

# Implementing an Untrusted Operating System on Trusted Hardware

David Lie

Chandramohan A. Thekkath

Mark Horowitz

University of Toronto, Microsoft Research,  
and Stanford University

# Protection in Systems: Hardware Approach

---

- Many platforms exist where trust is placed in hardware:
  - Trust Computing Platform Alliance (TCPA)
  - Microsoft's Palladium (NGSCB)
  - Intel LaGrande Architecture
- Advantages of hardware approach
  - Better tamper-resistance
  - Less dependent on OS correctness for security
- Trust in hardware means less trust (or no) in OS
  - We need to rethink operating system design



# Rethinking the Role of the OS

---

- A traditional OS performs both resource management and protection for applications
  - But now applications and OS are mutually suspicious
  - But applications don't trust OS to access their code and data
  - OS must be able to interrupt applications
- Use XOM (eXecute Only Memory) as an example platform
  - New operating system is called XOMOS
- XOMOS is UNIX-like, port of IRIX 6.5
  - Should support most standard UNIX applications



# XOM Hardware Architecture

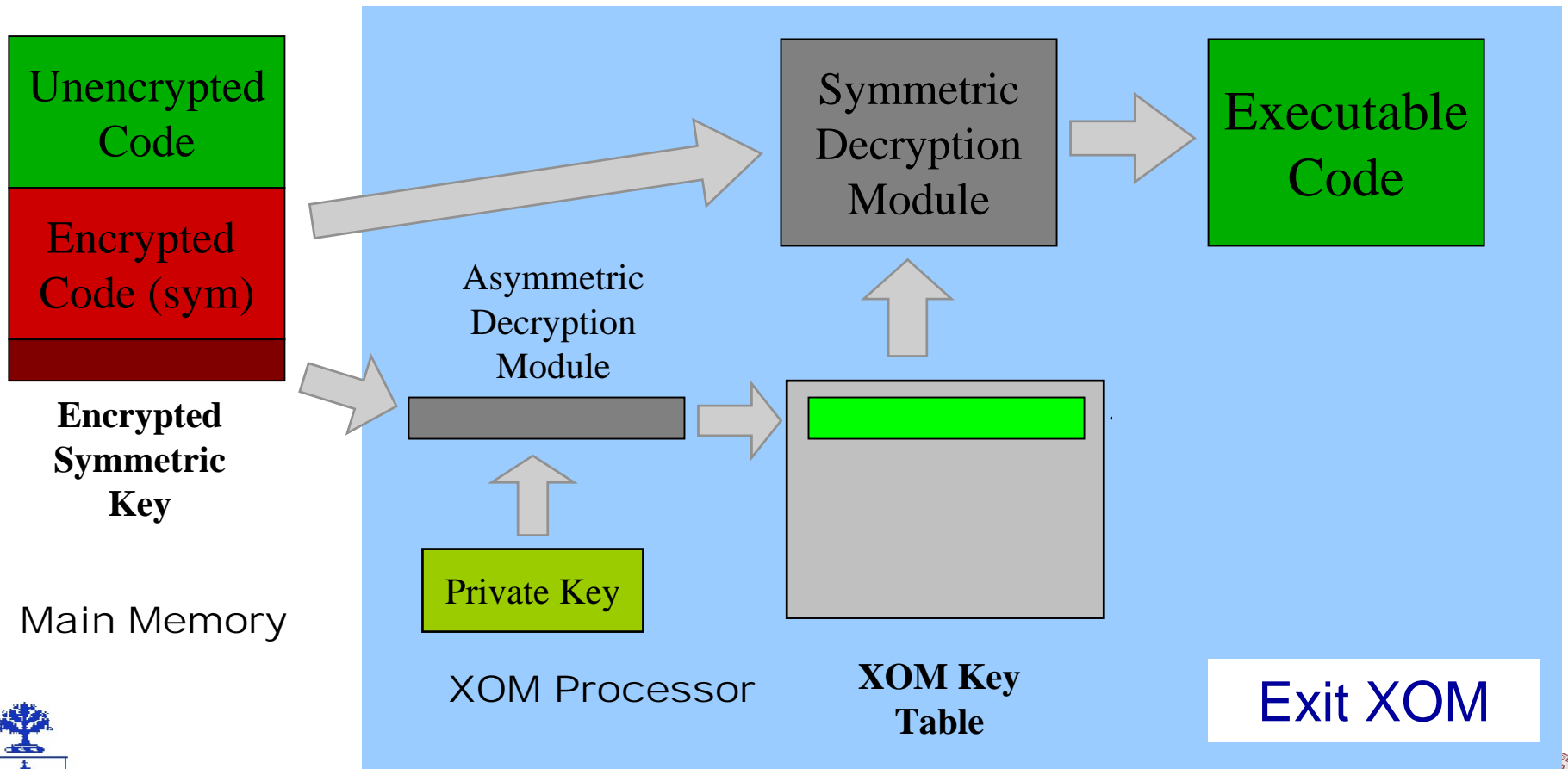
---

- XOM is implemented as a set of ISA extensions
- XOM Processor implements compartments for protection
  - Each compartment has a unique XOM ID
  - Architectural tags to control access to data on-chip
  - Cryptographic mechanisms protect data off-chip
- XOM Processor executes encrypted code
  - Combination of asymmetric/symmetric ciphers
  - Private key embedded on-chip hardware

