Overshadow:
Retrofitting Protection in Commodity Operating Systems

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

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

Tal Garfinkel
VMware Advanced Development

Stanford Security Forum
March 17, 2008

Our Problem: Commodity Systems, Sensitive Data

Many Applications Handle Sensitive Data

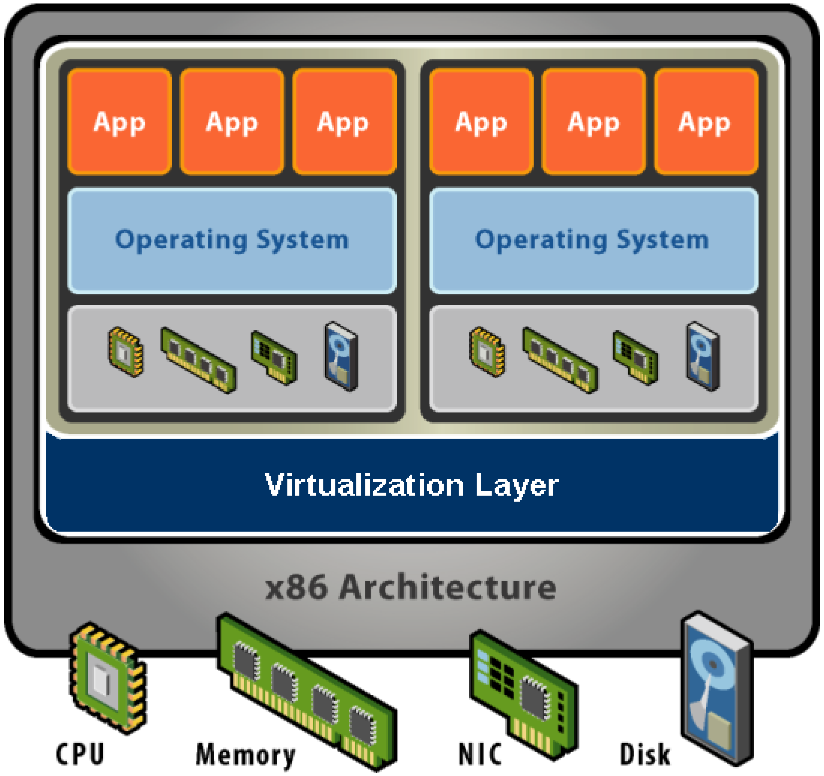
- Financial, medical, insurance, military …
- Credit cards, medical records, corporate IP …

Run on Commodity Systems

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

Why rely on all this, when we only care about our application?

