# Overshadow:
## A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems

Mike Chen Tal Garfinkel E. Christopher Lewis
Pratap Subrahmanyam Carl A. Waldspurger
*VMware, Inc.*

Dan Boneh — *Stanford*
Jeffrey Dwoskin — *Princeton*
Dan R.K. Ports — *MIT*

Carl Waldspurger
*VMware R&D*

ASPLOS '08
March 3, 2008

# Motivation

## Applications Handle Sensitive Data

- Financial, medical, insurance, military …

## Commodity Systems Vulnerable

- Large and complex TCB, broad attack surfaces
- OS kernel, file system, daemons, services …
- Hard to configure, manage, maintain
- Privilege escalation ⇒ game over

## Data Theft Soaring

- Reached "unprecedented levels" in 2007
- Identity theft, breach notification laws …

# Limitations of Existing Solutions

## Rewrite OS / Applications

- Split into low- and high-assurance portions
  *e.g.* microkernels, Microsoft Palladium/NGSCB
- Expensive, high barriers to adoption

## Multiple Virtual Machines

- Trusted/untrusted or specialized VMs (*e.g.* Proxos, Terra)
- Cumbersome, still vulnerable to OS compromise

## Hardware Approaches

- Special-purpose secure co-processors (*e.g.* IBM 4758)
- XOM and SP processor architectures
- Require substantial modifications to hardware/OS/apps

# Goals

**Protect Application Data**

- Privacy and integrity
- In memory and on disk

**Remove OS from TCB**

- Provide last line of defense
- Even if attacker compromises guest OS

**Backwards Compatibility**

- Unmodified commodity OS
- Unmodified application binary

**Non-Goal: Availability**

# Overshadow Topics

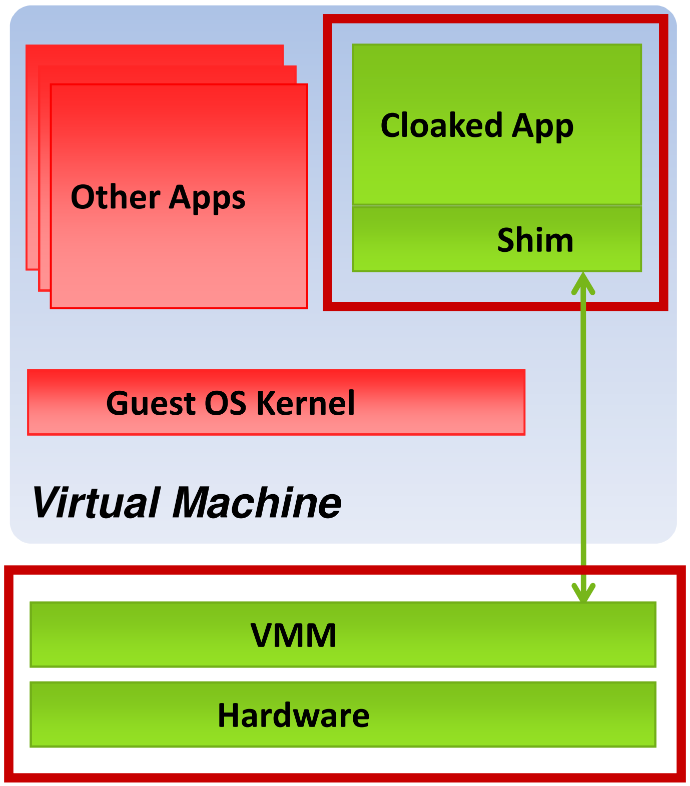
## Focus of Talk

- Protecting application memory
- Secure control transfers
- Adapting system call interface
- Performance

## In Paper

- Secure context identification
- Managing protection metadata
- Implications of malicious system call interface (work in progress)

# Overshadow Architecture




*Two Virtualization Barriers*

**VMM Protects App Memory**

- New virtualization barrier
- App trusts VMM, but not OS

**Cloaking: Two Views of Memory**

- App sees normal view
- OS sees encrypted view

**Shim: App/OS Interactions**

- Interposes on system calls, interrupts, faults, signals
- Transparent to application