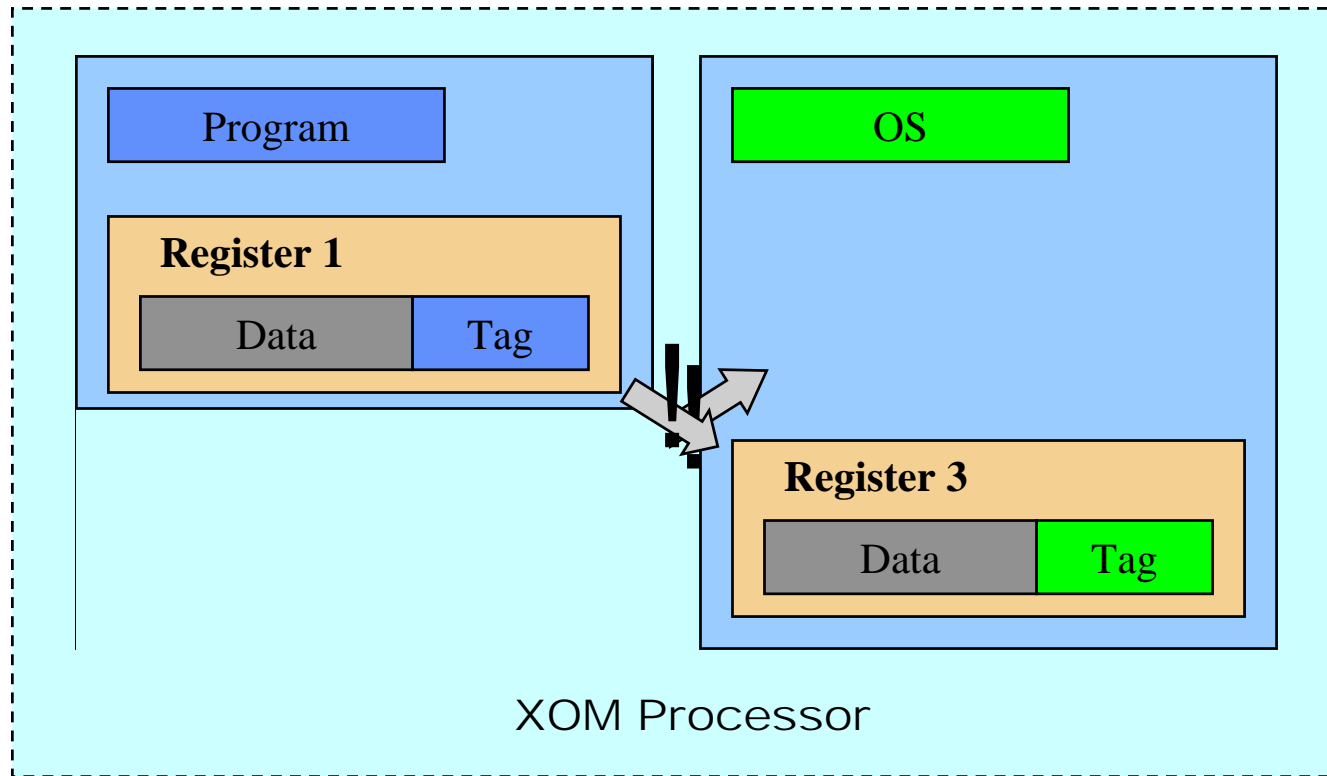


# Entering Secure Execution

- Applications Enter and Exit secure execution via instructions



# Implementing Compartments



- Problem: How does OS save registers during a trap or interrupt?



