HEISENBYTE:

Thwarting Memory Disclosure Attacks using Destructive Code Reads

Adrian Tang

Simha Sethumadhavan

Salvatore Stolfo

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

CCS 2015 - Oct 12-16, Denver, US

Key Idea: Destructive Code Reads

Problem

Executable memory can be read

↓

Memory disclosure bugs

↓

Dynamic code reuse attacks

Our Solution

Make executable memory indeterminate after it has been read

Our Inspiration

Werner Heisenberg, in 1933
(German theoretical physicist)

Image credits: Wikipedia

Observer Effect:

"The act of observing a system inevitably changes the state of the system."

HEISENBYTE's destructive code reads:

"Reading executable memory changes the executable state of the read memory."

Executing memory after reading it yields unpredictable behavior

HEISENBYTE in a Slide

Dynamic Code Reuse Attack

Memory disclosure
+
1 Scan memory at runtime for gadgets
+
2 Chain gadgets to generate shellcode
+
3 Redirect control flow

Prior Defenses

Memory disclosure

XnR (CCS'14)
HideM (CODASPY'15)
Readactor (Oakland'15)

Execute-only Mem

2 Chain gadgets to generate shellcode
+
3 Redirect control flow

Our Work

Memory disclosure
+
1 Scan memory at runtime for gadgets
+
2 Chain gadgets to generate shellcode
+

HEISENBYTE
(This talk)

Destructive CRs

Tolerates discovery of code reuse gadgets, *but* prevents them from being used as intended

Extends the benefits of execute-only memory to closed source COTS binaries, especially on Windows

Why defend at Step ❸?

Extends the benefits of execute-only memory to closed source COTS binaries, especially on Windows

1) Addresses the problem of incomplete separation of data from code in (Windows) COTS binaries

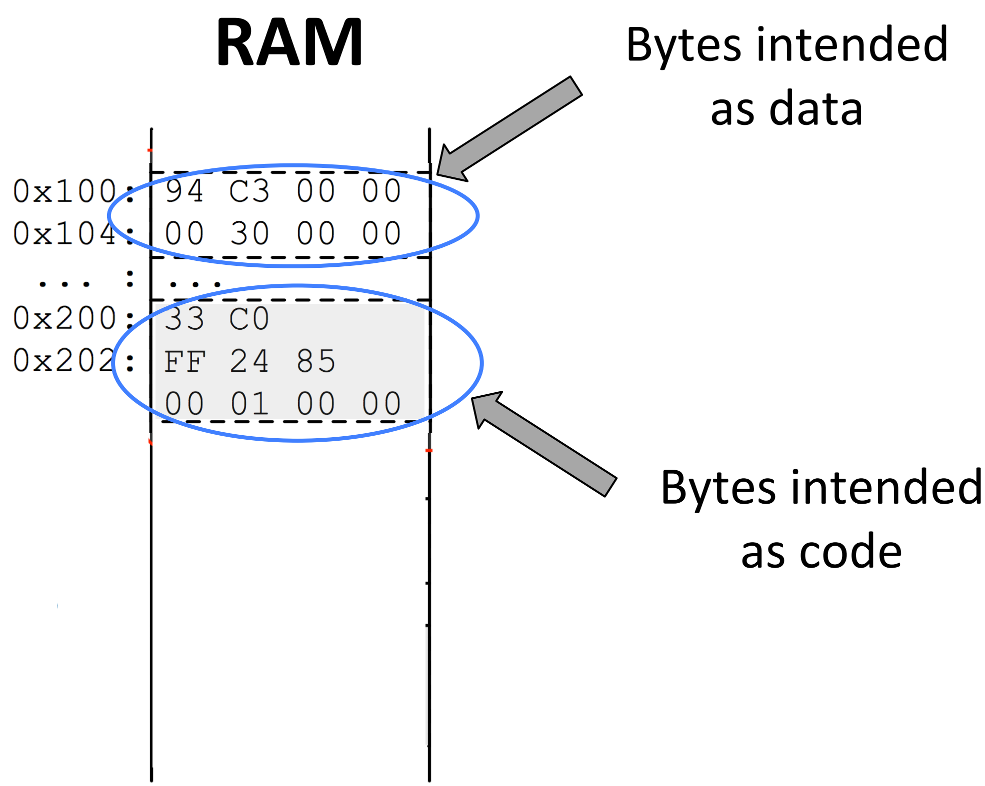
2) Protects transparently legacy programs that mix data and code in executable JIT dynamic code

