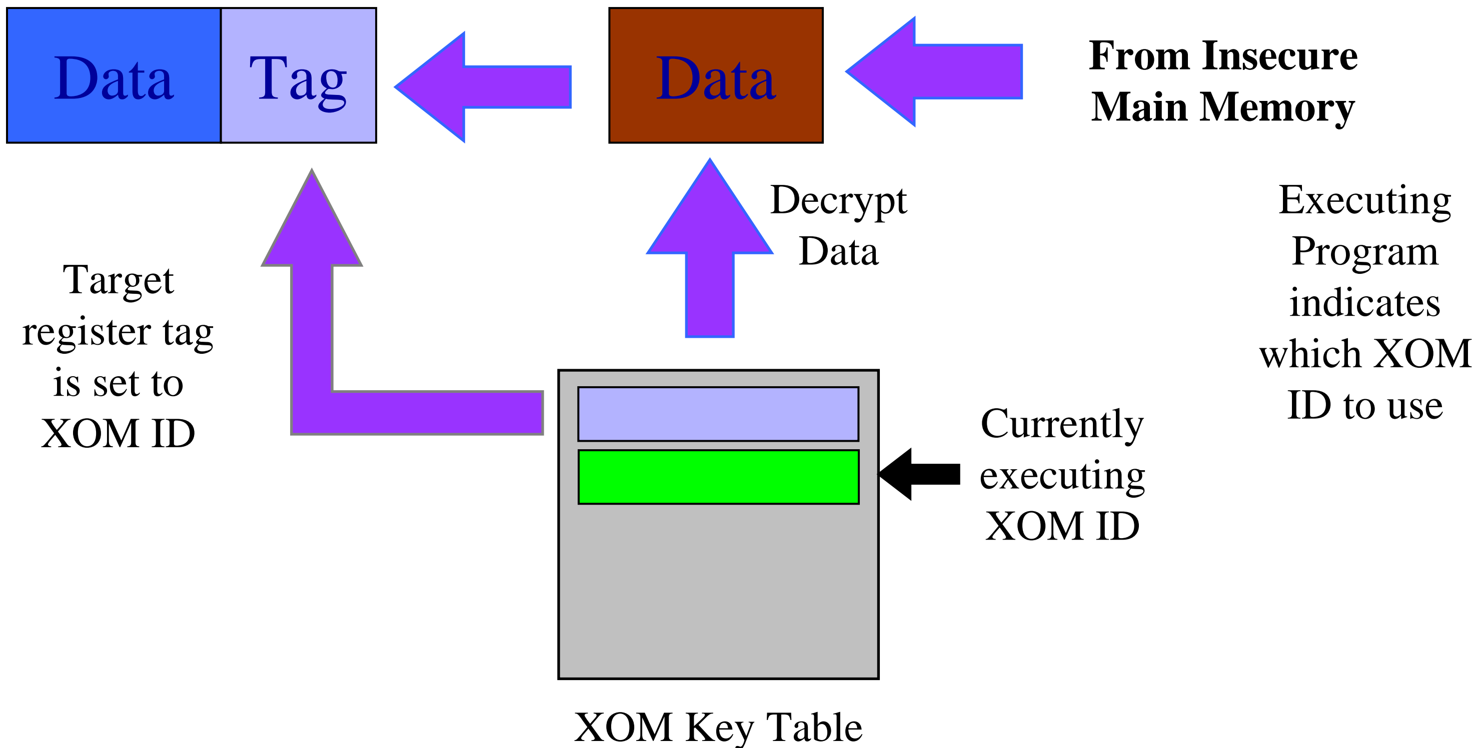
Supporting Interrupts

- *save_secure instruction*



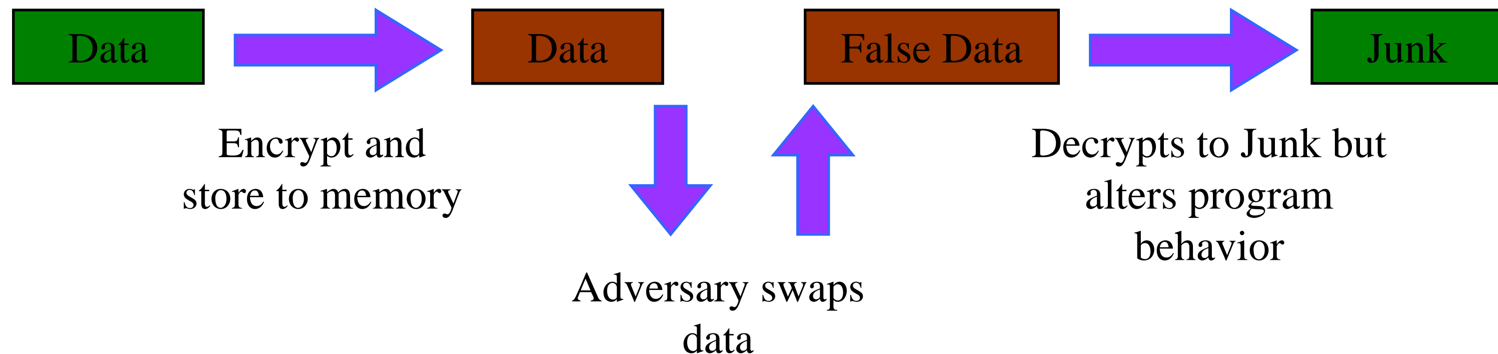
Supporting Interrupts

- *restore_secure instruction*



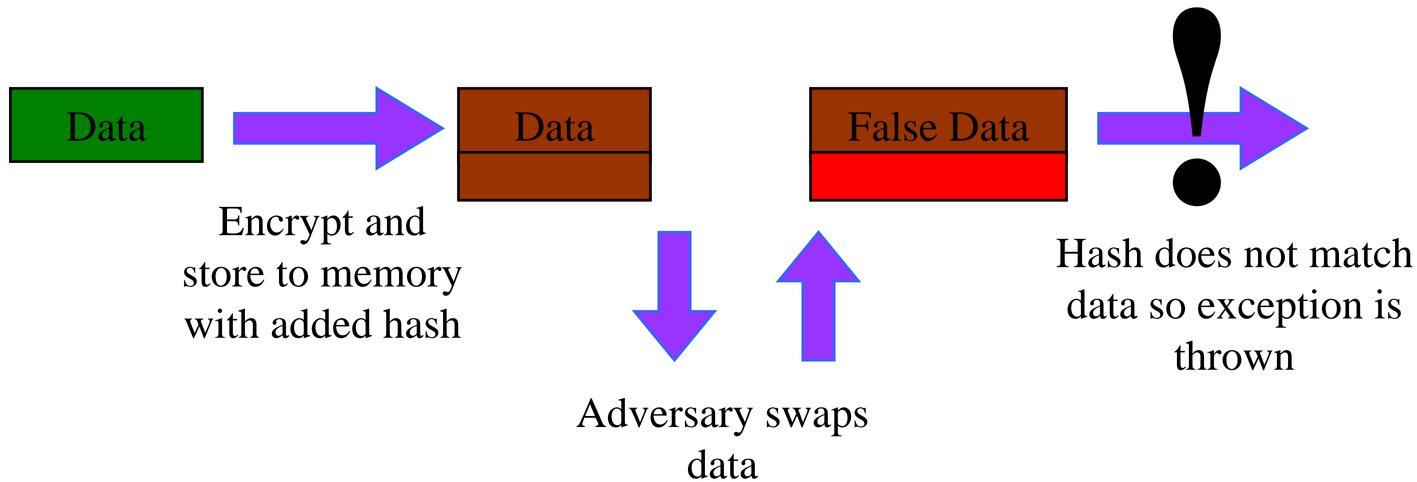