vmware

# Memory Mapping: OS

virtual → physical

*OS page table*





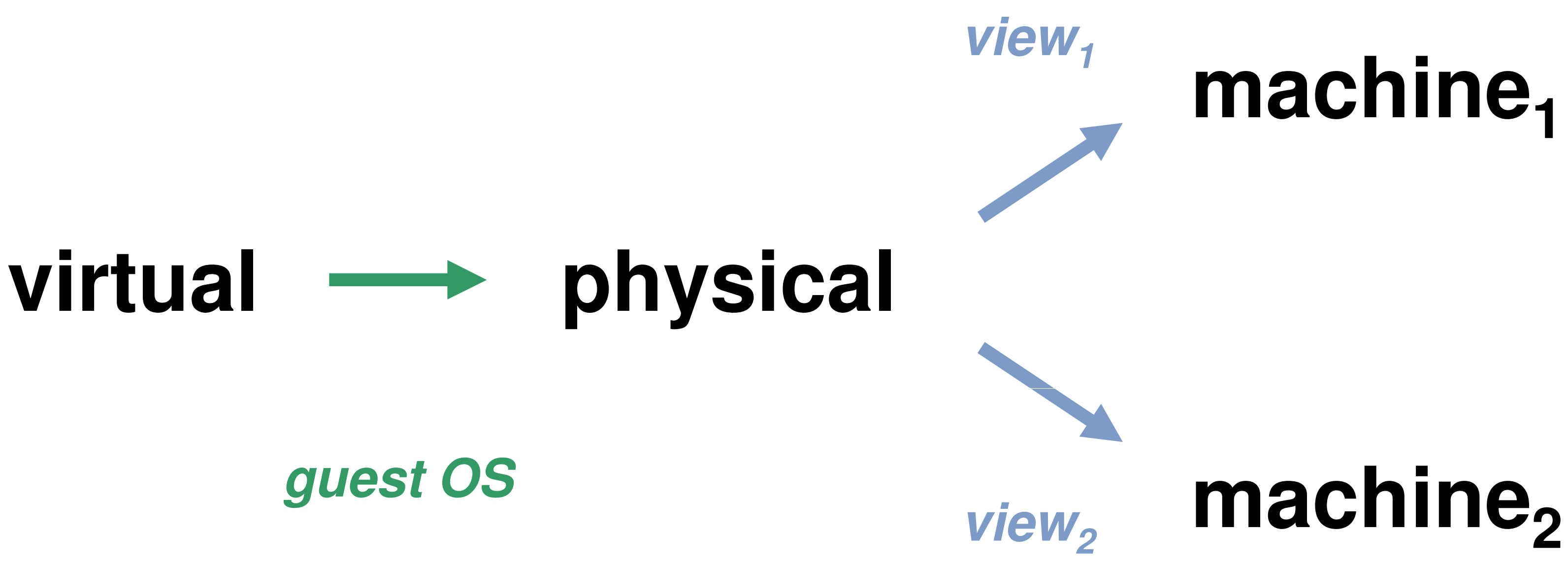
vmware

# Memory Mapping: VMM

**virtual** → **physical** → **machine**

*guest OS* *vmm*

# Multi-Shadowing: Context-Dependent Views



# Cloaking: Multi-Shadowing + Cryptography

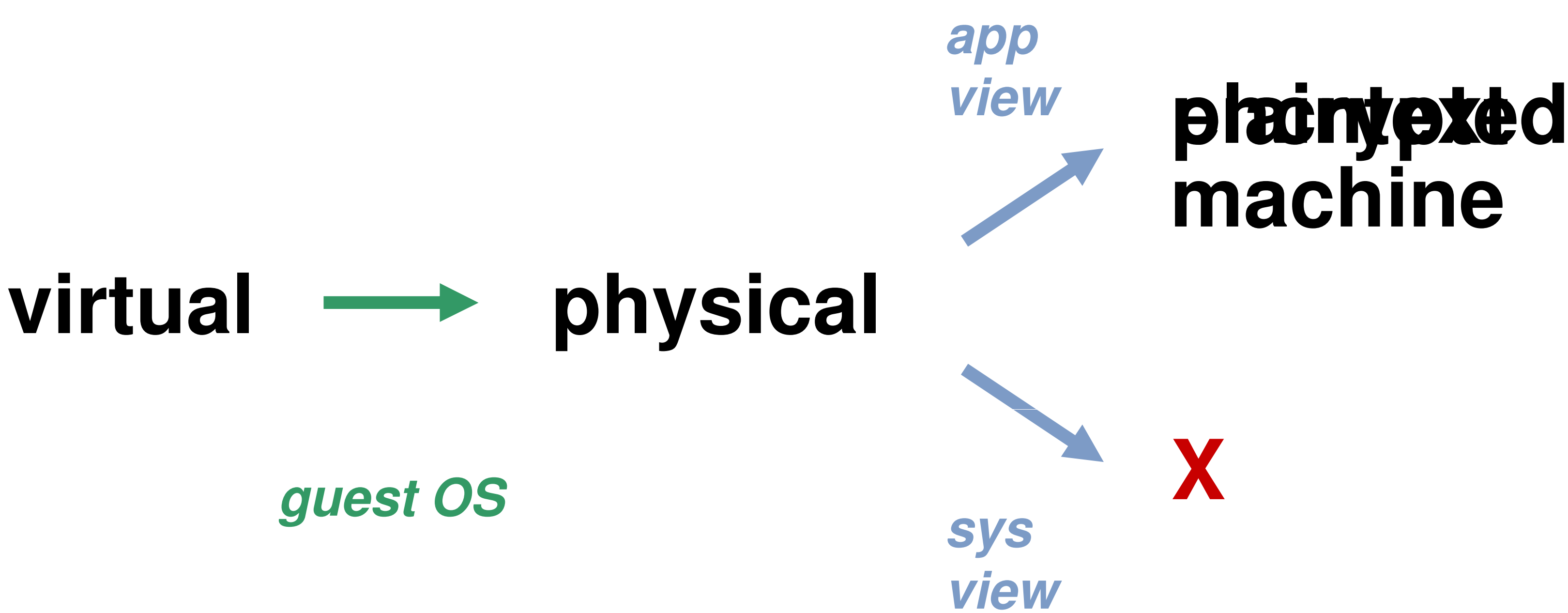
virtual → physical

*guest OS*

*app view* → plaintext machine

*sys view* → X unmapped

## Cloaking: System Accesses Page



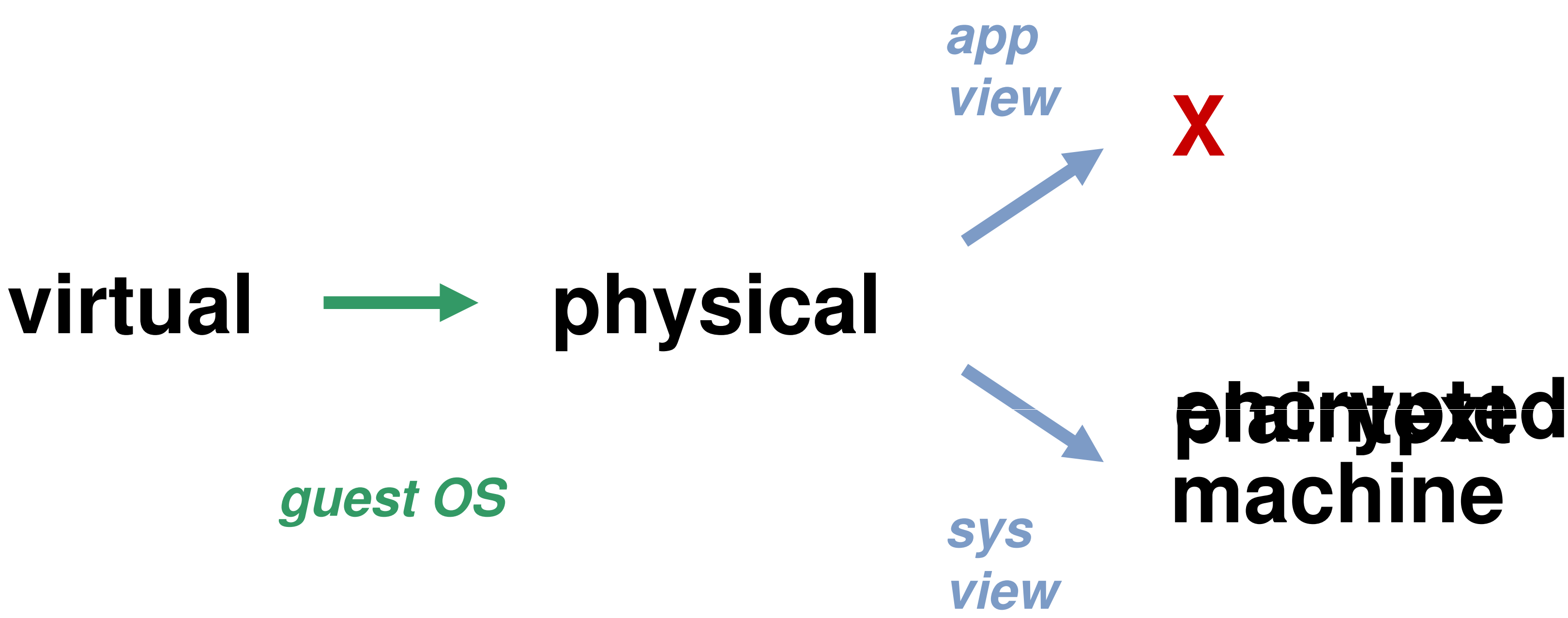

**Fault into VMM: encrypt/hash contents, remap**





vmware

# Cloaking: Application Accesses Page




## Fault into VMM: verify hash, decrypt, remap

# Cloaking Application Resources

## Basic Strategy

- Protect existing memory-mapped objects
  *e.g.* stack, heap, mapped files, shared mmaps
- Make everything else look like one
  *e.g.* emulate file read/write using mmap

## OS Still Manages Application Resources

- Including demand-paged application memory
- Moves cloaked data without seeing plaintext contents
- Encryption/decryption typically infrequent