Outline

- Destructive Code Reads
- System Implementation
- Evaluation
- Future Work

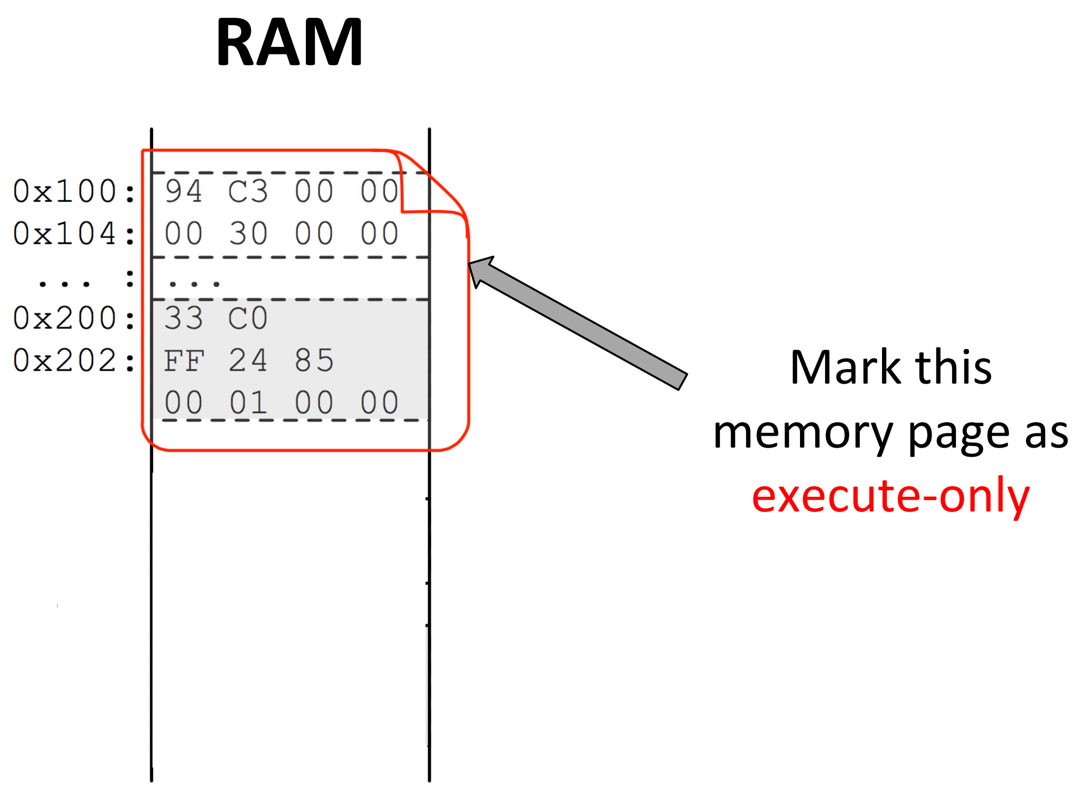
Destructive Code Reads

Detecting read operations into executable memory



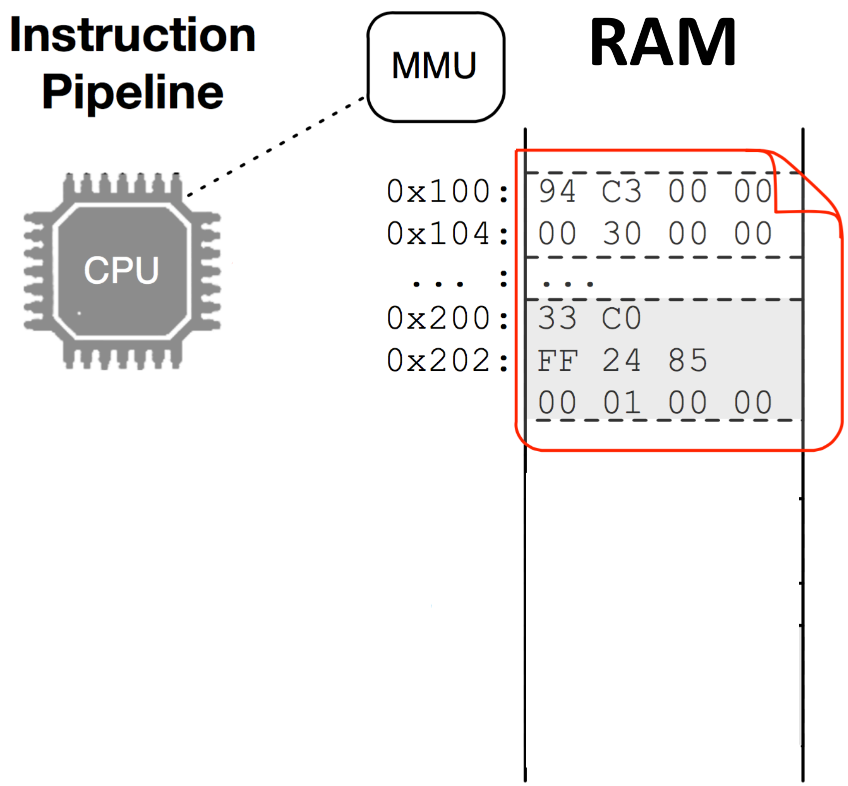
Destructive Code Reads