Spoofing Attacks

- 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



Spoofing Prevention

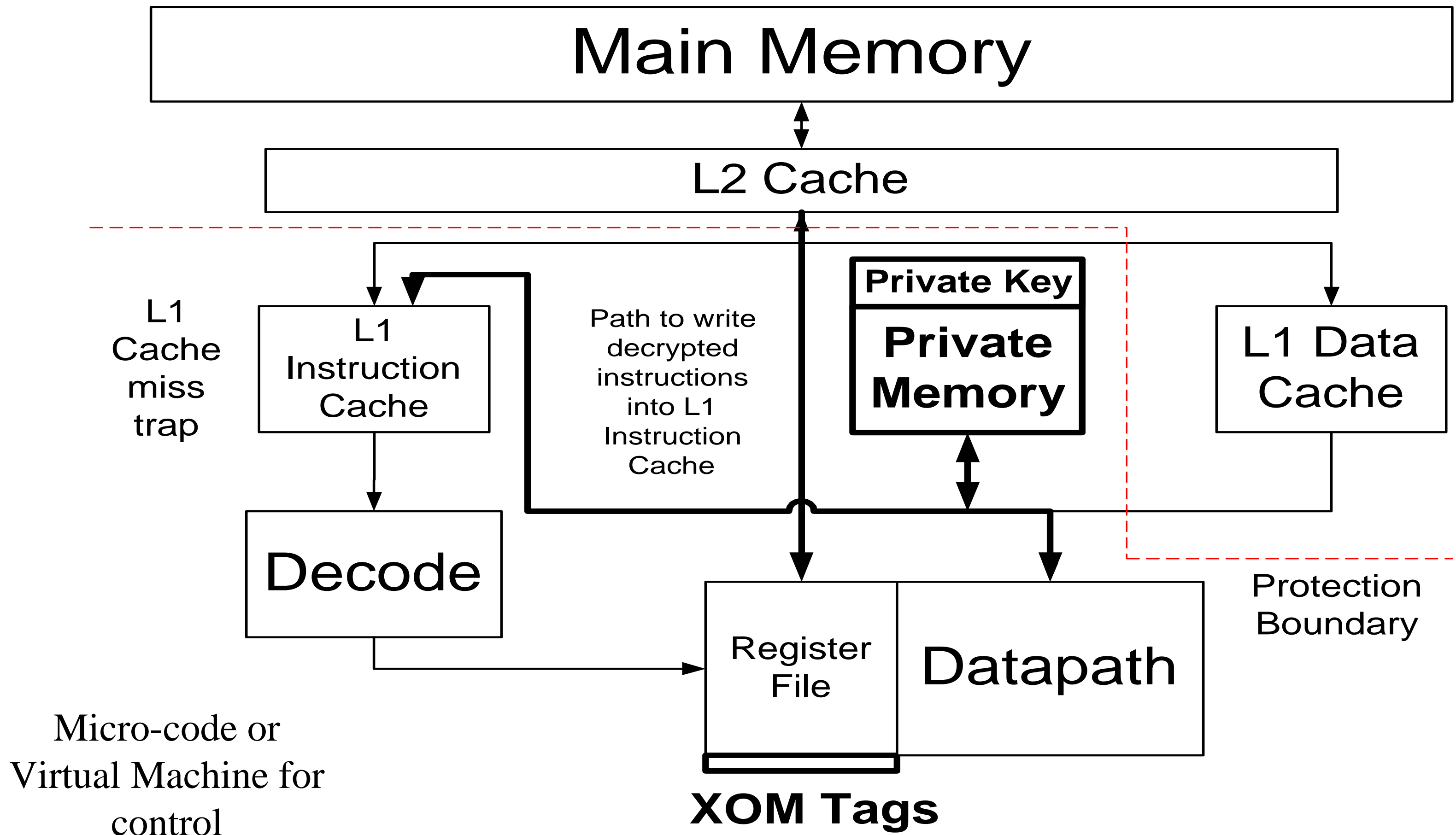
- Solution is to add an integrity hash to the encryption
  - *Message Authentication Code (MAC)*