# XOMOS Overview

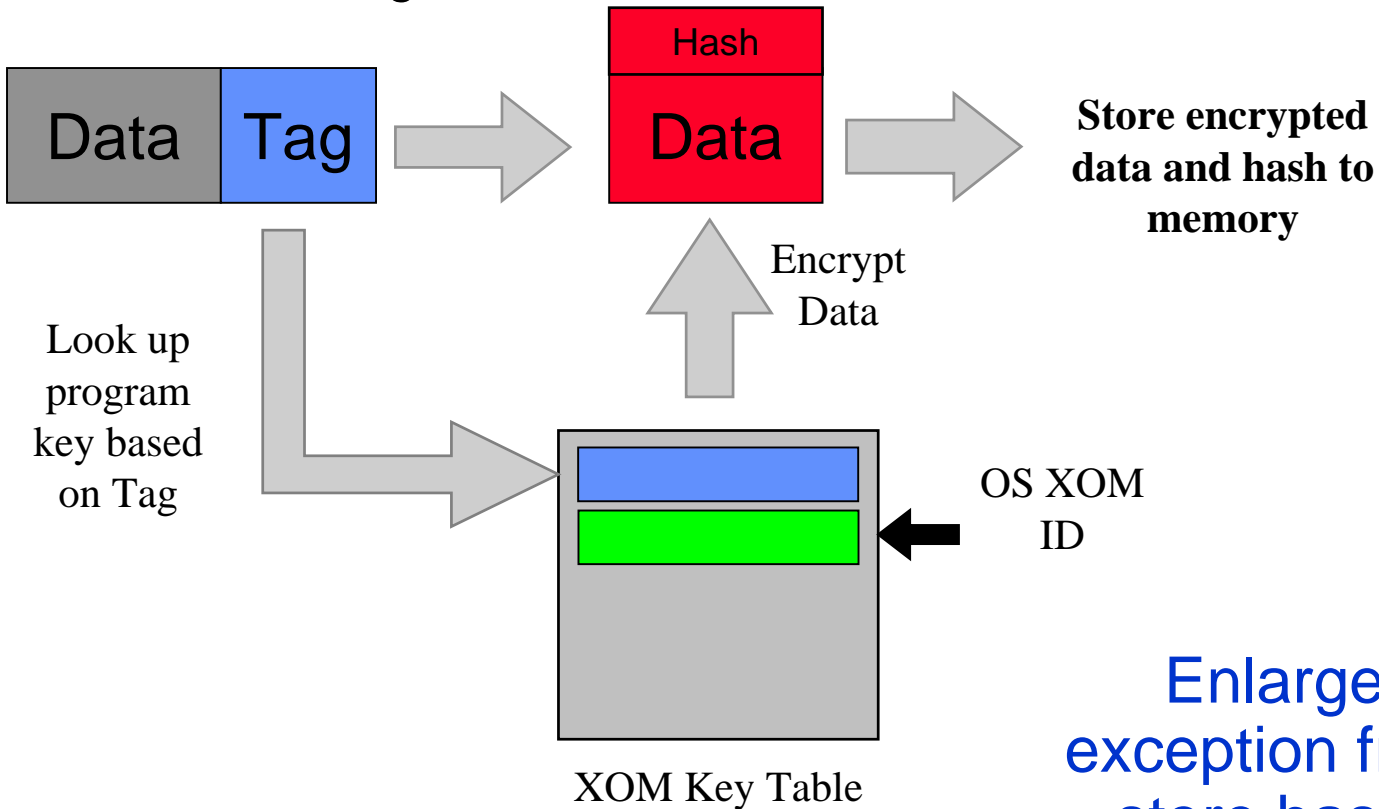
---

- Implement XOM CPU in SimOS hardware simulator
  - MIPS-based architecture with XOM ISA extensions
- Ported the IRIX 6.5 Operating System to run on XOM processor
- Main areas that need modification:
  - Resource management of secure data
    - **Saving Registers on Interrupt, Memory Management**
  - Need support for XOM Key Table
    - Loading/Unloading, Management
  - Compatibility with original system
    - **Fork, Signal Handling, Shared Libraries/System Calls**