Our Hammer: The Virtual Machine Monitor



Hardware-Level Abstraction

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

Where Overshadow Sits

- Interpose at the CPU/Memory Interface to add new protection mechanism

Our Goals

Protect Individual Application Data

- Privacy and integrity
- In memory and on disk

Get OS out of Trusted Computing Base

- Only have to trusted application code
- Last line of defense

Backwards Compatibility

- Unmodified commodity OS
- Unmodified application binary

Non-Goal: Availability

Outline

E2E Architecture

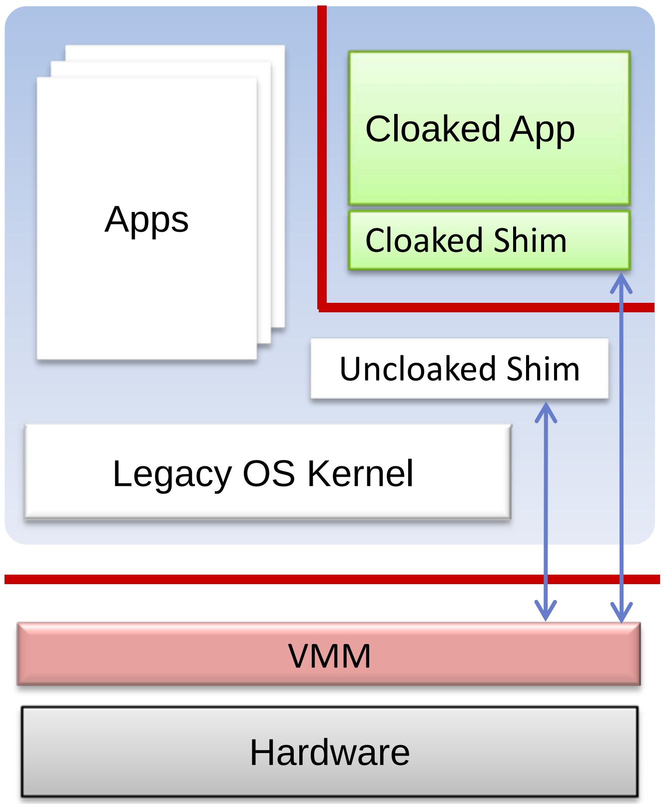
Memory Cloaking

Secure Control Transfer

Implementation

Conclusions

E2E: Big Picture



Two Virtualization Barriers

Application Data Protected

- On disk
- In memory while running

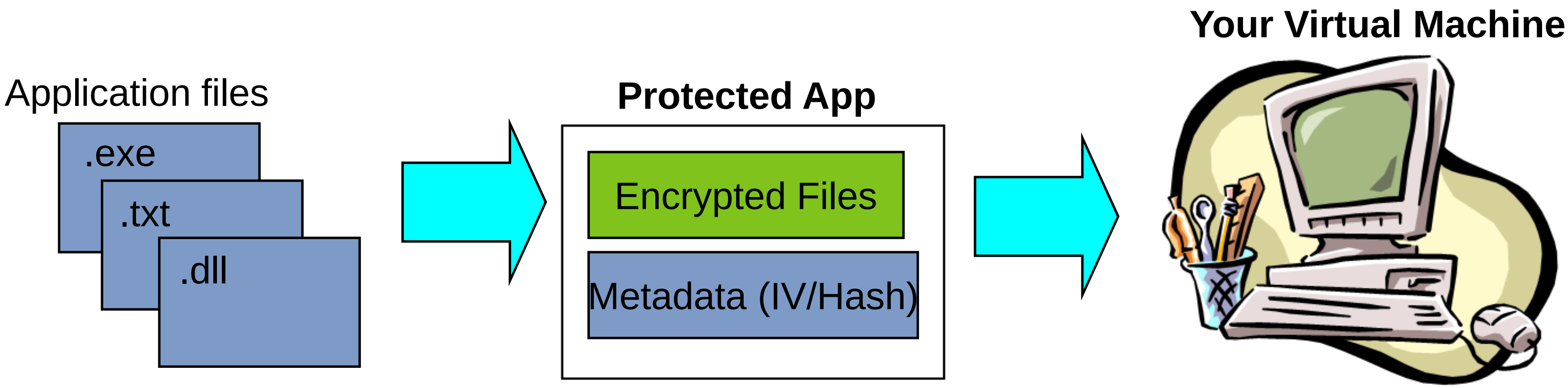
Cloaking: Two Views of Memory

- App sees normal view
- OS sees encrypted view

App/OS Interactions

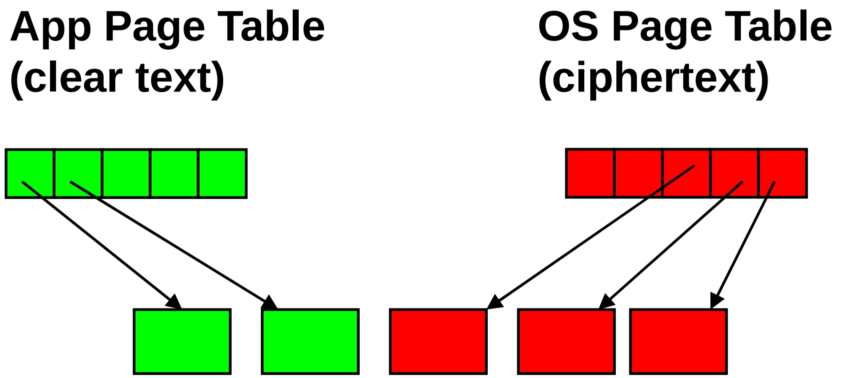
- Mediated by “shim”
- Interposes on system calls, interrupts, faults, signals
- Transparent to application

E2E: Setting Up a Protected App



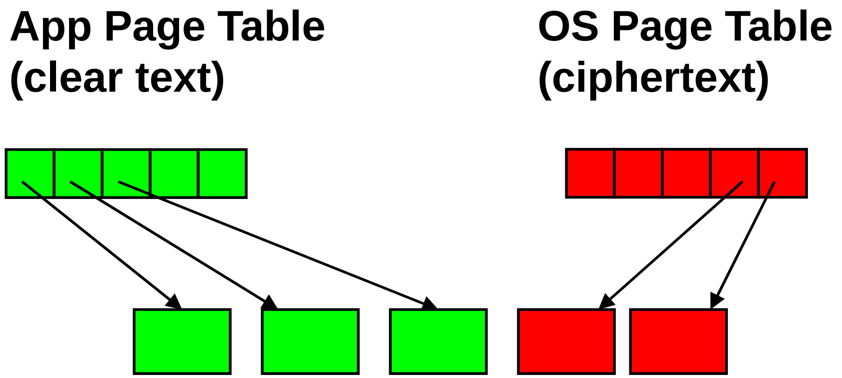
E2E: Running a Protected App

1. Trusted Loader is invoked run (checked by VMM) to start app
2. Loader memory maps app code
3. Application code/data is encrypted/decrypted on demand.
4. VMM Provides context dependant view of process memory.



E2E: Running a Protected App

1. **Trusted Loader is invoked run (checked by VMM) to start app**
2. **Loader memory maps app code**
3. **Application code/data is encrypted/decrypted on demand.**
4. **VMM Provides context dependant view of process memory.**



E2E: Protecting Application Resources

Basic Strategy

- Protect existing memory-mapped objects
  e.g. stack, heap, mapped files, shared mmaps
- Make everything else look like a memory mapped object
  e.g. open() becomes mmap(), read()/write() becomes memcpy()
- VMM Provides Memory Isolation

OS Still Manages (Encrypted) Application Resources

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

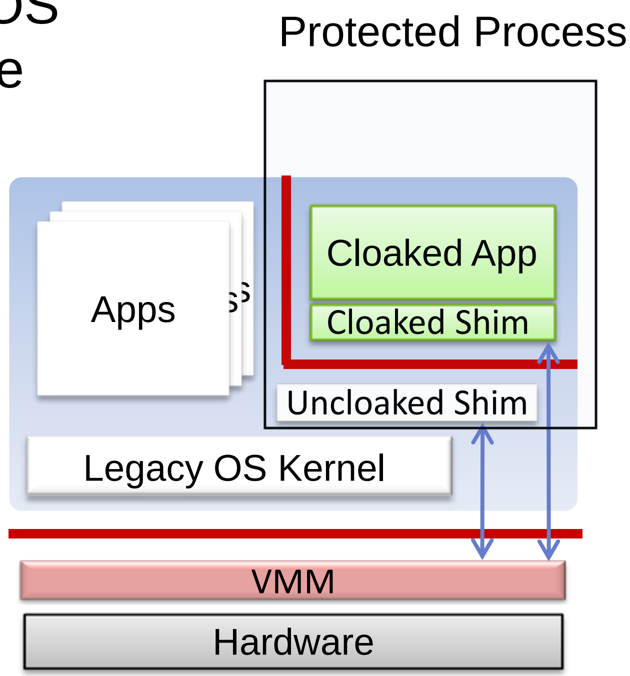
E2E: Supporting Unmodified Applications

Problem: Doesn't look like normal ABI

Examples: Modified control transfers between OS and app, OS can't access app address space directly

Solution: Shim

- Loaded into application address space
- Communicates with VMM via hypercalls
- Interposes on system calls, signals, etc.



Outline

- E2E Architecture
- Memory Cloaking
- Secure Control Transfer
- Implementation
- Conclusions