Splicing Attacks and Replay Attacks

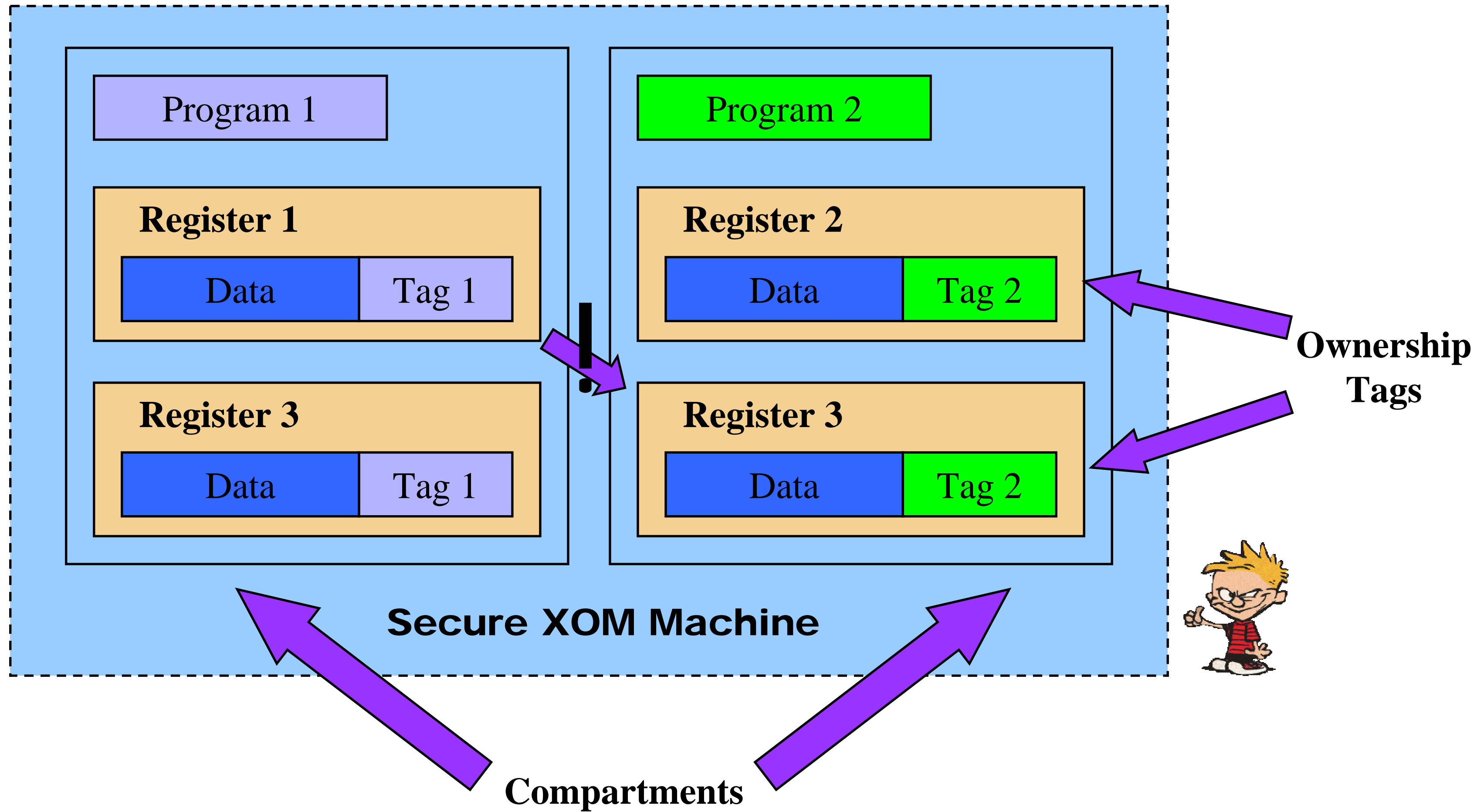
- Splicing Attacks
  - Attacker moves valid data from one location to another location
  - Add position dependent hash:
    - Virtual Address for secure load/stores
    - Register number for secure save/restores
- Replay Attacks
  - Attack records and reuses old register and memory values