# Saving Registers

- OS uses *save register* instruction

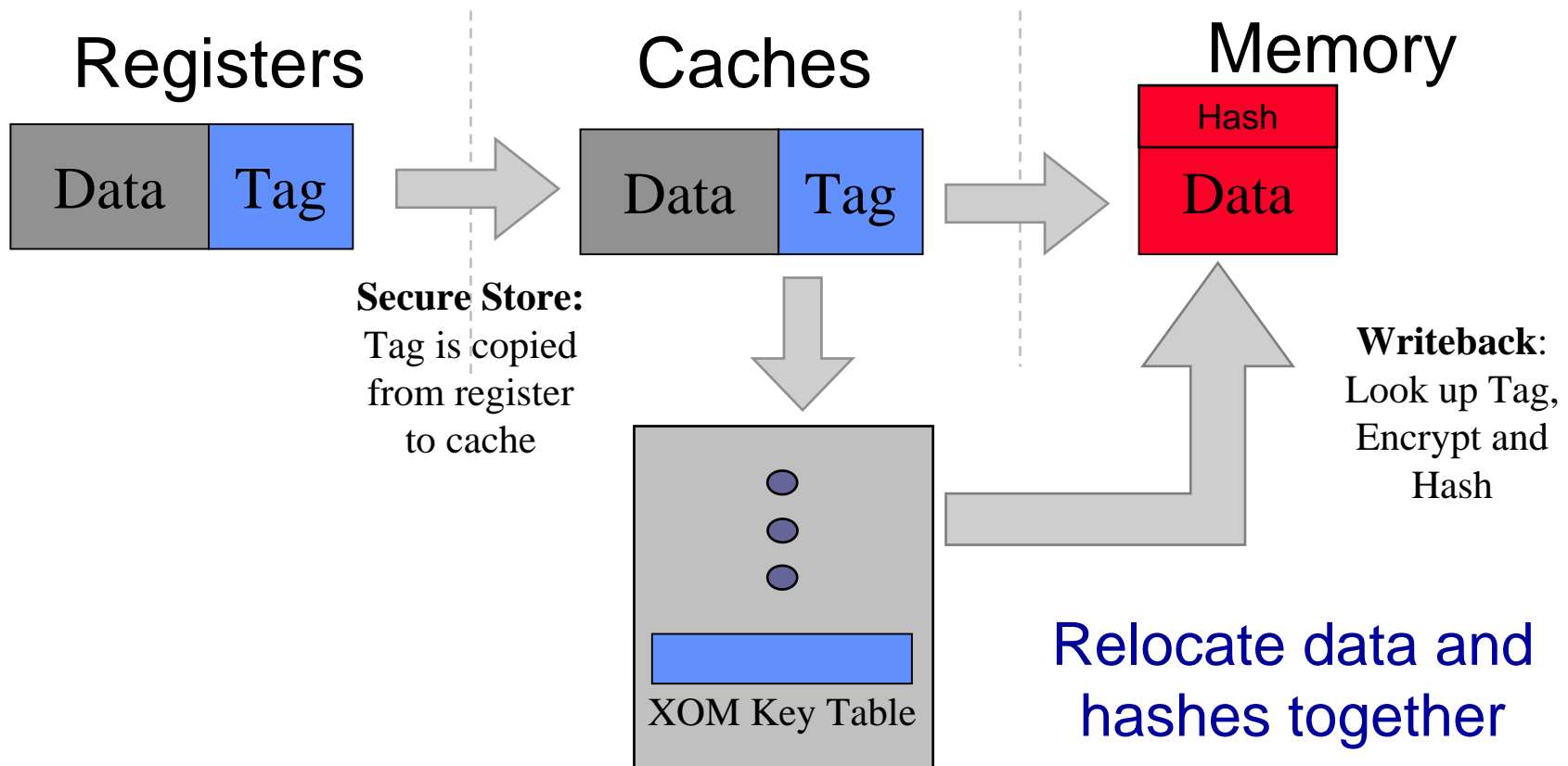


Enlarge the exception frame to store hash and XOM ID



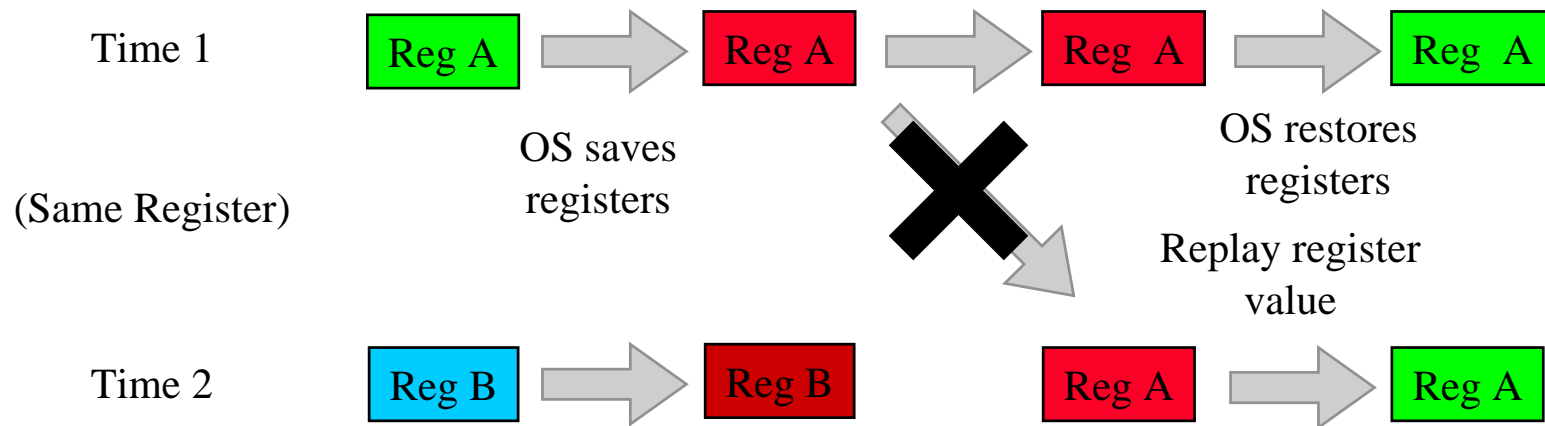
# Protecting and Managing Memory

- Applications use *secure store* instruction



# Replay Attacks and Fork

- An OS Fork must be differentiated from a Replay Attack
  - Replay Attack: OS duplicates registers and replays them
  - Fork: OS must duplicate register set and restore them