# Shim: Supporting Unmodified Applications

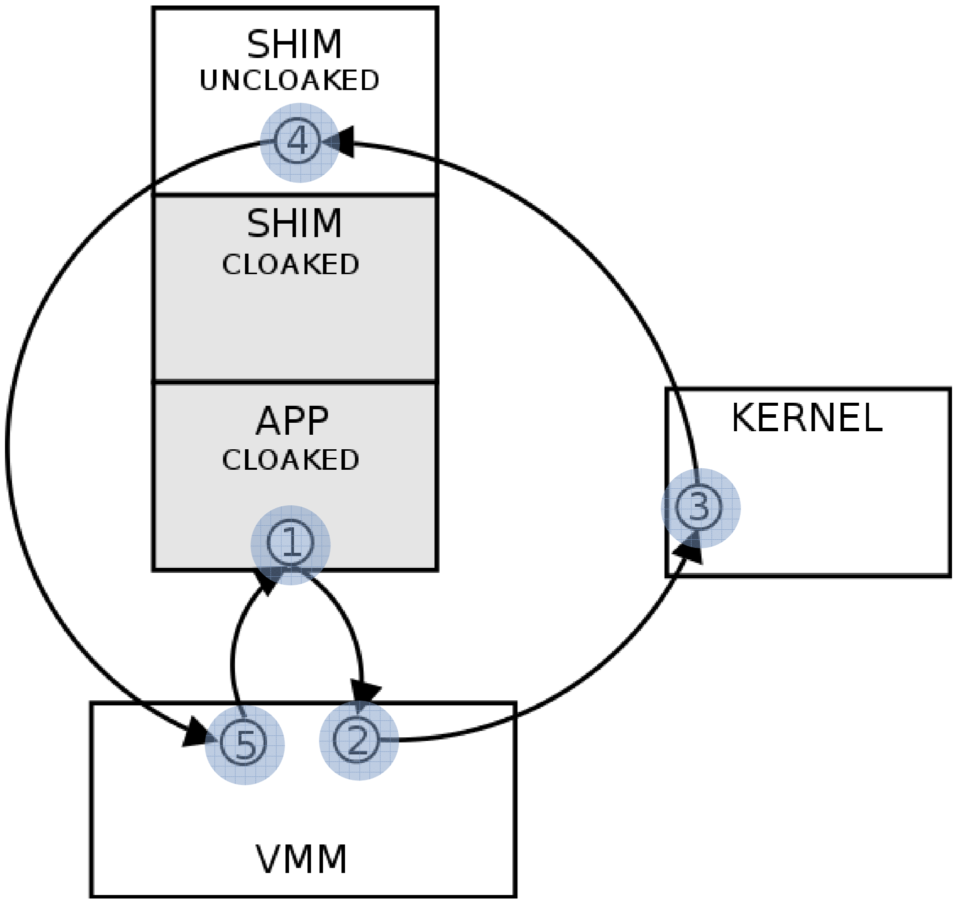
## Challenges

- Securely identify which app is running
- Secure control transfers between OS and app
- Adapting system calls

## Solution: Shim

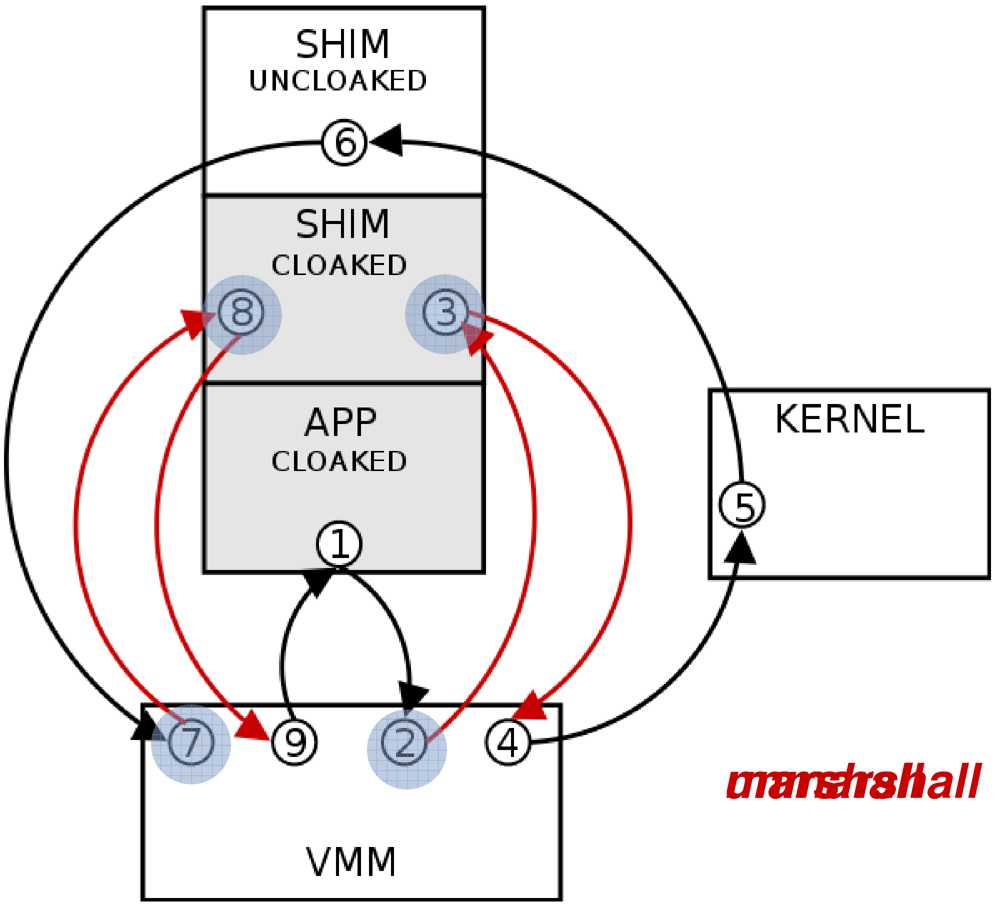
- OS-specific user-level program
- Linked into application address space
- Mostly cloaked, plus uncloaked trampolines and buffers
- Communicates with VMM via hypercalls