  - Add a regenerative key to Key Table that is used for save/restores
  - Use protected registers to protect memory values

Required Hardware



Micro-code or
Virtual Machine for
control

XOM Provides Isolation



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

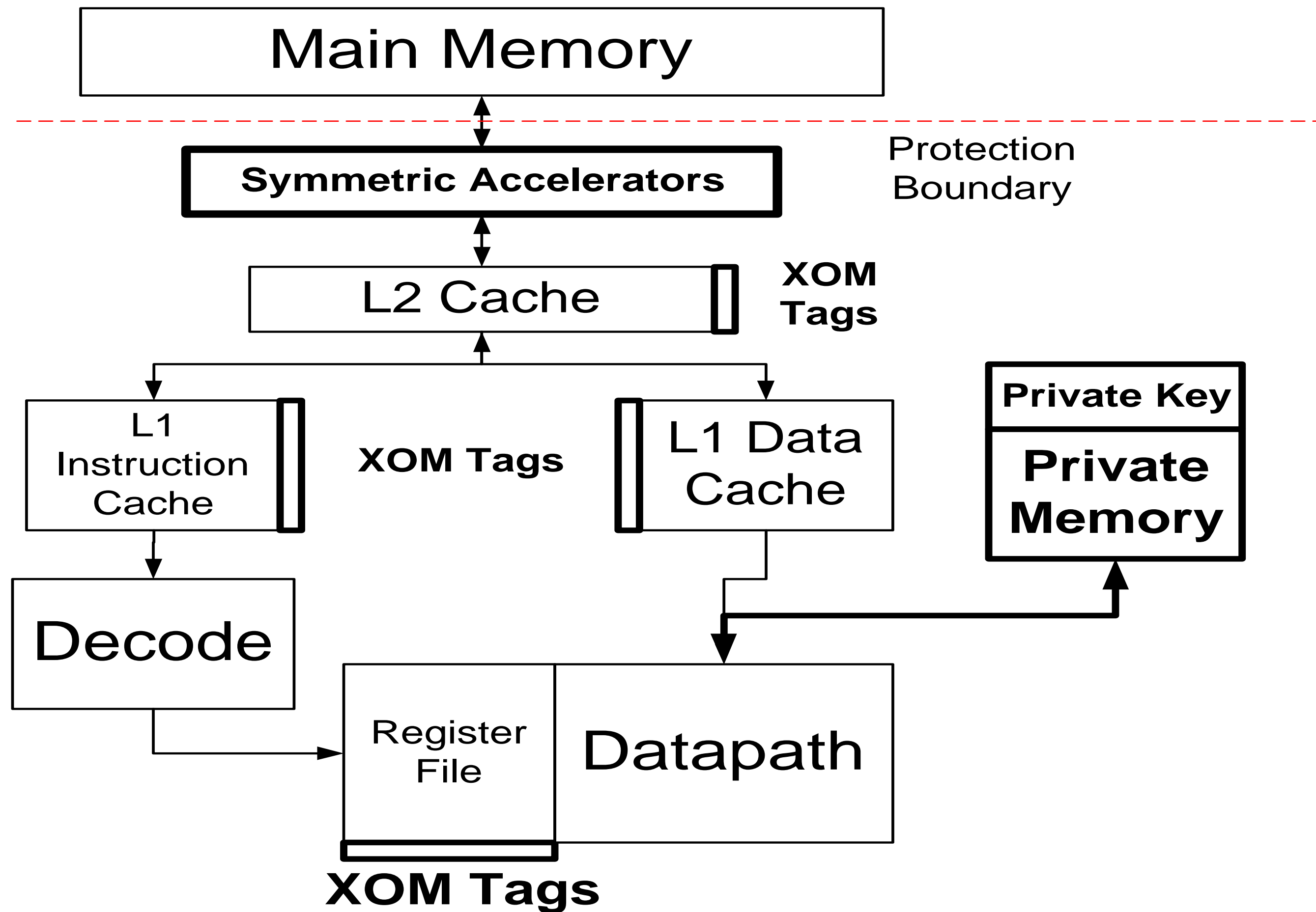
- The accesses to memory occur the most often