Memory Mapping: OS

virtual → **physical**

OS page table

Memory Mapping: VMM

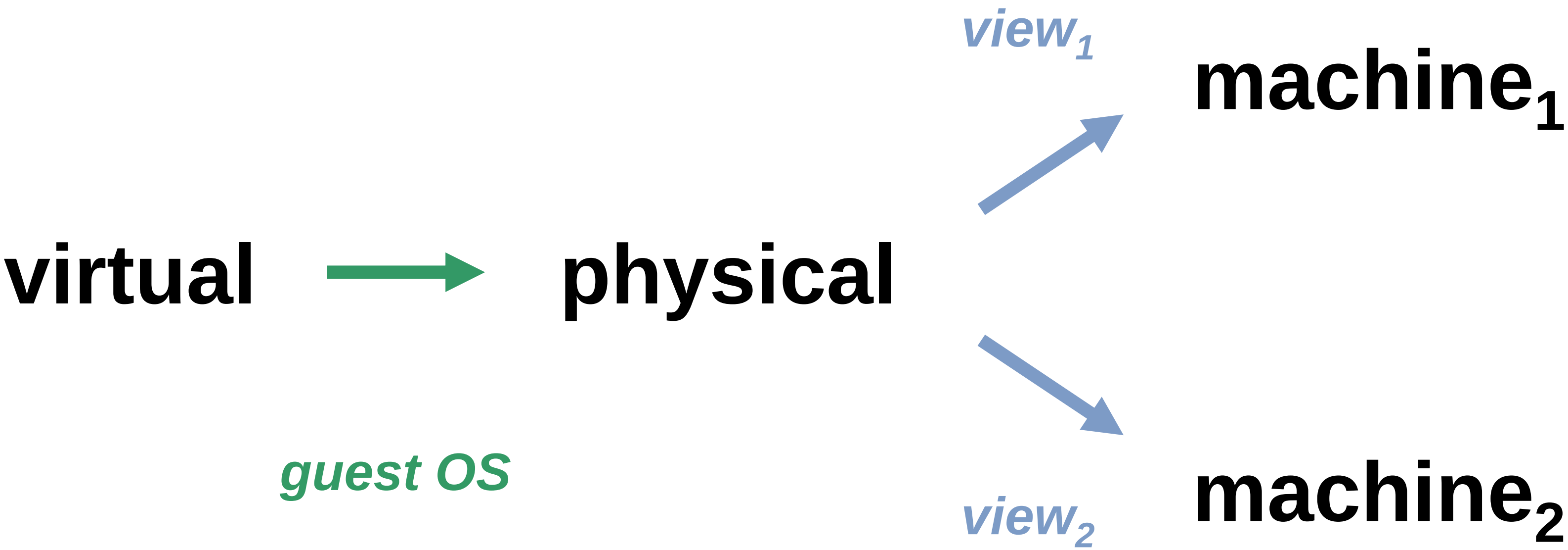
virtual → physical → machine

guest OS (virtual → physical)

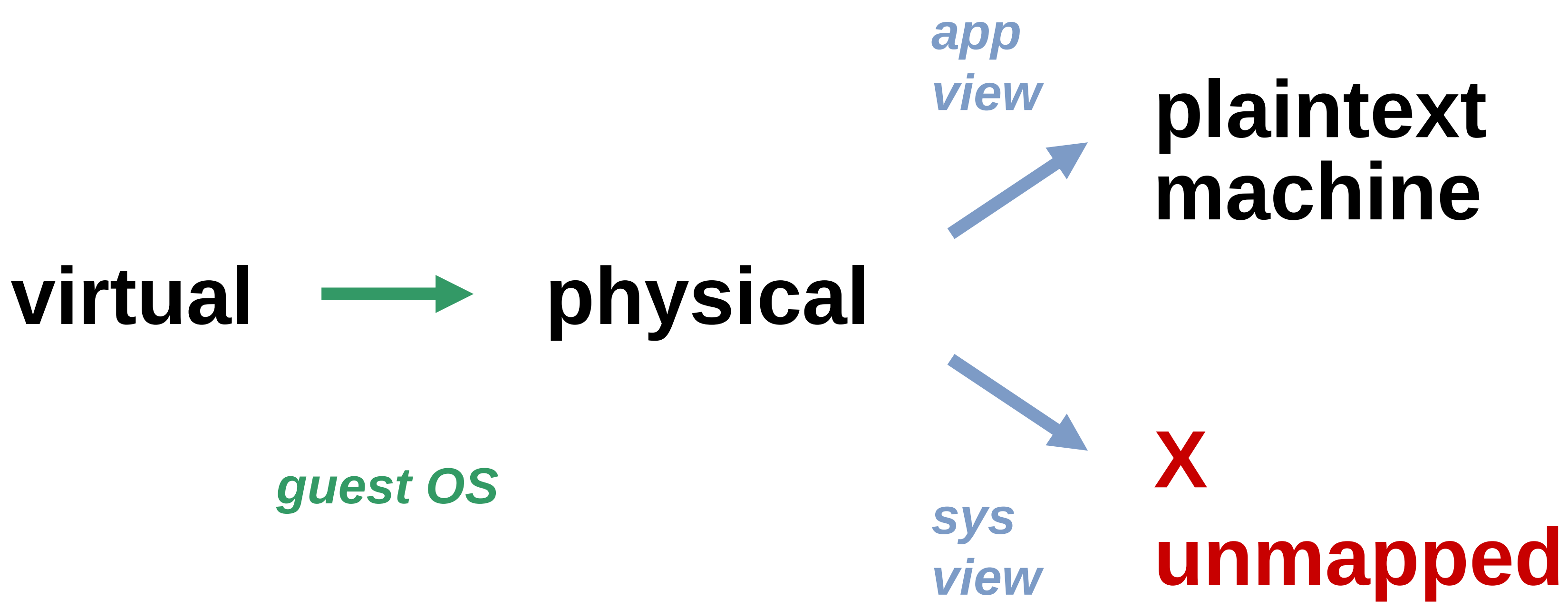
vmm (physical → machine)