# Shim: Handling Faults and Interrupts




1. **App is executing**
2. **Fault traps into VMM**
   - Saves and scrubs registers
   - Sets up trampoline to shim
   - Transfers control to kernel
3. **Kernel executes**
   - Handles fault as usual
   - Returns to shim via trampoline
4. **Shim hypercalls into VMM**
   - Resume cloaked execution
5. **VMM returns to app**
   - Restores registers
   - Transfers control to app

# Shim: Handling System Calls




## Extra Transitions

- Superset of fault handling
- Handlers in cloaked shim interpose on system calls

## System Call Adaptation

- Arguments may be pointers to cloaked memory
- Marshall and unmarshall via buffer in uncloaked shim
- More complex: pipes, signals, fork, file I/O

# Protecting Data Integrity

## Challenges

- Enforce integrity, ordering, freshness
- For code, data, memory-mapped files …

## VMM Manages Per-Page Metadata

- Tracks what's "supposed to be" in each memory page
- IV – randomly-generated initialization vector
- H – secure integrity hash

# Implementation

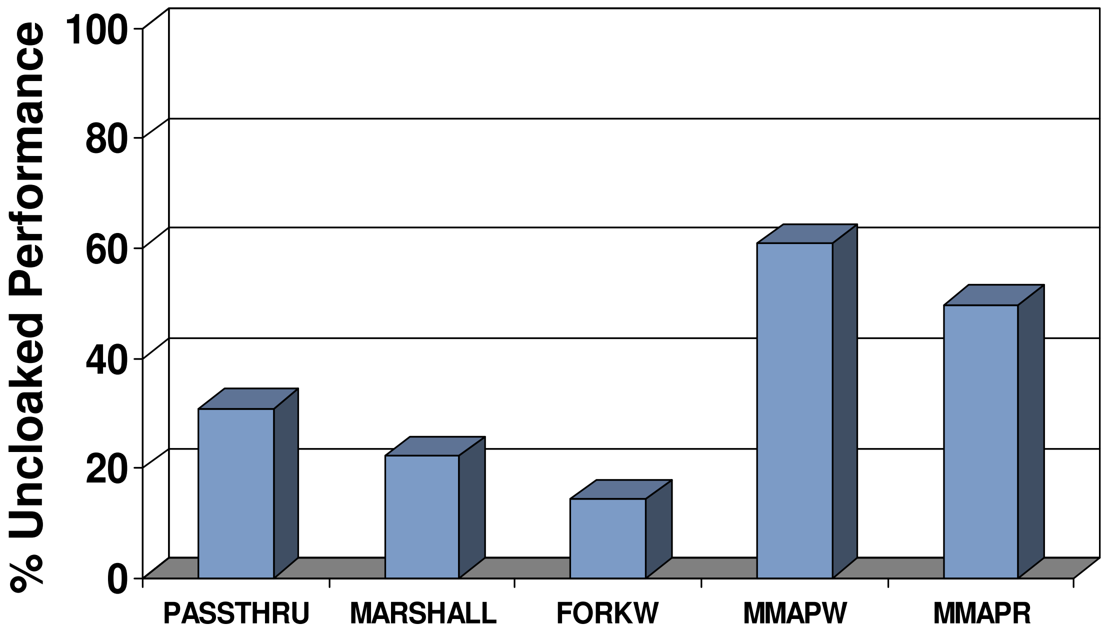
## Overshadow System

- Based on 32-bit x86 VMware VMM
- Shim for Linux 2.6.x guest OS
- Full cloaking of application code, data, files
- Lines of code: + 6600 to VMM, ~ 13100 in shim
- Not heavily optimized