- 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

Full XOM Machine



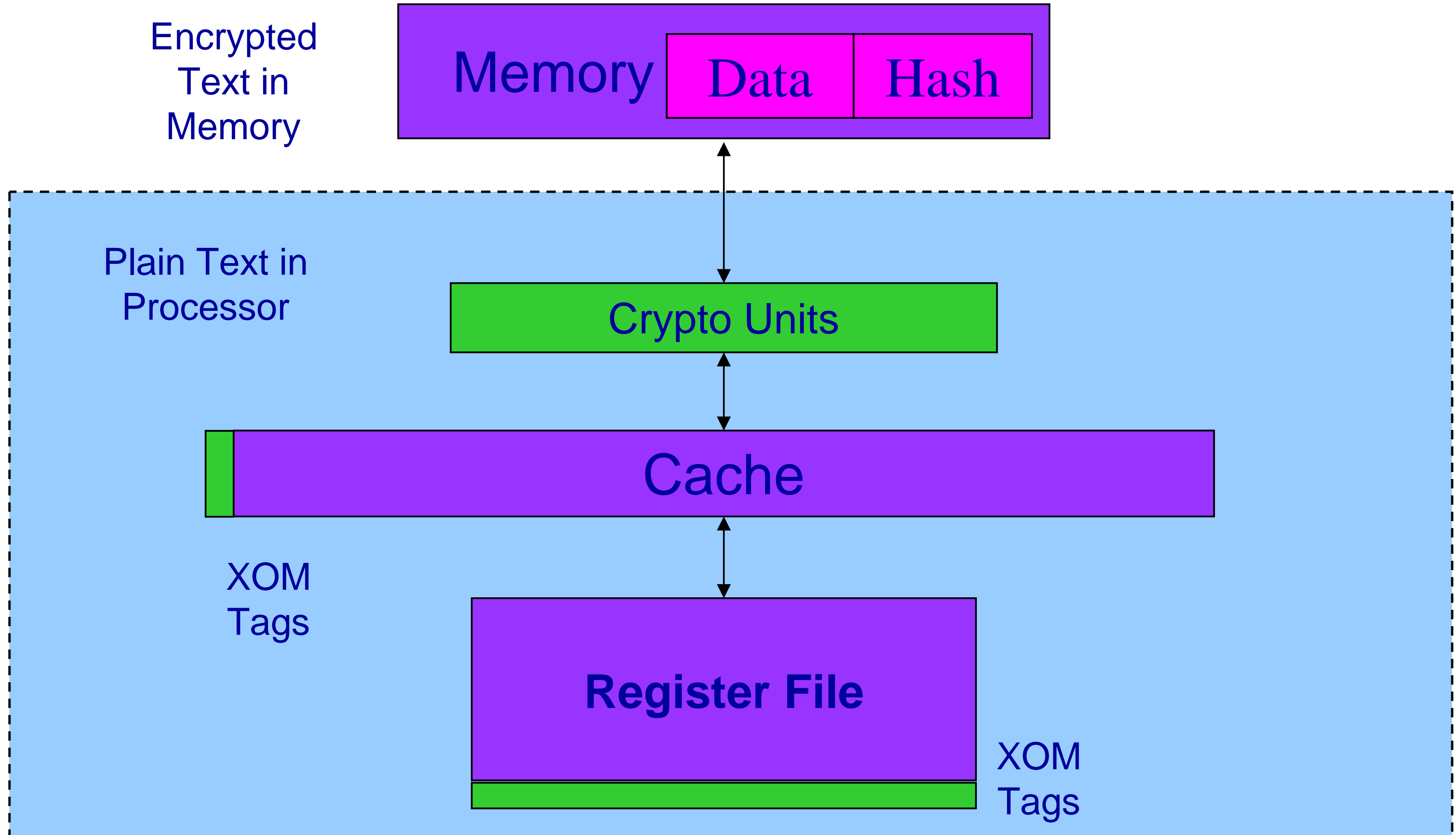
XOM architecture Summary

- Implement compartments with architectural support
- Trust only the processor; assume memory, 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

Model Checking XOM

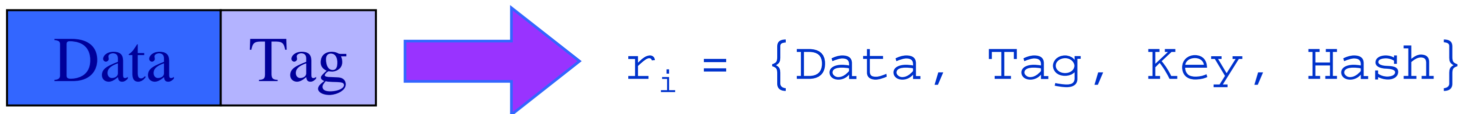
- XOM Model is a state machine:
  - State Vector
    - A set of all things on the chip that can hold state
    - Based on the Processor Hardware
  - Next-State Functions
    - A set of state transitions that the hardware can have