vmware

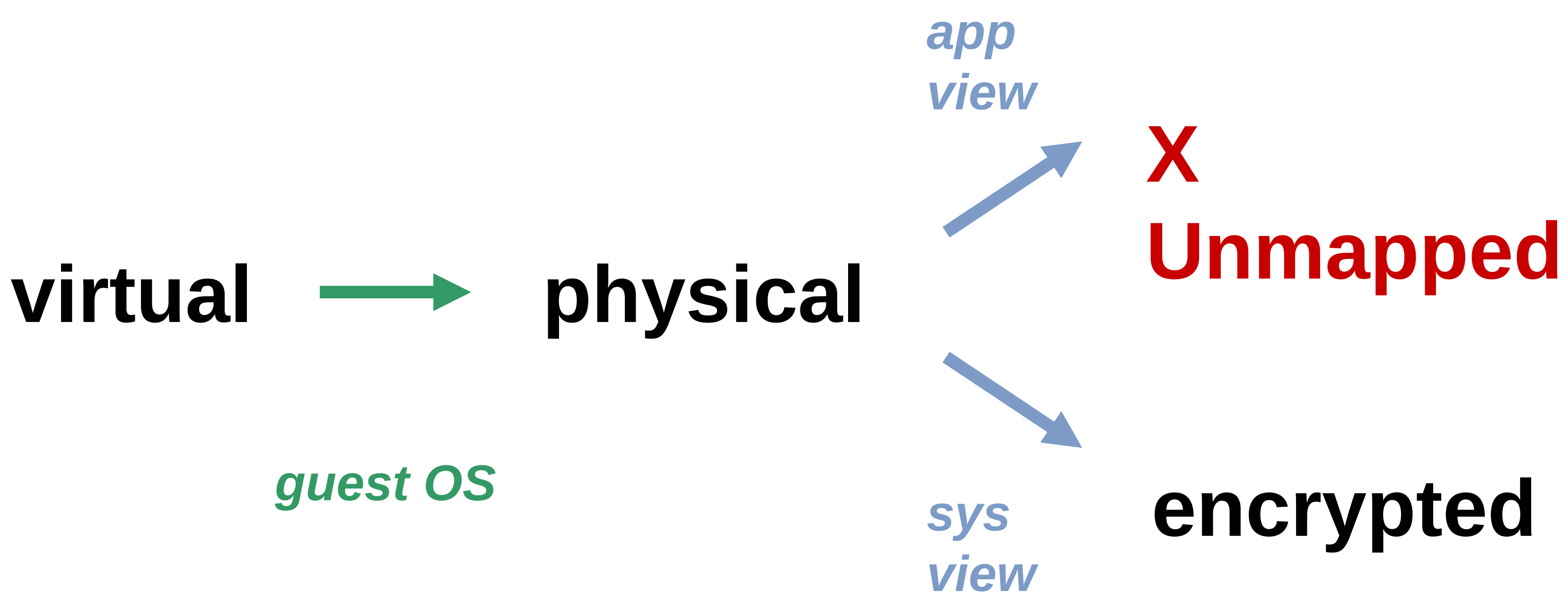
Multi-Shadowing: Context-Dependent Views



Cloaking: Multi-Shadowing + Cryptography



Cloaking: System Accesses Page



Fault into VMM: encrypt/hash contents, remap

Cloaking: Application Accesses Page

app view

plaintext machine

virtual → **physical**

guest OS

sys view

X unmapped

Fault into VMM: verify hash, decrypt, remap

Protecting Data Integrity

Challenges

- Enforce integrity, ordering, freshness

VMM Manages Per-Page Metadata

- Tracks what’s “supposed to be” in each memory page
  - E.g. infer based on mmap()
- IV – randomly-generated initialization vector
- H – secure integrity hash

See paper for more…

Outline

E2E Architecture

Memory Cloaking

Secure Control Transfer

Implementation

Conclusions

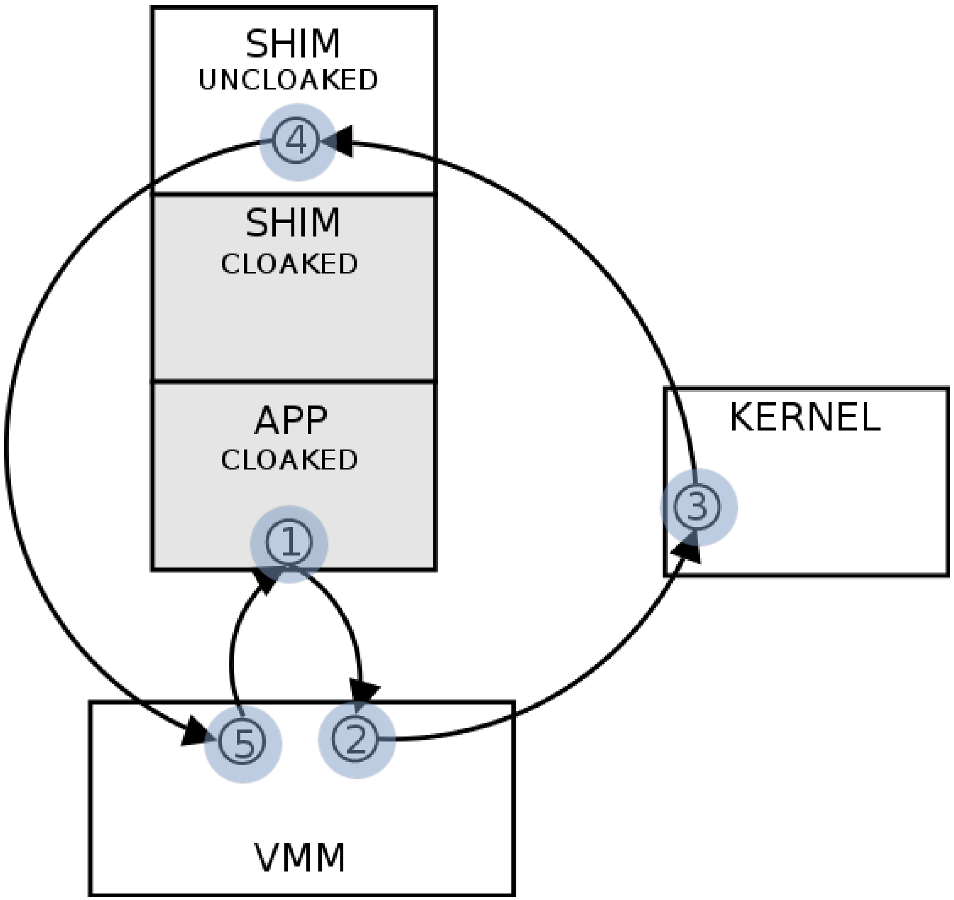
Secure Control Transfer

Problem: Can’t let OS tranfer control to arbitrary place in app (with arbitrary registers).

Solution: Enforce control transfer protocol.