## Runs Real Applications

- Apache web server, PostgreSQL database
- Emacs, bash, perl, gcc, …

# Microbenchmark Performance




**System Calls**

- Simple PASSTHRU
- MARSHALL args

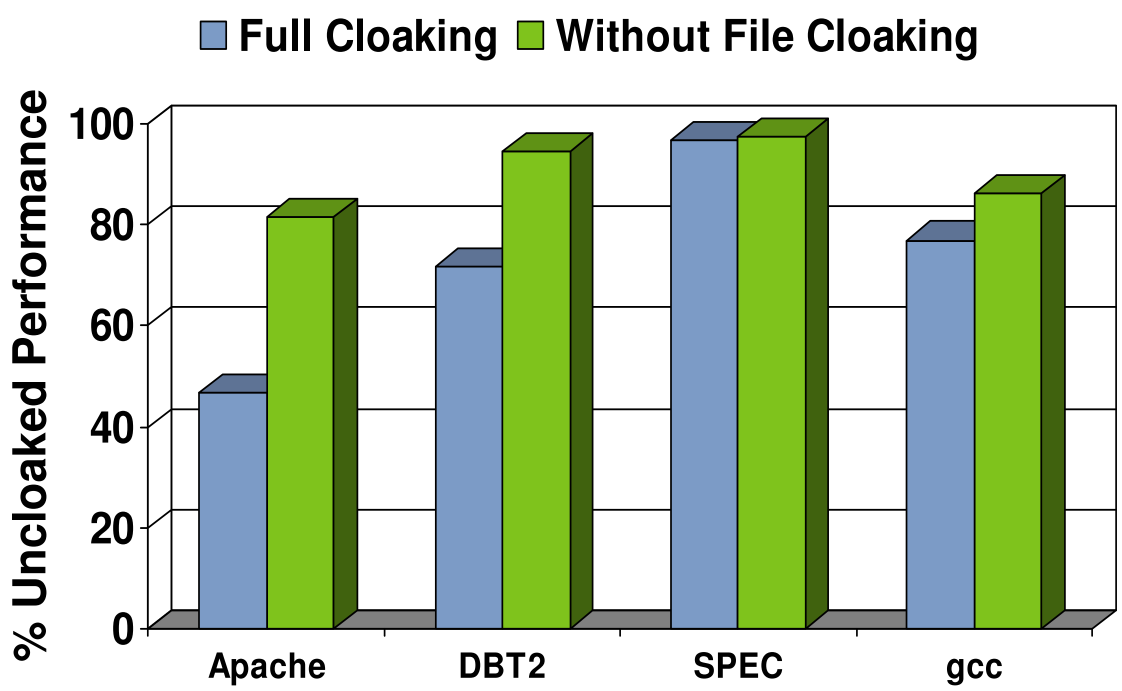
**Processes**

- FORKW – fork/wait process creation, COW overheads

**File-Backed mmaps**

- MMAPW – write word per page, flush to disk
- MMAPR – read words back from buffer cache

# Benchmark Performance




**Web**

- Apache web server caching disabled
- Remote load generator ab benchmark tool

**Database**

- PostgresSQL server DBT2 benchmark

**Compute**

- SPECint CPU2006
- gcc – worst individual SPEC benchmark

# Conclusions

## Promising New Approach

- VM-based protection of application data
- Privacy and integrity, even if OS compromised
- Backwards compatible

## Powerful New Mechanisms

- Multi-shadowing, cloaking
- Shim extends reach of VMM

## Future Directions

- Security implications of a malicious OS
- Additional uses of multi-shadowing

# Questions?

## For More Information

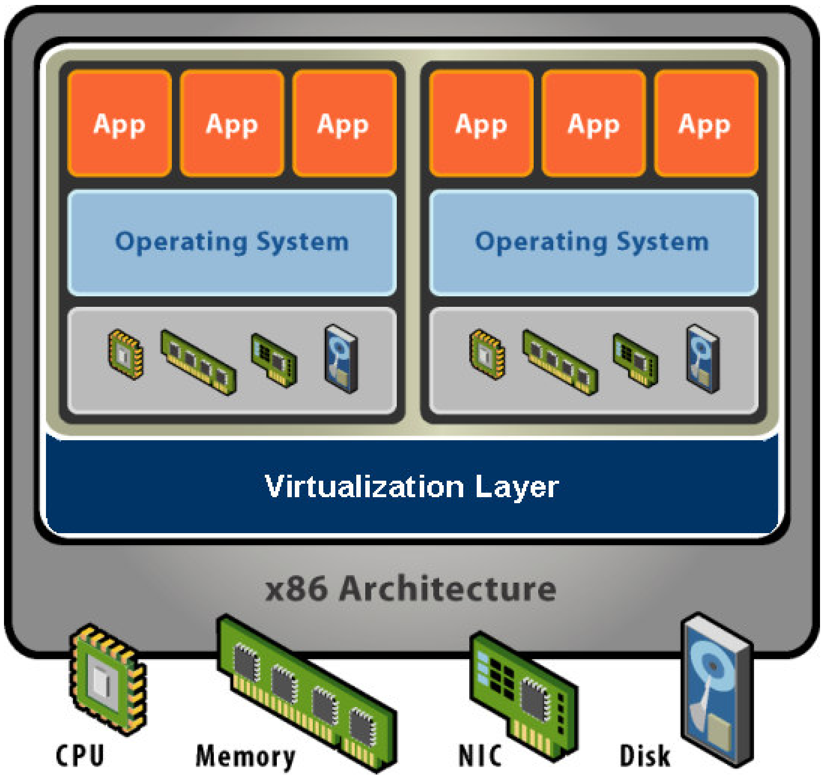
- Read the paper
- Send feedback to mailing list
  overshadow@vmware.com