    - Derived from the instructions that can be executed on the processor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

XOM Processor Hardware

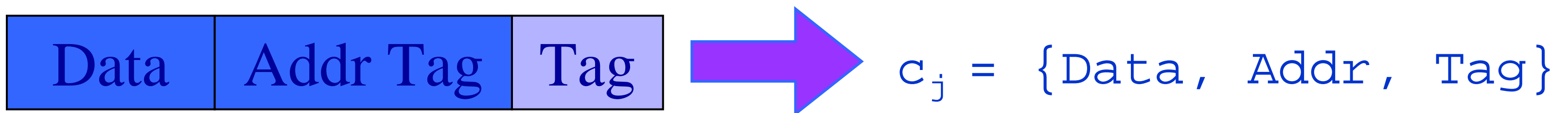


Modeling XOM

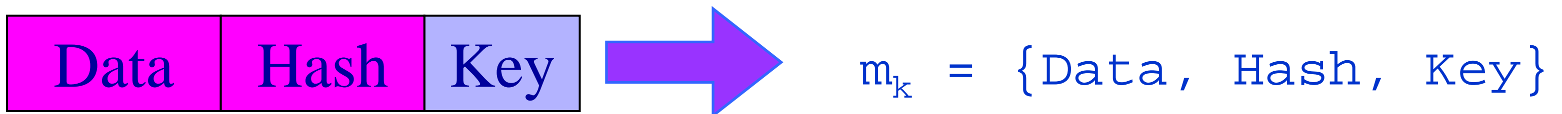
- Defining the state space:
  - Hardware units are modeled as arrays of elements
  - Number of elements is scaled down
- 3 Registers:



- 3 Cache Lines:



- 3 Memory Words:



XOM User Instructions

Instruction	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**

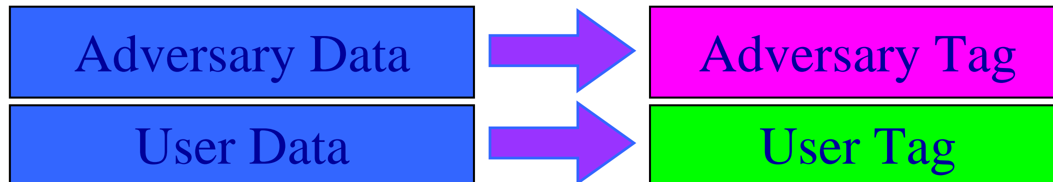
  c_k = {data = r_i.data, addr = j, tag = r_i.tag}