Detecting read operations into executable memory



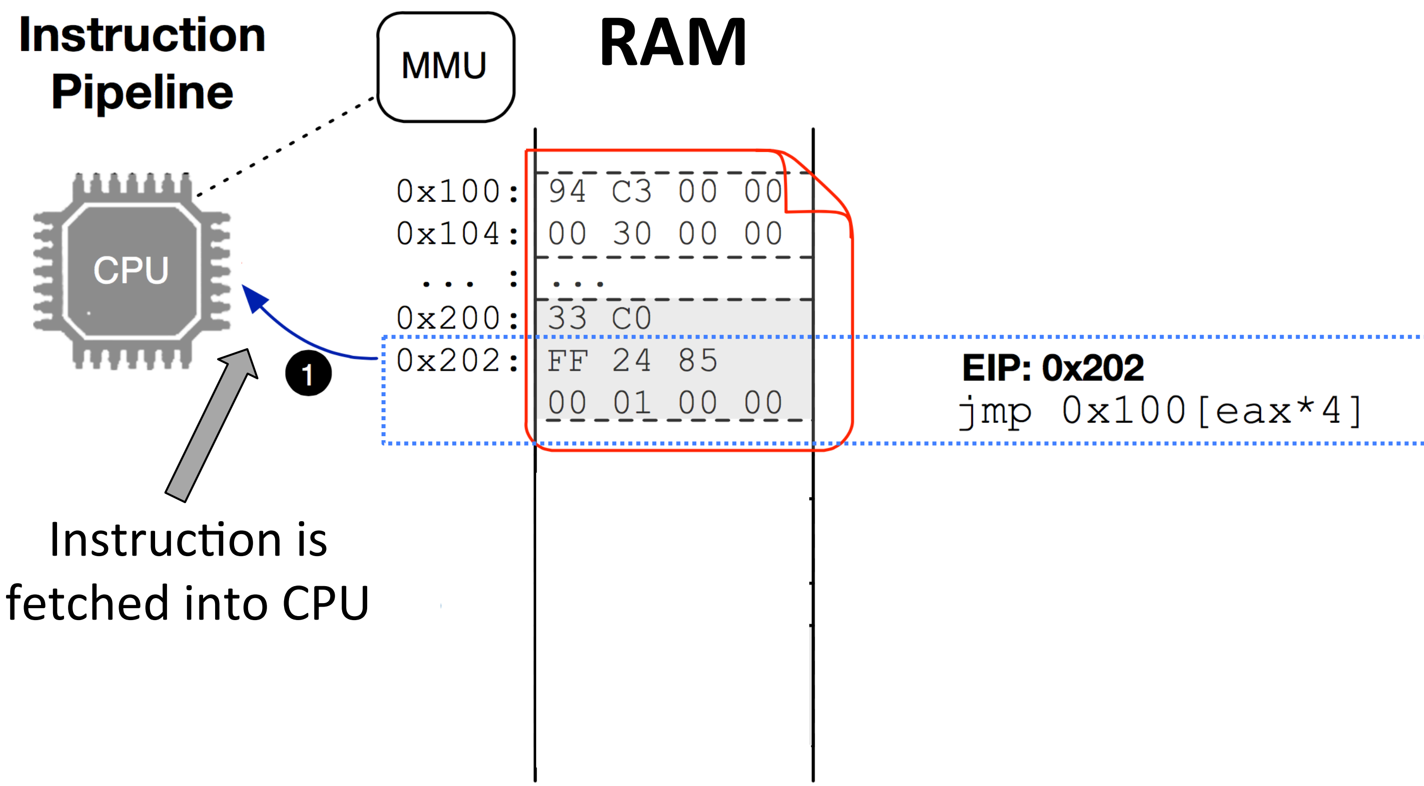
Destructive Code Reads

Detecting read operations into executable memory



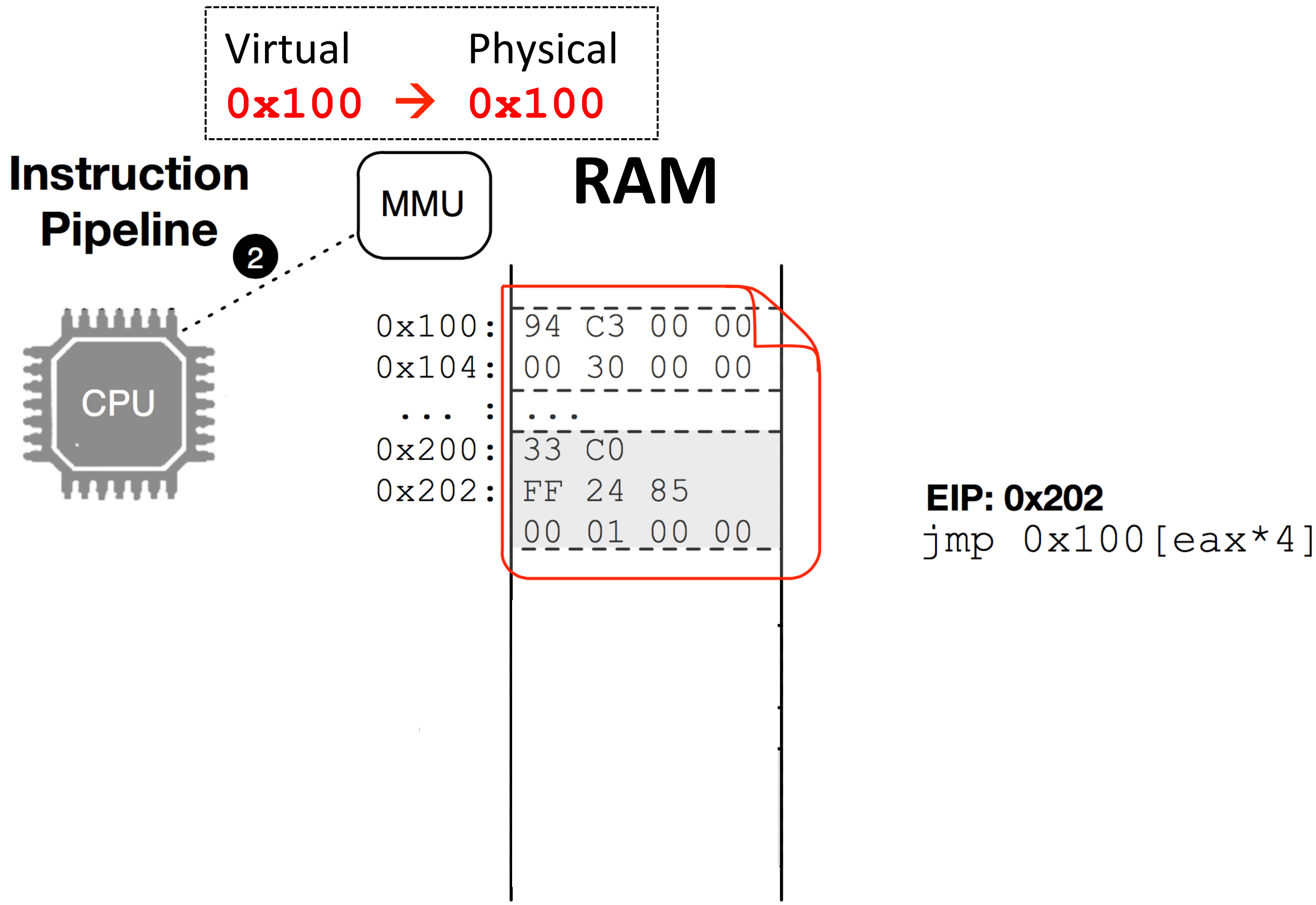
Destructive Code Reads

Detecting read operations into executable memory