## Job Opportunities

- VMware is hiring!
- Interns and full-time positions
- Feel free to contact me directly
  carl@vmware.com

# Backup Slides

# What is a Virtual Machine?



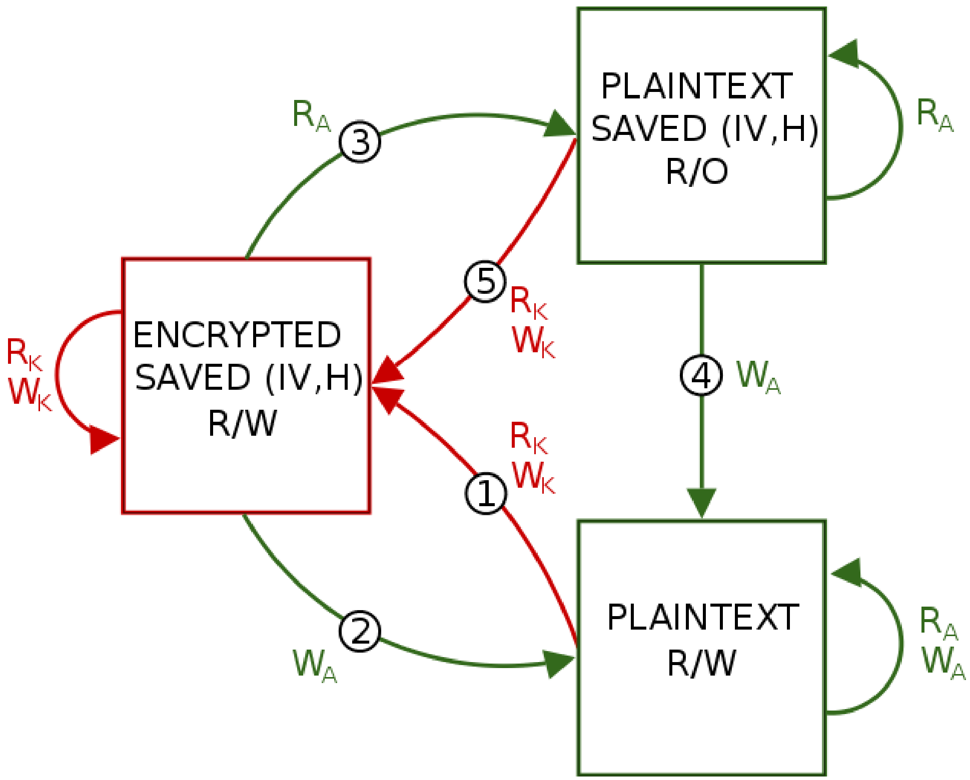

## Hardware-Level Abstraction

- Virtual hardware: processors, memory, chipset, I/O devices, etc.
- Encapsulates all OS and application state

## Virtualization Software

- Extra level of indirection decouples hardware and OS
- Multiplexes physical hardware across multiple "guest" VMs
- Strong isolation between VMs
- Manages physical resources, improves utilization

# Basic Cloaking Protocol




## State Transition Diagram

- Single cloaked page
- Privacy and integrity

## Single Page, Two Views

- App (A) sees plaintext via application shadow
- Kernel (K) sees ciphertext via system shadow

## Protection Metadata

- IV – randomly-generated initialization vector
- H – secure hash

# Secure Context Identification

## Application Contexts

- Must identify uniquely to switch shadow page tables
- Must work even with adversarial OS

## Shim-Based Approach

- Cloaked Thread Context (CTC) in cloaked shim
- Initialized at startup to contain ASID and random value
- Random value is protected in cloaked memory
- Transitions from uncloaked to cloaked execution use self-identifying hypercalls with pointer to CTC
- VMM verifies expected ASID and random value in CTC

# Cloaked File I/O

## Interpose on I/O System Calls

- Read, write, lseek, fstat, etc.
- Uncloaked files use simple marshalling

## Cloaked Files

- Emulate read and write using mmap
- Copy data to/from memory-mapped buffers
- Decrypted automatically when read by app;
  Encrypted automatically when flushed to disk by kernel
- Shim caches mapped file regions (1MB chunks)
- Prepend file header containing size, offset, etc.

# Protection Metadata: Overview

## Per-Page Metadata

- Required to enforce privacy, integrity, ordering, freshness
- IV – randomly-generated initialization vector
- H – secure integrity hash

## Tracked by VMM

- Metadata for pages mapped into application address space
- Intuitively, what's "supposed" to be in each memory page
- (ASID, GVPN) $\rightarrow$ (IV, H)

# Protection Metadata: Details

## Protected Resource

- Need indirection to support sharing and persistence
- (RID, RPN) – unique resource identifer, page offset
- Ordered set of (IV, H) pairs in VMM “metadata cache”