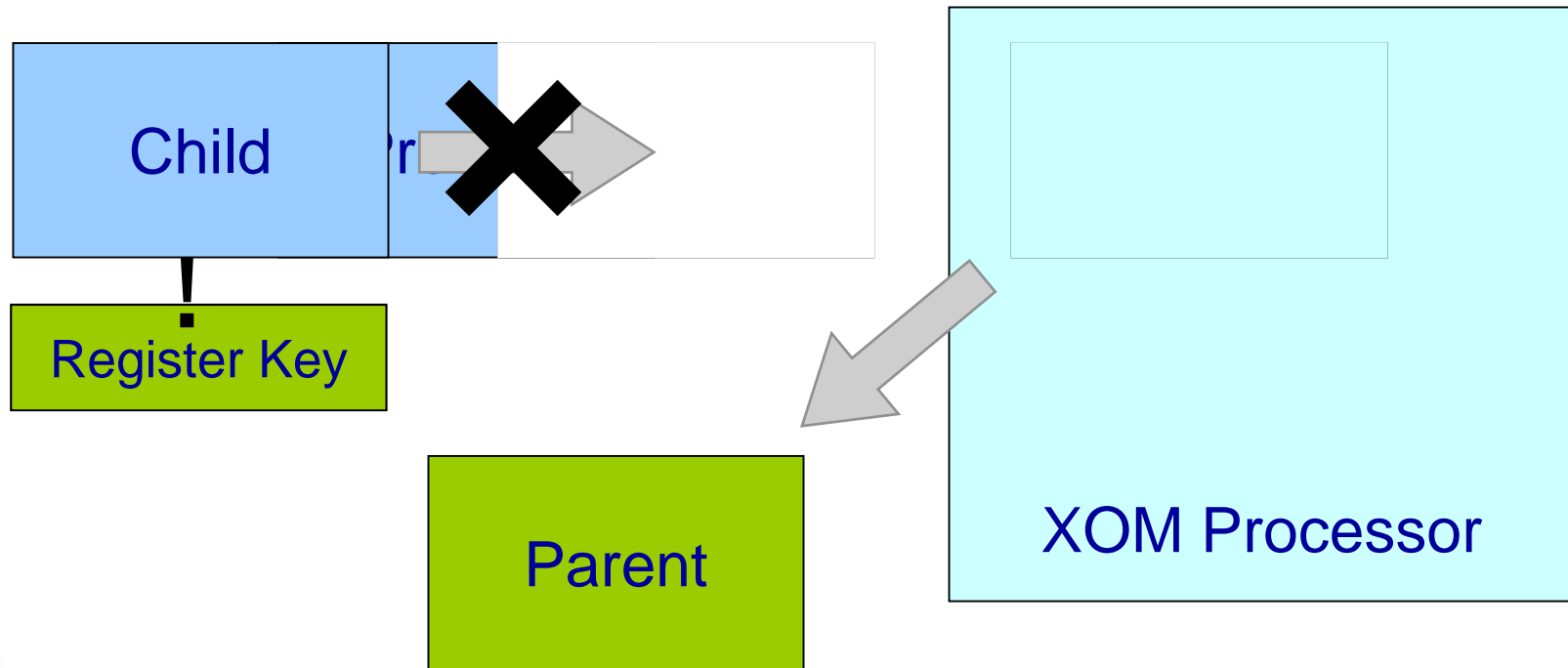
- Revoke and regenerate register keys on every interrupt
- This hardware defense breaks traditional fork code