- **Implicit: Faults/Premption**
- **Explicit: System Calls**

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

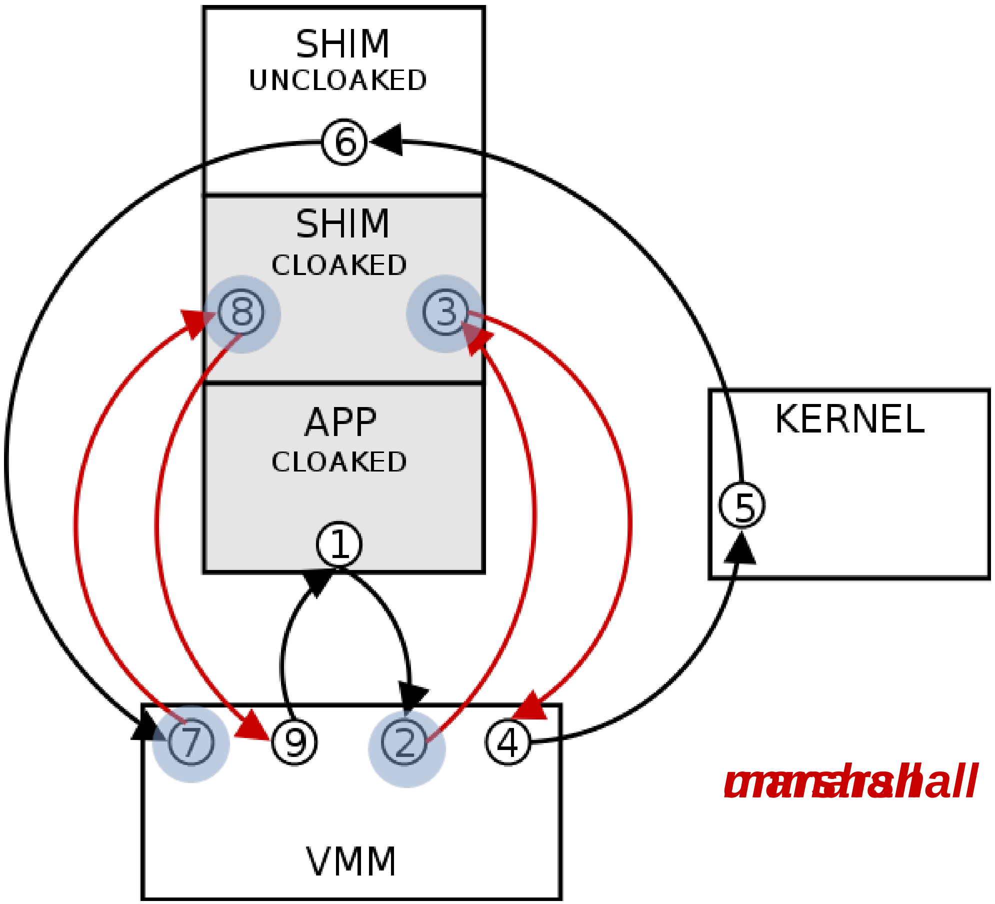
4. Shim hypercalls into VMM

- 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

Outline

E2E Architecture

Memory Cloaking

Secure Control Transfer

Implementation

Future Work

Related Work

Conclusions

Implementation

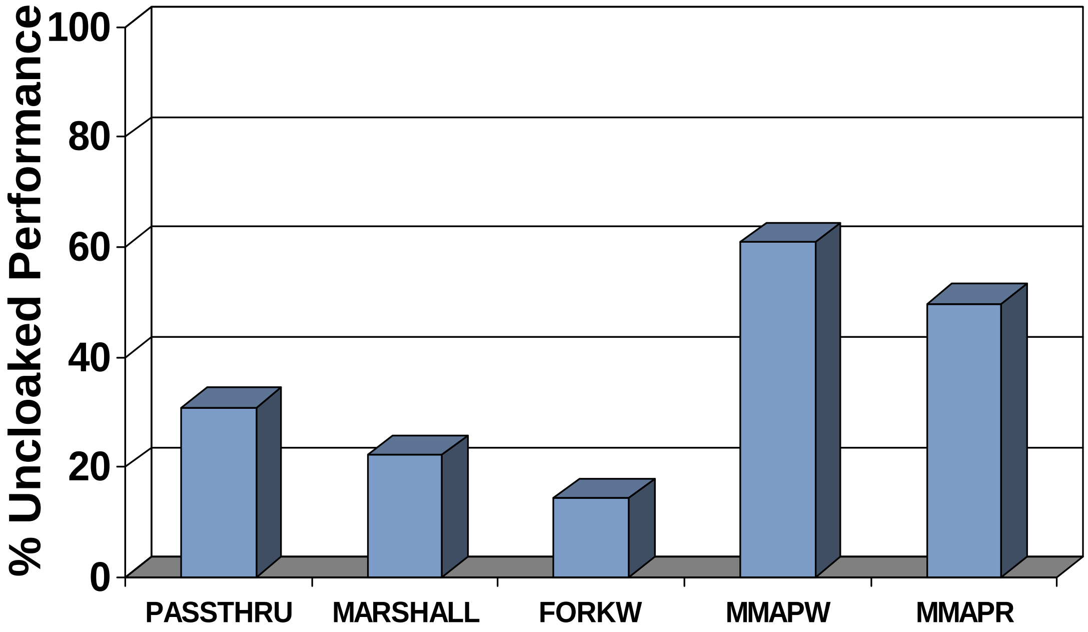
Overshadow System

- 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

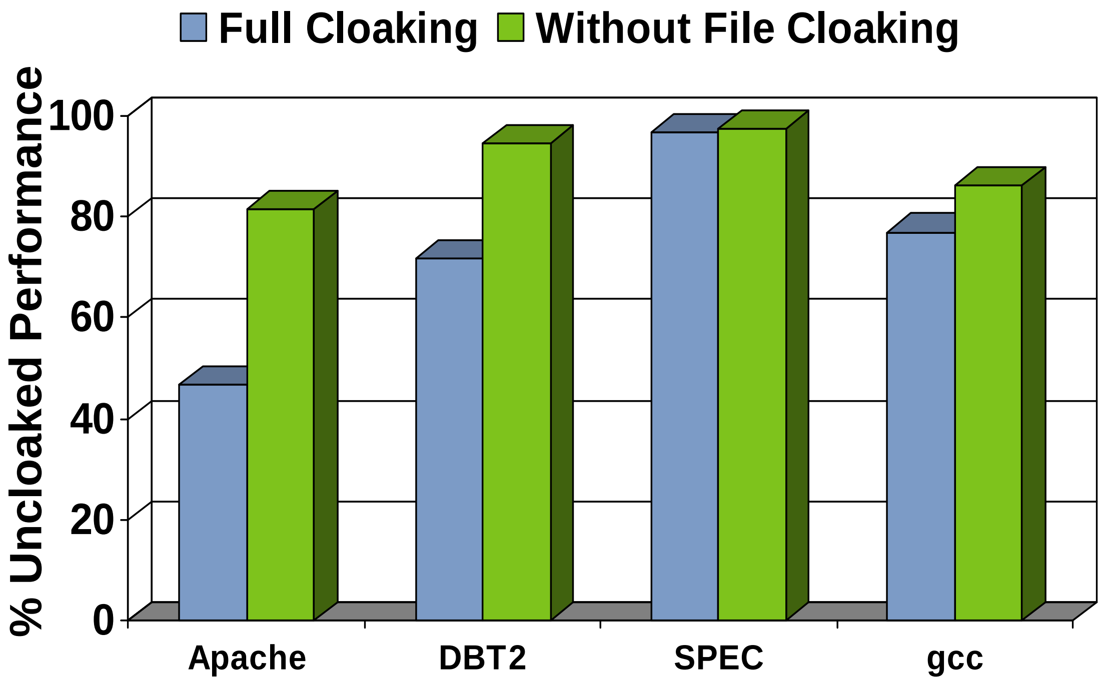
Processes

- 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

vmware

Conclusions

Promising New Approach