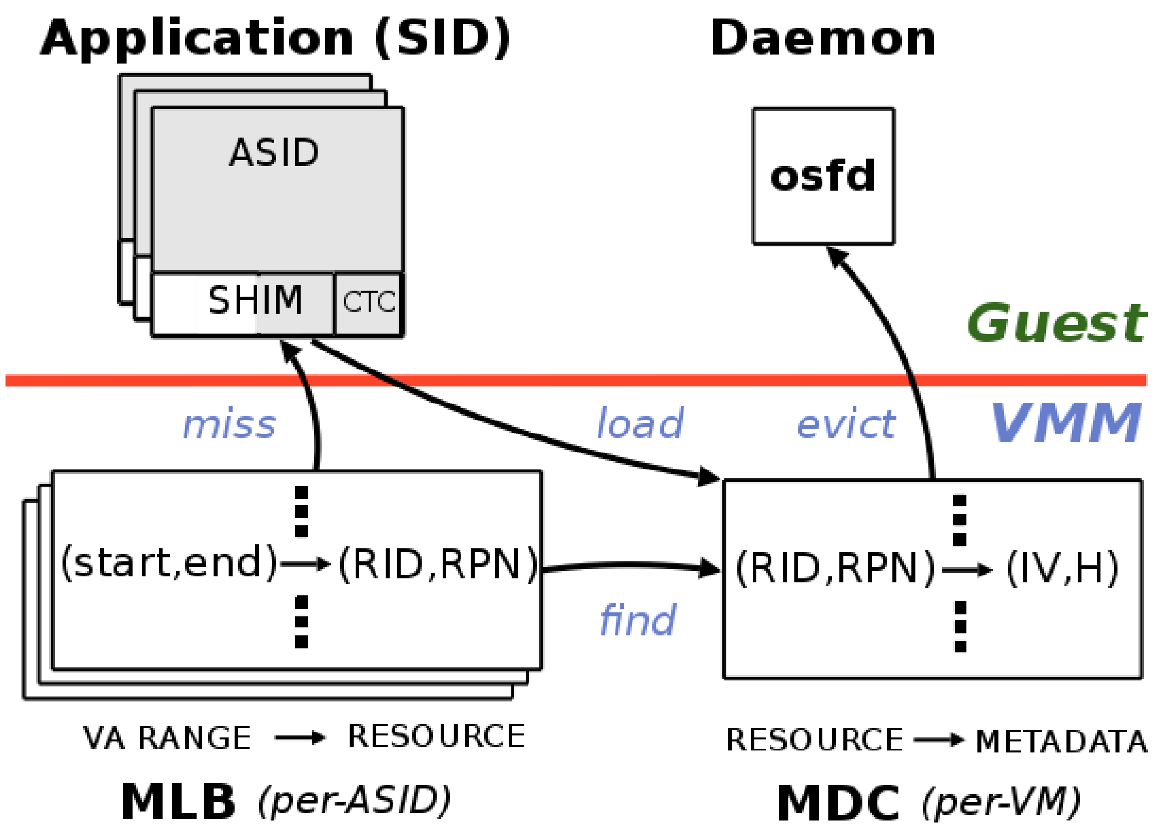
## Protected Address Space

- Shim tracks mappings (start, end) → (RID, RPN)
- VMM caches in “metadata lookaside buffer”
- VMM upcalls into shim on MLB miss

## Metadata Lookup

- (ASID, VPN) → (RID, RPN) → (IV, H)
- Persistent metadata stored securely in guest filesystem

# Managing Protection Metadata








vmware

# Q: Can OS Modify or Inject Application Code?

**Answer: No.**

- Application code resides in cloaked memory;
  it's encrypted and integrity-protected.
- Any modifications will be detected by integrity checks;
  modified page contents won't match hash in MDC.

# Q: Can OS Modify Application Instruction Pointer?

**Answer: No.**

- Application registers, including the instruction pointer (IP), are saved in the cloaked thread context (CTC) after all faults/interrupts/syscalls, and restored when cloaked execution resumes.
- The CTC resides in cloaked memory; it's encrypted and integrity-protected, so the OS can't read or modify it.

# Q: Can OS Tamper with Loader?

**Answer: No.**

- Before entering cloaked execution, the VMM can verify that the shim was loaded properly by comparing hashes of the appropriate memory pages with their expected values.
- If this integrity check fails, it will prevent the application from accessing any cloaked resources (files or memory), except in encrypted form.
- So while the OS could execute an arbitrary program instead, it would be unable to access any protected data.

# Q: Can OS Pretend to Be Application and Issue "Resume Cloaked Exec" Hypercall?

**Answer: Yes, but it can't execute malicious code.**

- When an application returns from a context switch or other interrupt, the uncloaked shim makes a hypercall asking the VMM to resume cloaked execution.
- The OS could pretend to be the application, and make this same hypercall, but integrity checking will cause it to fail unless the CTC is mapped in the proper location.
- Even if the OS succeeds, the VMM will enter cloaked execution with the proper saved registers, including the IP. All application pages must be unaltered or integrity checks will fail.
- Thus, the OS can only cause cloaked execution to be resumed at the proper point in the proper application code, so it still can't execute malicious code.