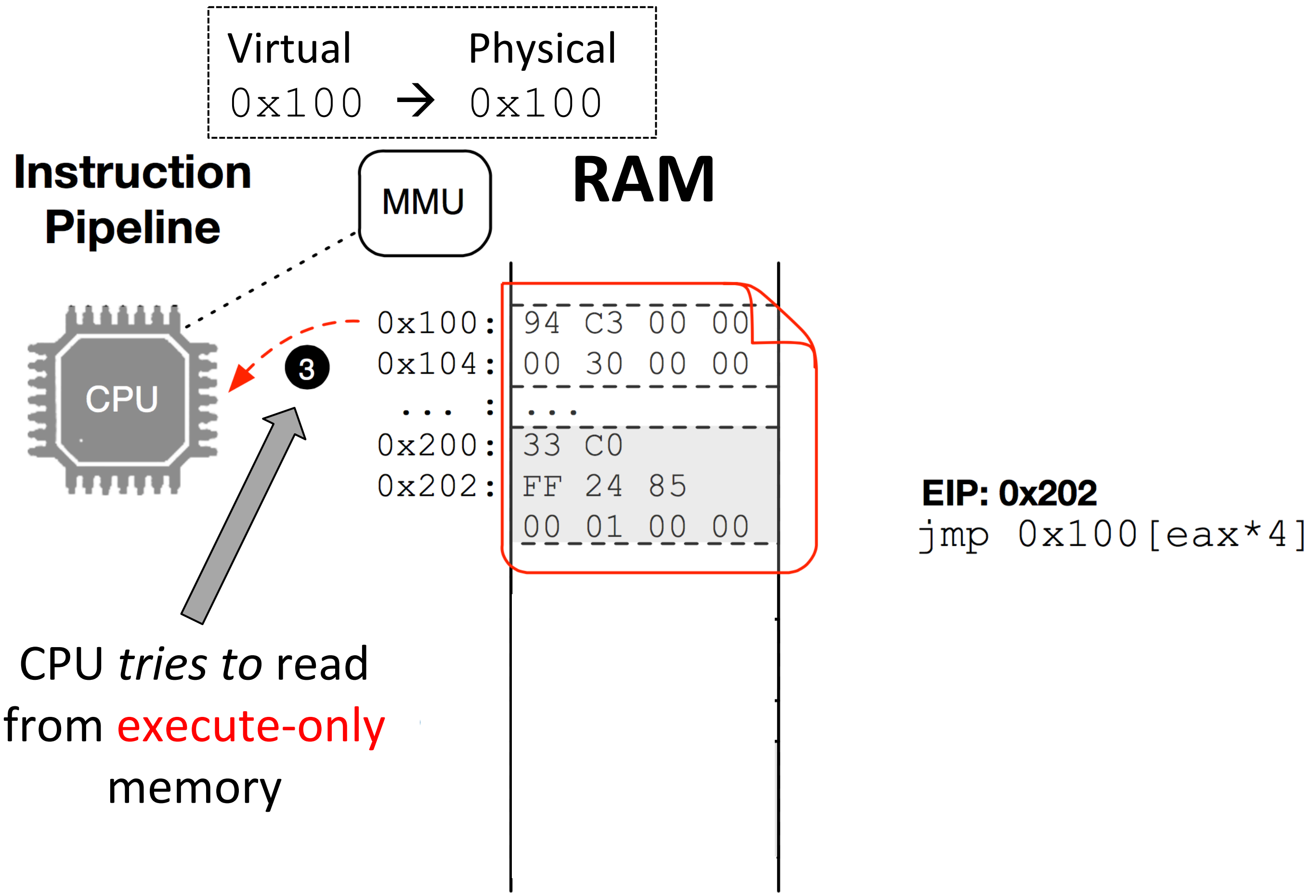
Destructive Code Reads

Detecting read operations into executable memory



Destructive Code Reads

Detecting read operations into executable memory

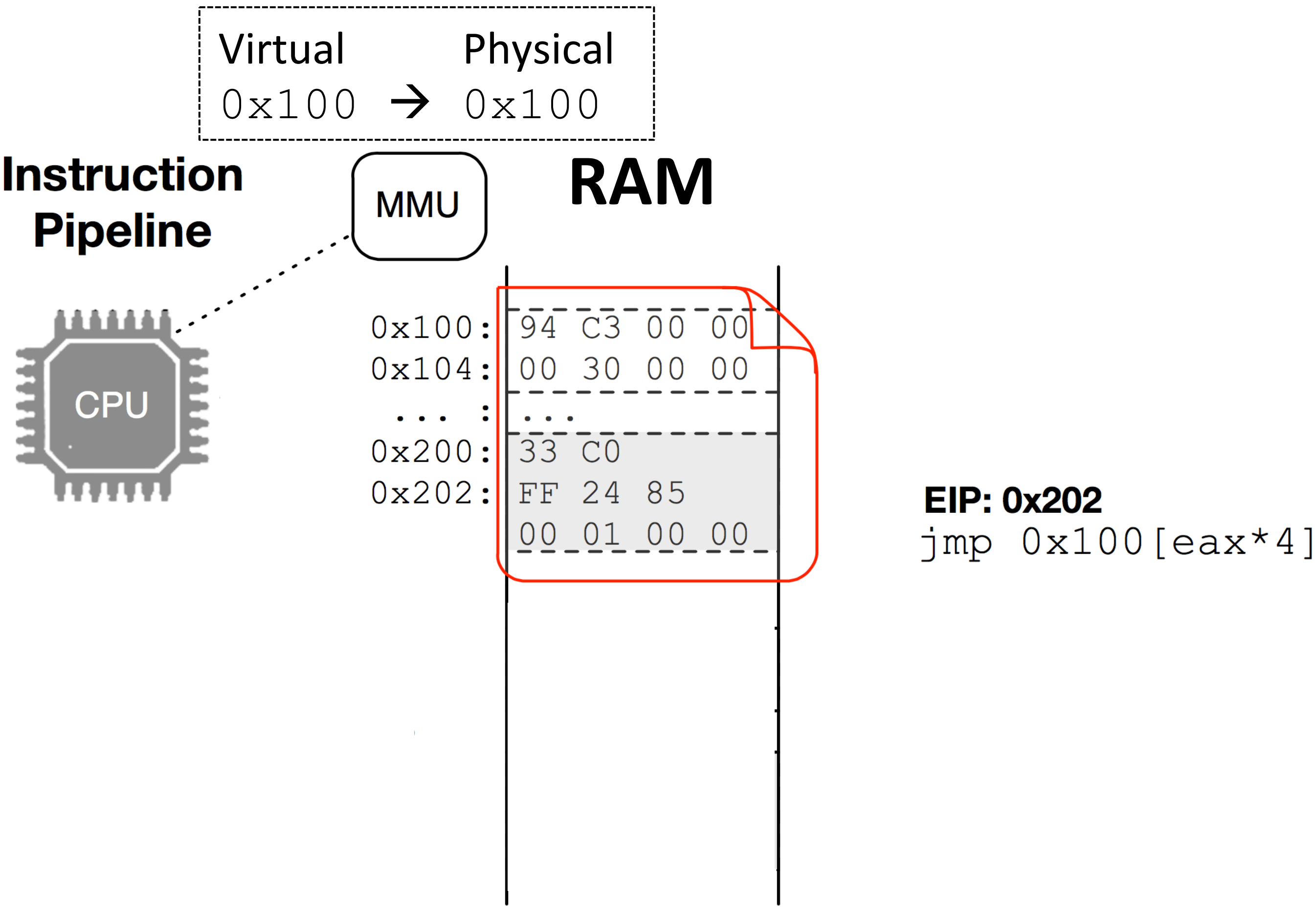


CPU *tries to* read from execute-only memory