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

Powerful New Mechanisms

- Multi-shadow memory cloaking
- Shim allows transparent ABI modification

Future Directions

- Security implications of a malicious OS
- Additional uses of Cloaking

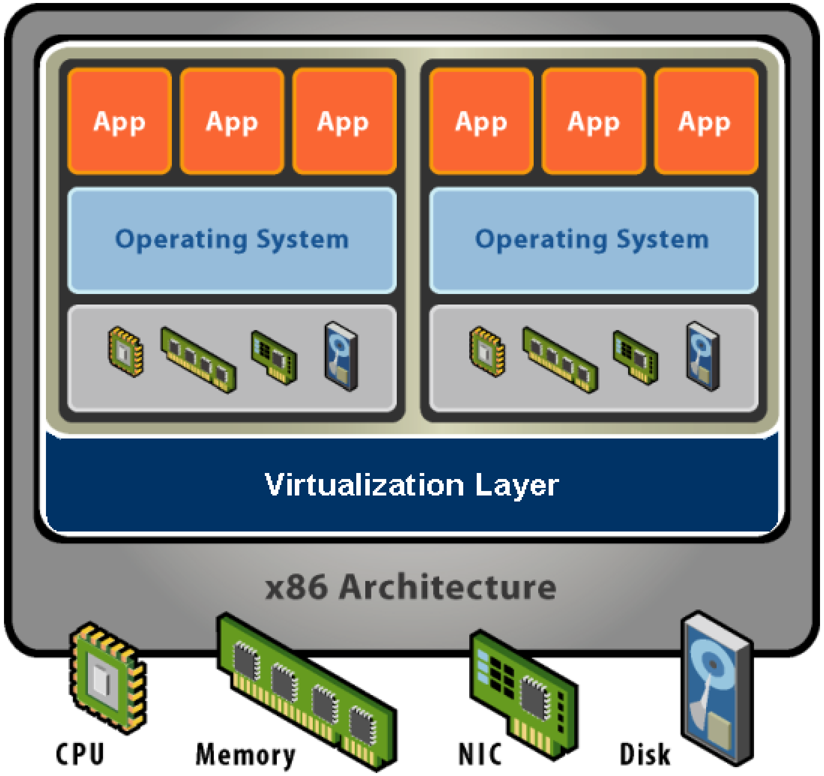
Questions?

For More Information

- Read the paper
  - See ASPLOS 08 Proceedings
  - Google: $MY_NAME
- Send feedback to mailing list
  overshadow@vmware.com

Backup Slides

What is a Virtual Machine?



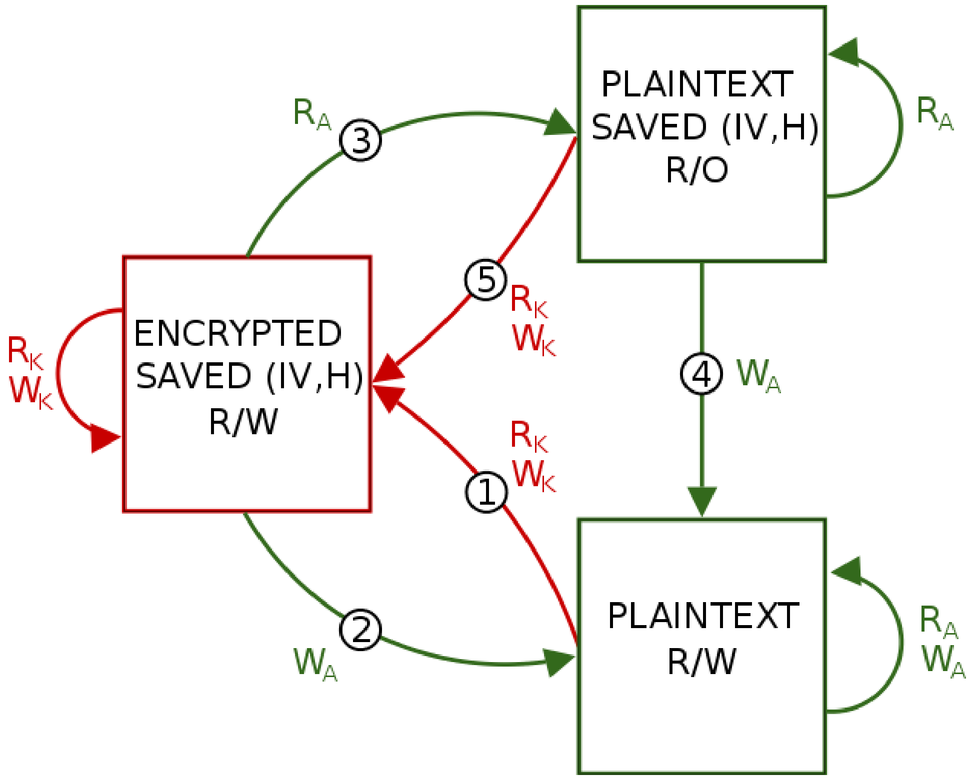
Hardware-Level Abstraction

- 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

vmware

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”