# Naïve Fork Implementation

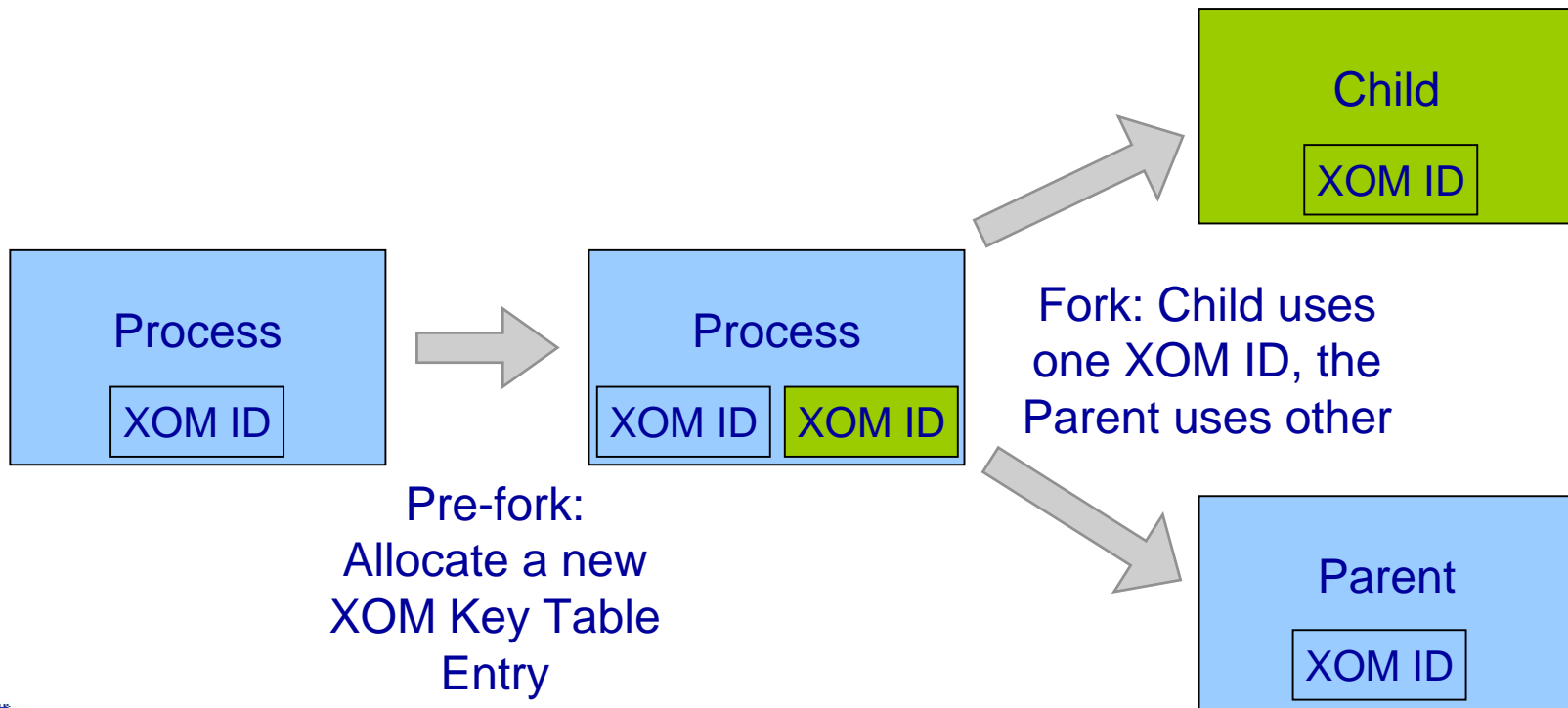
- Parent and Child are exactly the same, have the same XOM ID
- Problem: Both threads use the same register key

Fork



# Fork Solution

- OS must allocate a new XOM ID for child process
  - Child has a separate XOM ID and register key