Read into executable memory detected

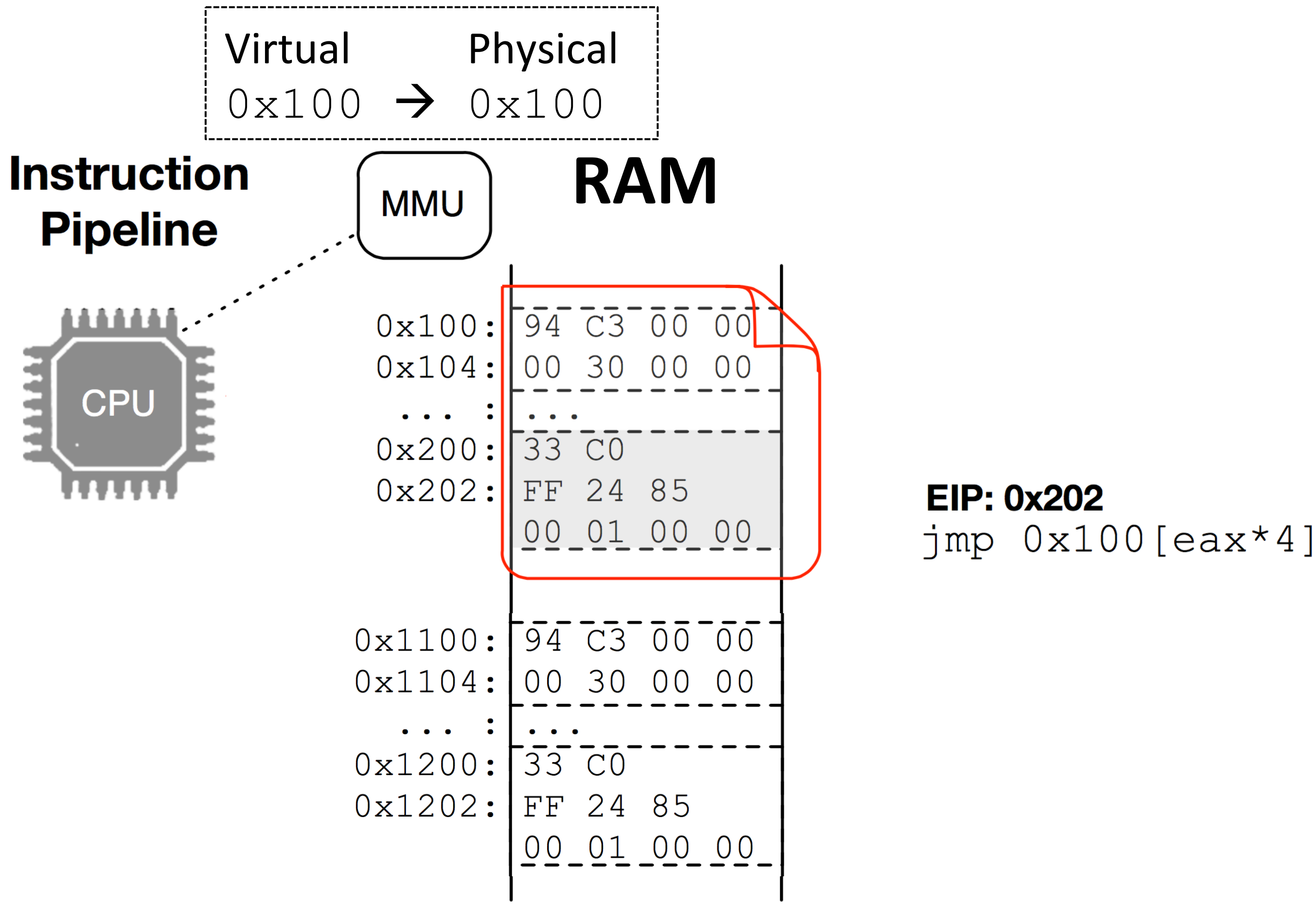
Destructive Code Reads

"Destroying" the executable byte that is read



Destructive Code Reads

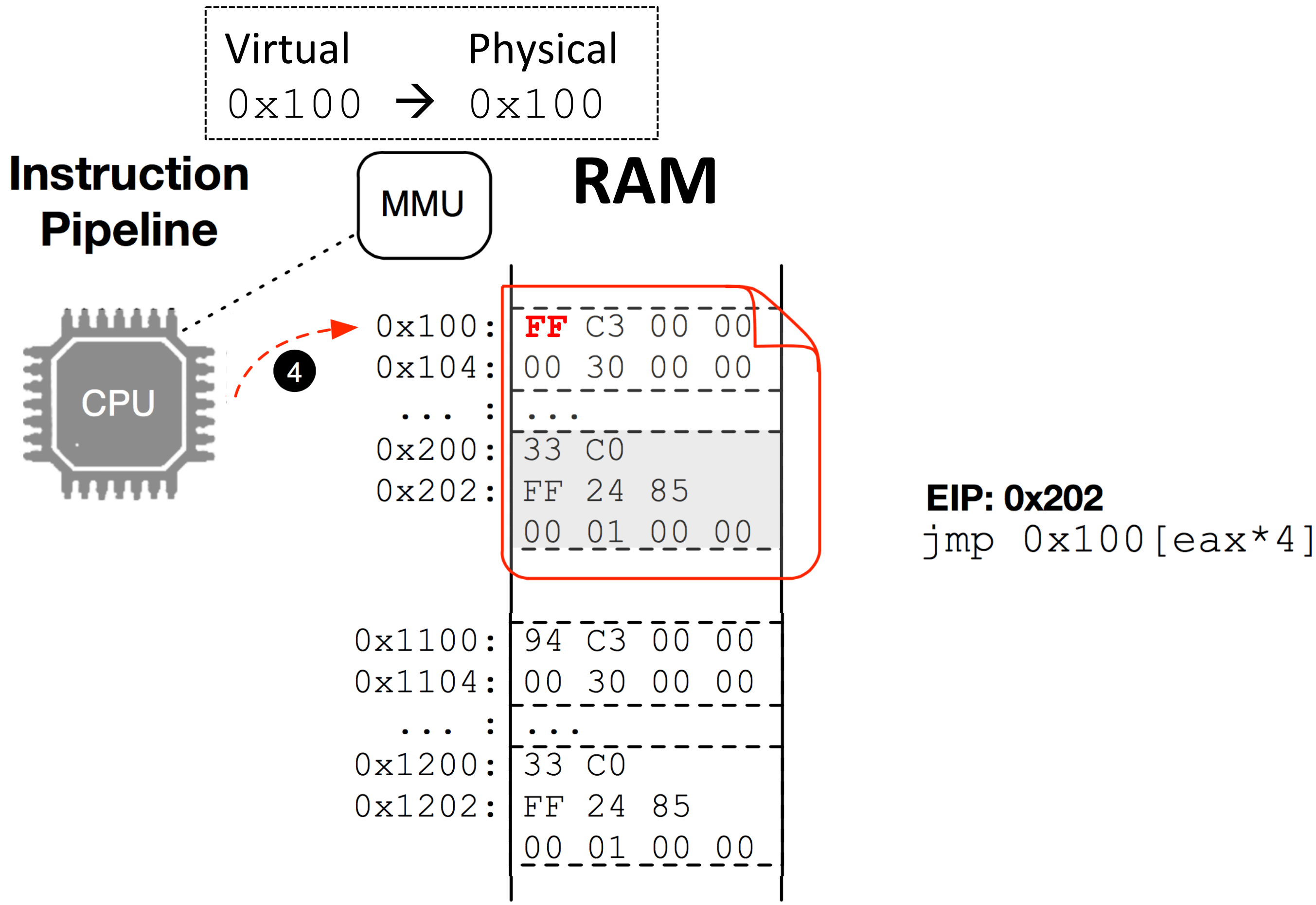
"Destroying" the executable byte that is read