  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**

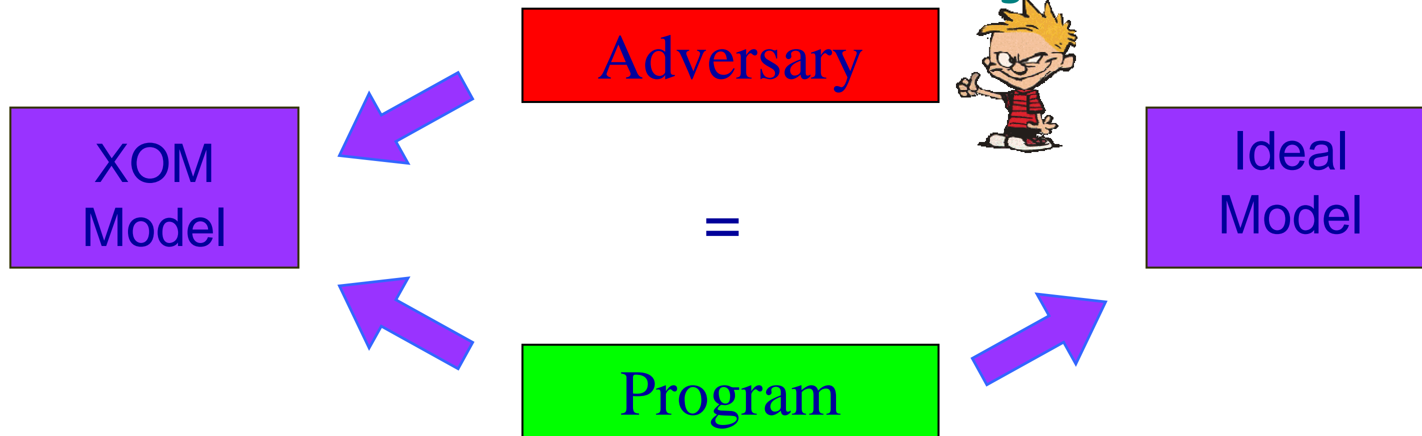
  check that: r_i`.tag = user`

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.m`$_i$`.data` = `user data` **then**