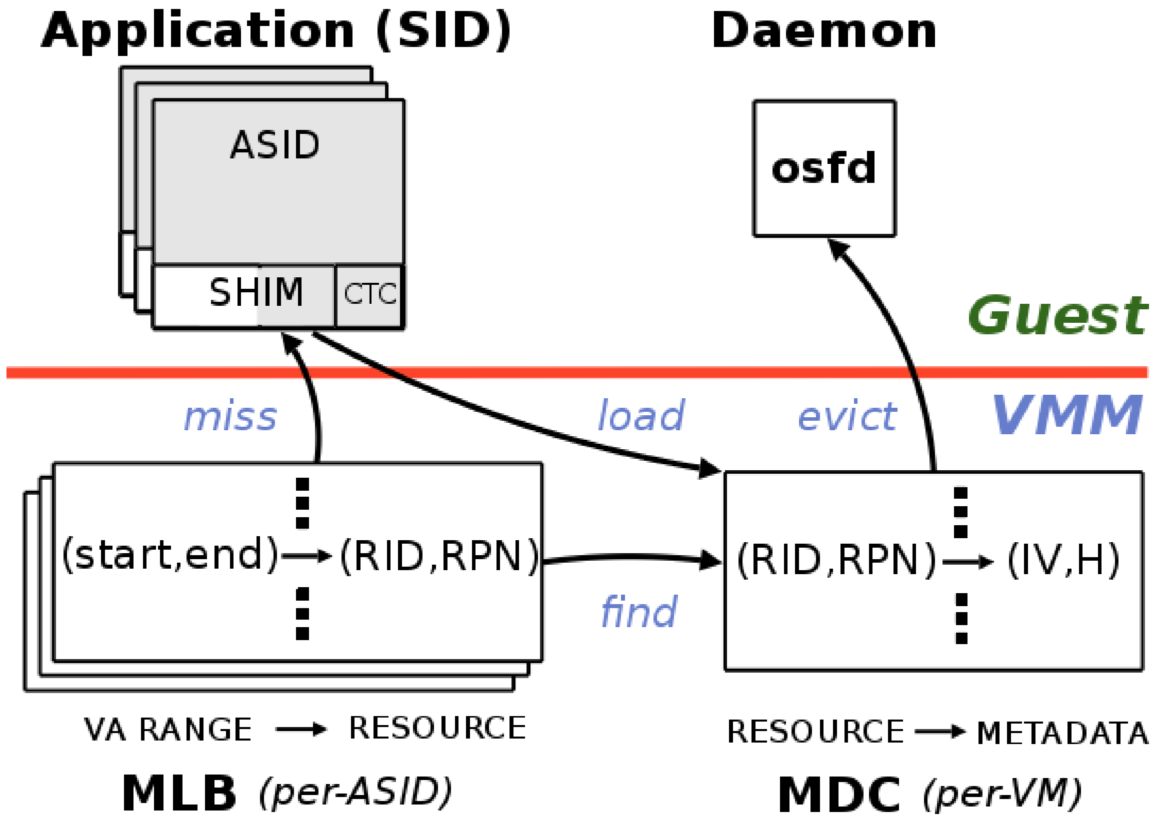
Protected Address Space

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

Metadata Lookup

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

Managing Protection Metadata



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.

More Backup Slides

Motivation: Vulnerable Systems

Many Applications Handle Sensitive Data

- Financial, medical, insurance, military …
- Credit cards, medical records, corporate IP …

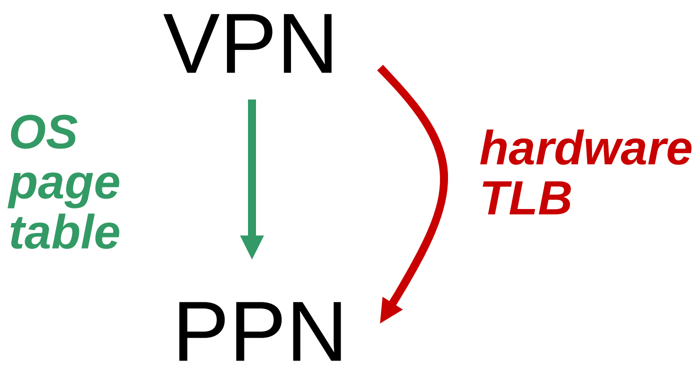
Yet Trust Commodity Systems

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

Example: Database Server

- Containing all sorts of sensitive information
- Secure, but runs on commodity OS
- Game over if attacker becomes root (*e.g.* via /dev/mem)

Review: Virtual Memory

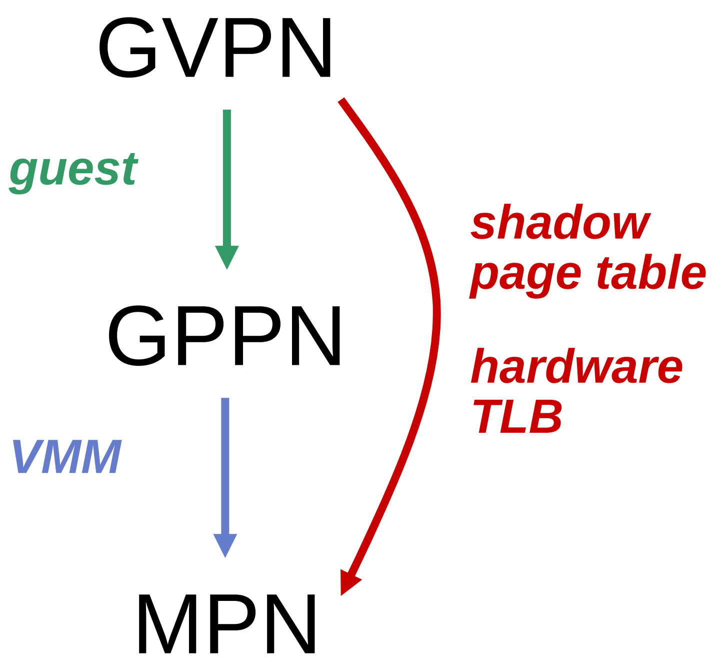


Traditional OS Approach

Level of Indirection

- Virtual → Physical
- OS page table maps
  VPN (virtual page number) to
  PPN (physical page number)
- Cached by hardware TLB