Duplicate original executable memory page

Destructive Code Reads

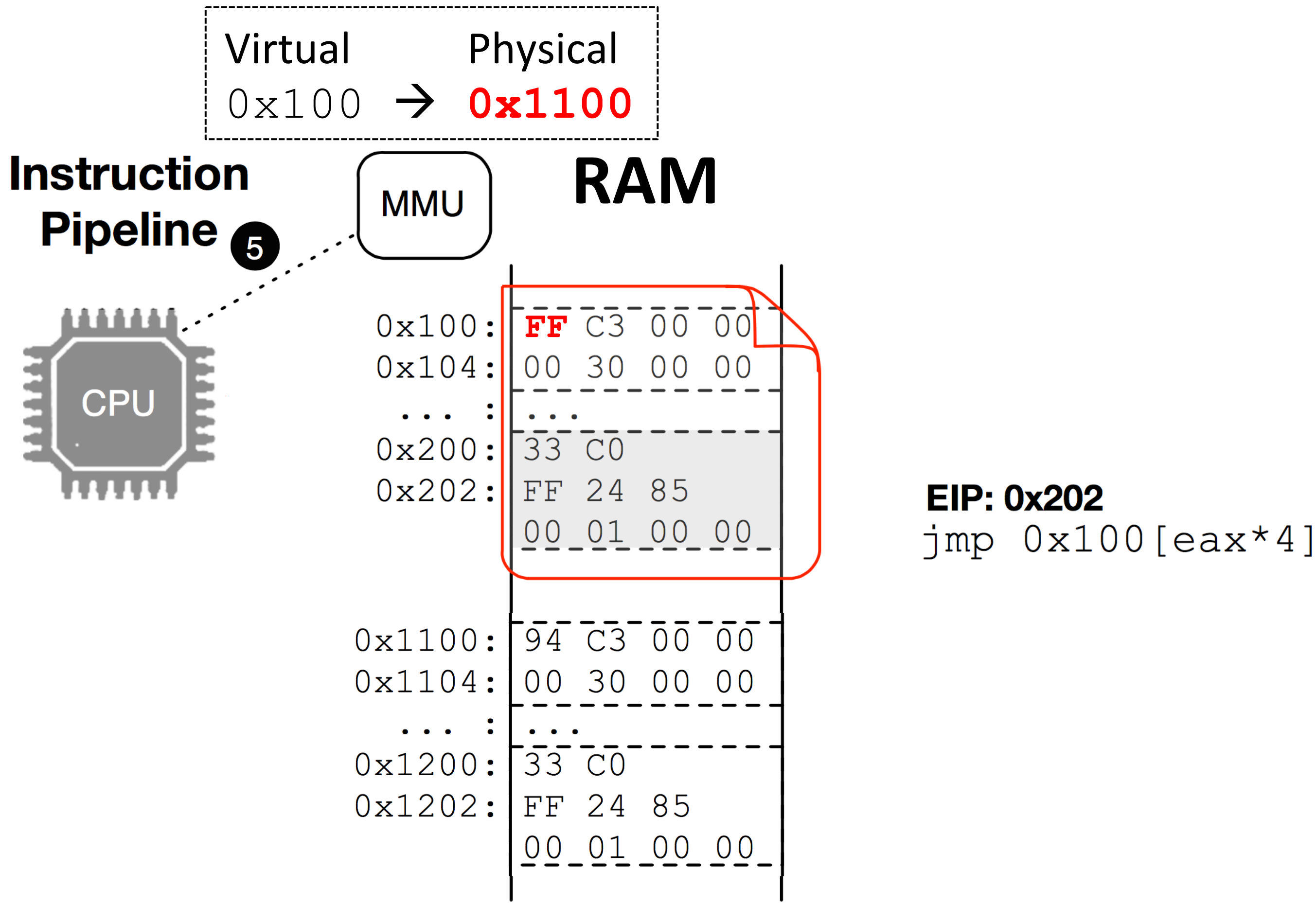
"Destroying" the executable byte that is read