  check that: `ideal.m`$_i$`.data = xom.m`$_i$`.data`

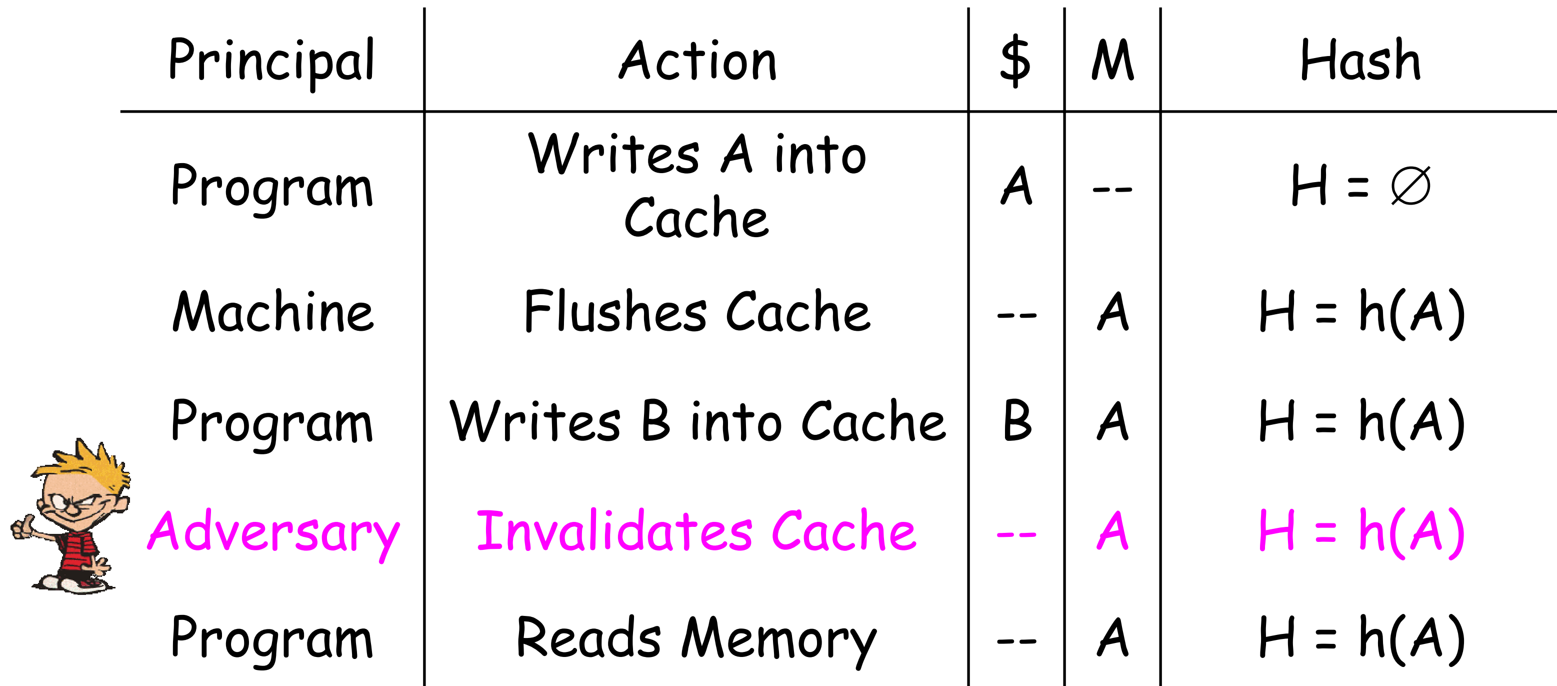
Checking for Correctness

- 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

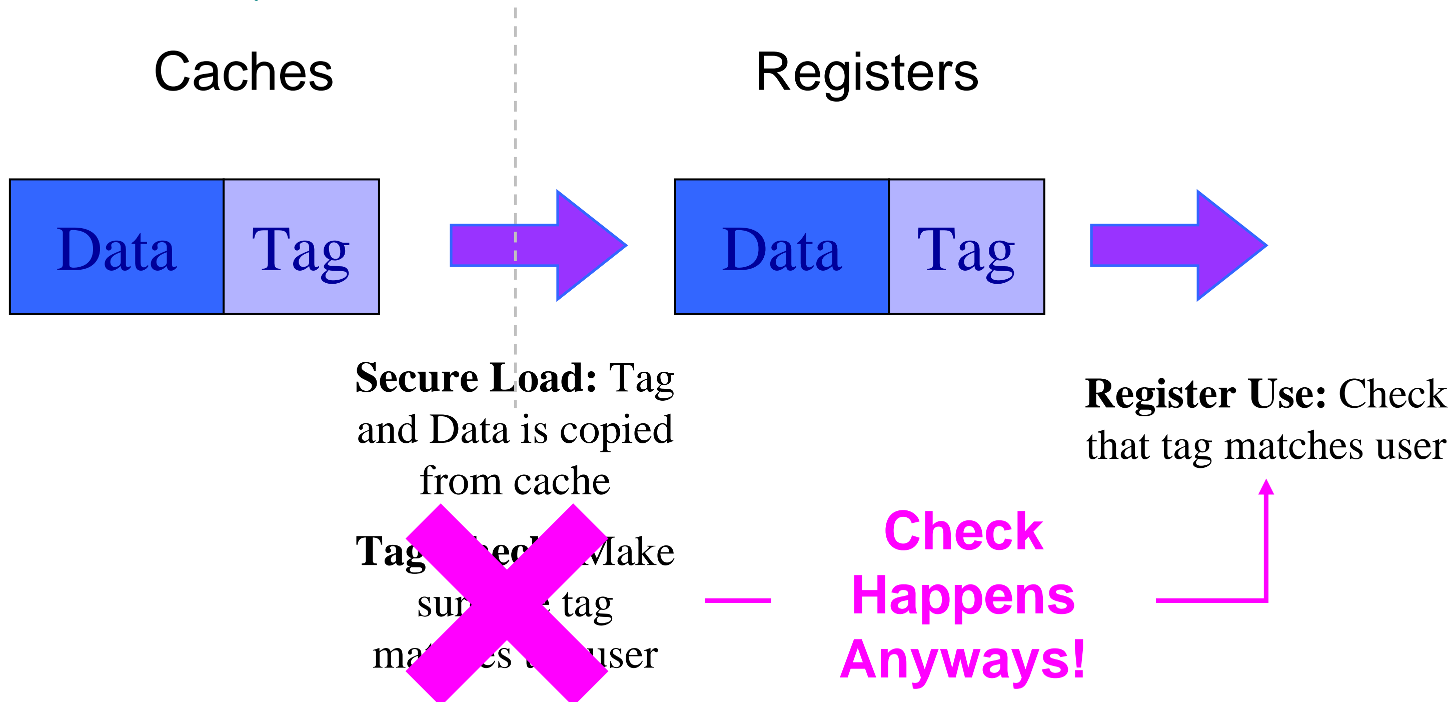


Principal	Action	$	M	Hash
Program	Writes A into Cache	A	--	H = ∅
Machine	Flushes Cache	--	A	H = h(A)
Program	Writes B into Cache	B	A	H = h(A)
Adversary	Invalidates Cache	--	A	H = h(A)
Program	Reads Memory	--	A	H = h(A)

- **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