# XOM Performance Simulation

---

- The SimOS Simulator:
  - Performance modeling processor, caches, memory and disk
- Simulation Parameters
  - Memory latency increases 10% due to encryption
    - Based on AES (Rijndael) cipher implementation
  - Public key decryption: 400,000 cycles
    - Once per process creation



# Operating System Overhead

	Total Cycles			Cache Misses		
	IRIX	XOM	OV	IRIX	XOM	OV
System Call	5.7K	6.3K	11%	4.2	5.6	33%
Signal Handling	31K	40K	27%	38.4	48.3	26%
Fork	138K	126K	-9%	1035	1058	2%

- OS overhead due to poor cache behavior:
  - More cache misses due to bigger code/data footprint
  - Should check to see if OS needs to do extra work
    - Work is only needed if application is in compartment



# End-to-end Application Overhead

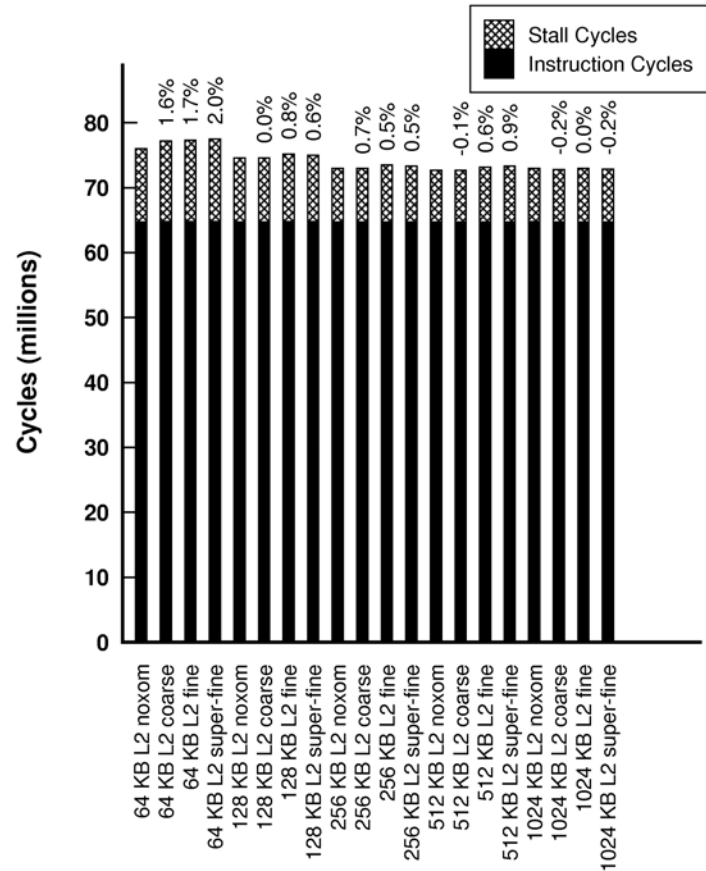
---

- Applications use Enter/Exit XOM to execute sections securely
  - How large to make secure sections?
  - Executing in compartment has memory access overhead
  - Entering/leaving compartment can cause cache misses
- Different compartment sizes: granularities
  - 3 levels: Coarse, Fine, Super-Fine
  - Applications are MPEG Decode and RSA Encryption



# Application Overhead

- Turns out that granularity has little impact on performance overhead!
- Overheads for most applications turn out to be less than 5%
  - If cache behavior is similar
- Coarse application has smallest effect on cache behavior



# Conclusions

---

- Possible to construct an Untrusted Operating System
  - Modest modifications
    - Port required 1900 lines of low level code
  - Modifications limited to low level interface to the hardware
    - Able to support most traditional OS mechanisms
- Performance hit is modest ( $< 5\%$ )
  - Can run applications completely inside compartment
    - As long as cache behavior is unaffected
  - This is because of hardware acceleration