Destroy the original executable byte

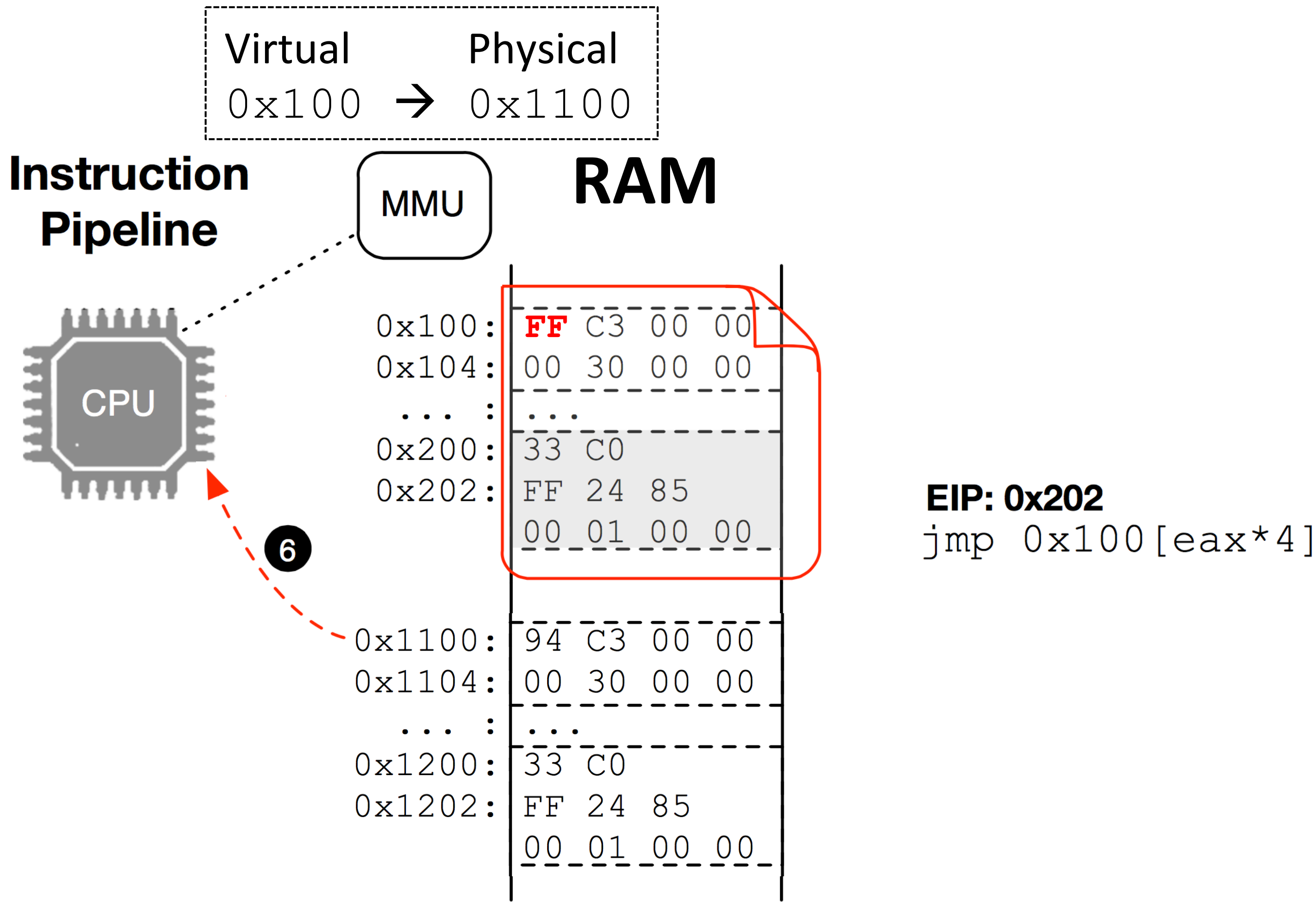
Destructive Code Reads

"Destroying" the executable byte that is read



Destructive Code Reads

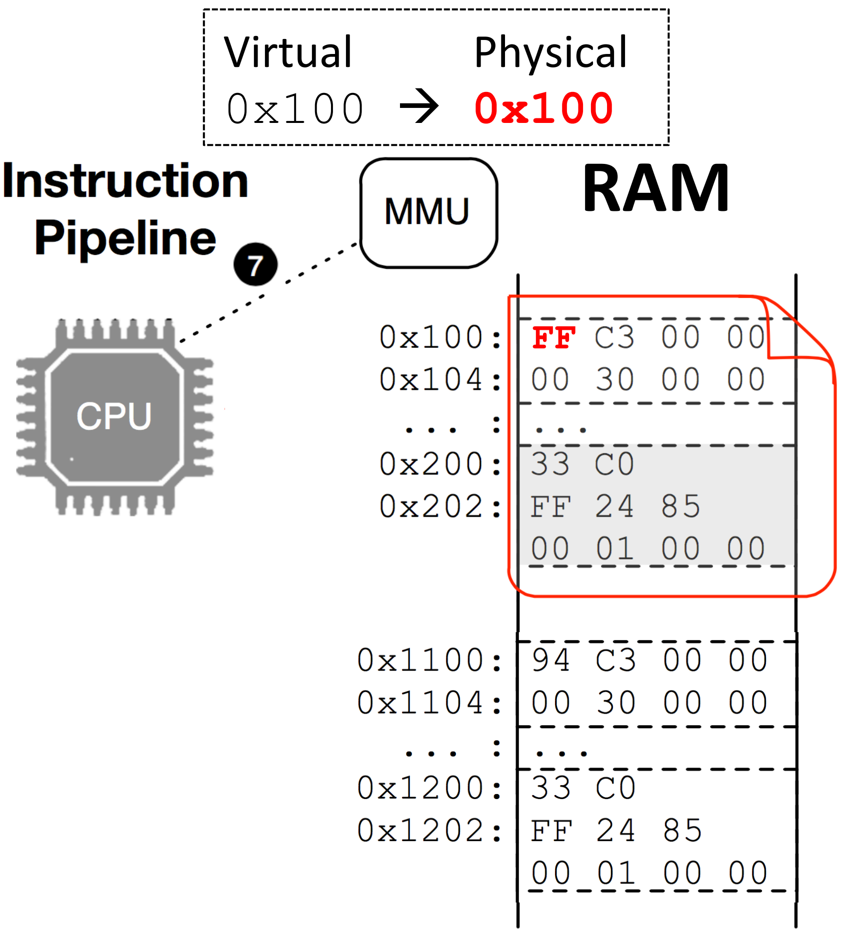
"Destroying" the executable byte that is read