Classical Memory Virtualization



Traditional VMM Approach

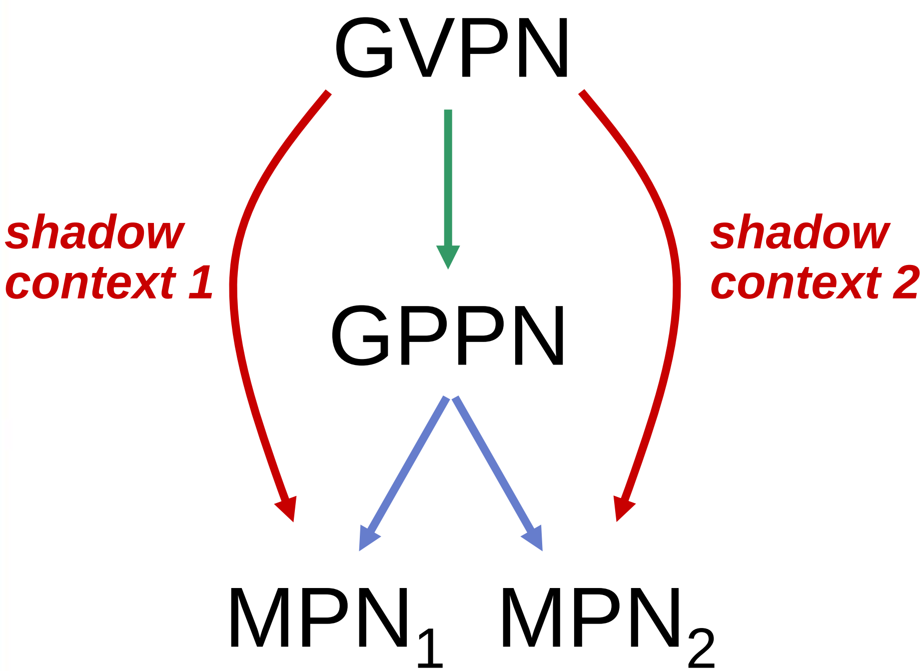
Extra Level of Indirection

- Virtual → Physical
  Guest OS page table maps
  GVPN (virtual page number) to
  GPPN (physical page number)
- Physical → Machine
  VMM maps GPPN to MPN

Shadow Page Table

- Composite of two mappings
- Directly maps GVPN to MPN
- Cached by hardware TLB

Multi-Shadowing Primitive



New Way to Leverage VMM

Multiple Views of Memory

- GPPN maps to multiple MPNs
- Using multiple shadow page tables
- View depends on "context" accessing page

General Mechanism

- Orthogonal to protection domains defined by OS and processor
- Enables new protection schemes

Cloaking: Multi-Shadowing + Cryptography

Single Page, Dual Views

- GPPN maps to single MPN
- Encrypt/decrypt MPN contents dynamically
- Hash encrypted contents to protect integrity

Access to Cloaked Page

- *By kernel*: encrypt, generate hash, update shadow mappings
- *By app*: verify integrity hash, decrypt, update shadow mappings

Responsibilities

- OS manages application resources (without seeing contents)
- VMM manages protection (including metadata and keys)

Cloaking OS Resources

Page-Oriented Protection

- Using low-level cloaking primitive
- Building block for higher-level OS abstractions

Memory-Mapped Objects in Modern OS

- Private process memory: stack, heap …
- File-backed memory: code regions, mmaps …
- Shared memory: fork, shared mmaps …

Basic Strategy

- Protect existing memory-mapped objects
- Make everything else look like one

Shim: Supporting Unmodified Applications

What's a Shim?

- OS-specific user-level program
- Linked into application address space
- Separate cloaked and uncloaked regions
- Communicates with VMM via hypercalls

Functionality

- Extends reach of VMM to applications
- Interposes on privilege-mode transitions
- Secure context identification and control transfer
- Tracks application resources
- Adapts system calls

Protection Metadata

Protected Resource

- Ordered set of pages
- Portions mapped into application address space
- May be persistent or transient

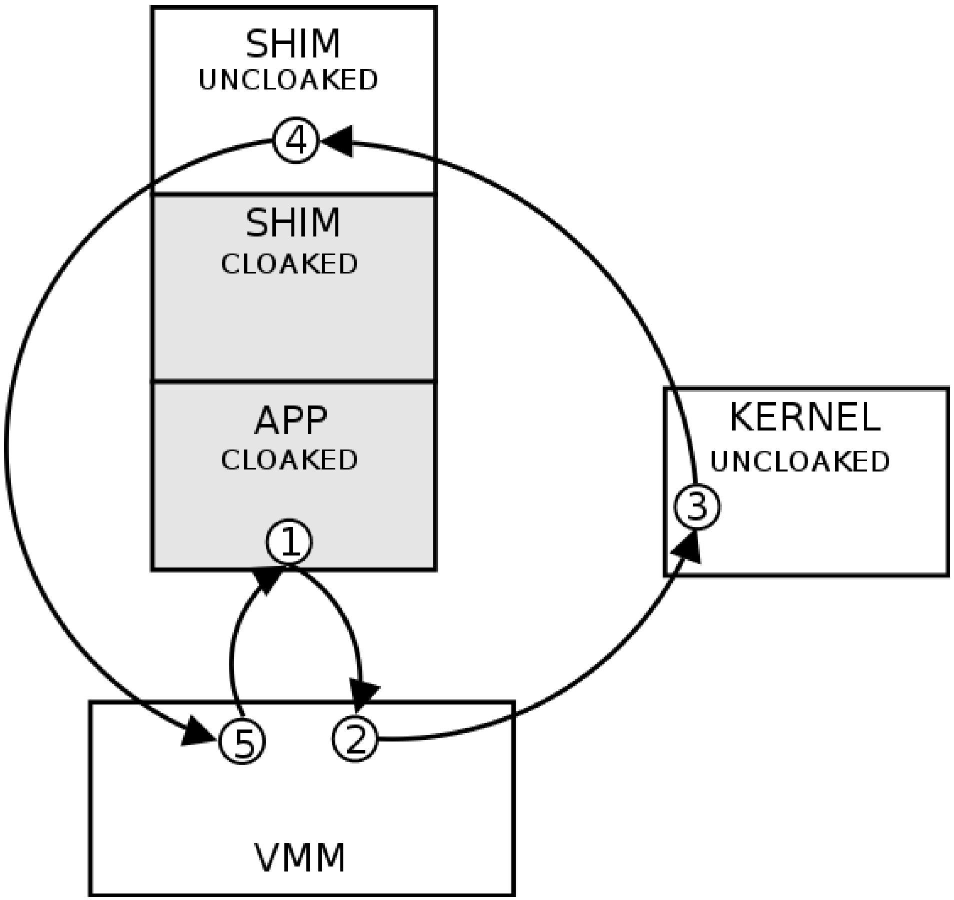
Per-Page Metadata

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

Managed by VMM

- Tracks what's "supposed to be" in each memory page
- Shim helps VMM map GVPN $\rightarrow$ (IV, H)

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**
   - Self-identifying hypercall to resume cloaked execution
5. **VMM returns to app**
   - Restores regs
   - Transfers control to app