Service the original read operation

Destructive Code Reads

"Destroying" the executable byte that is read



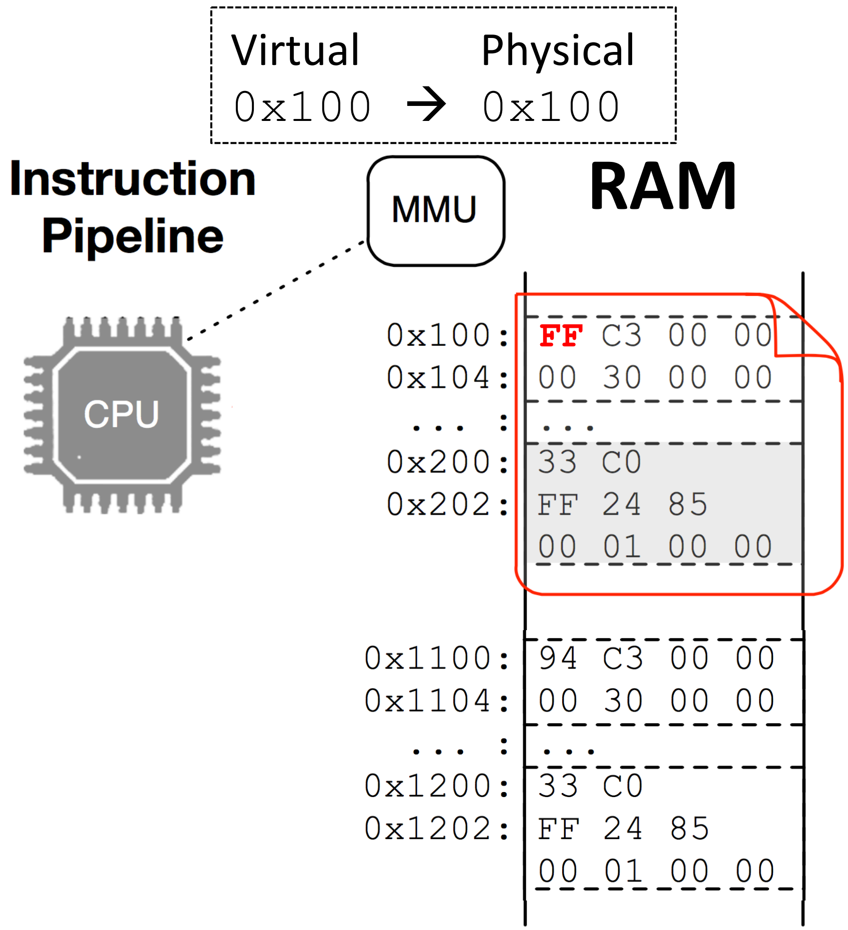
EIP: 0x202

`jmp 0x100[eax*4]`

Restore MMU translation

Destructive Code Reads

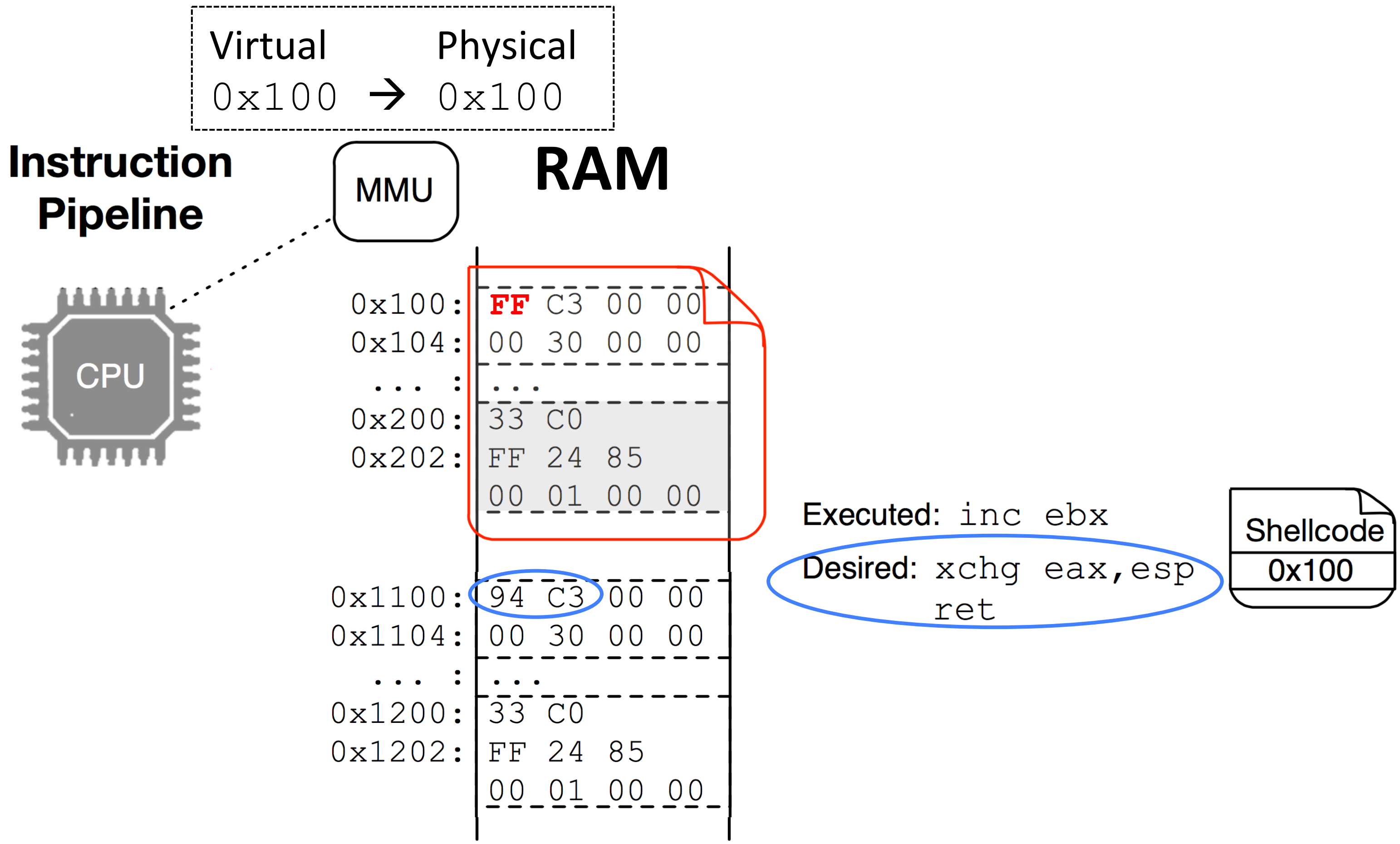
Stopping a dynamic code reuse attack



Assume memory at 0x100 was disclosed as part of an attack

Destructive Code Reads

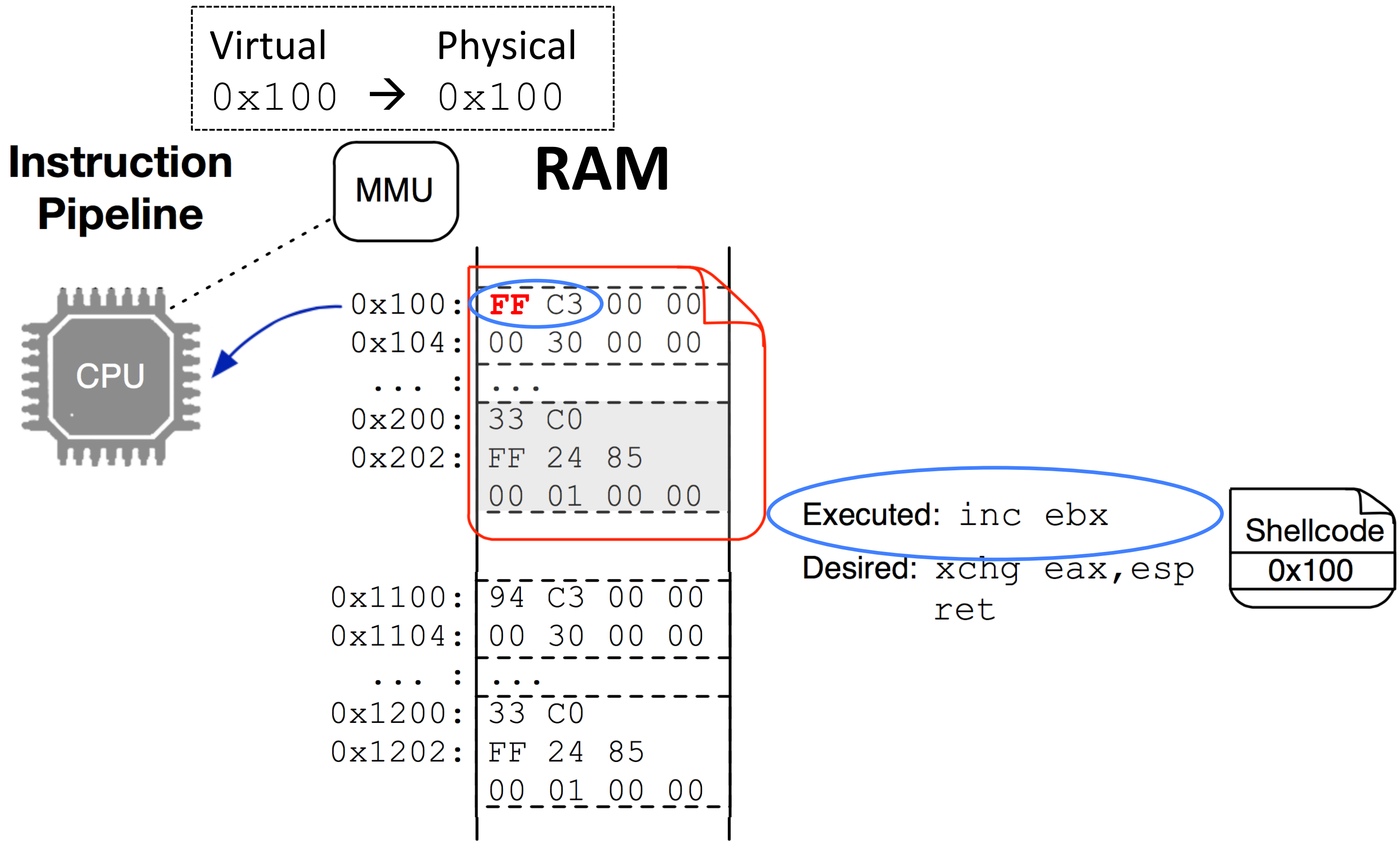
Stopping a dynamic code reuse attack



Attacker *assumes* that he found a stack pivot gadget

Destructive Code Reads

Stopping a dynamic code reuse attack

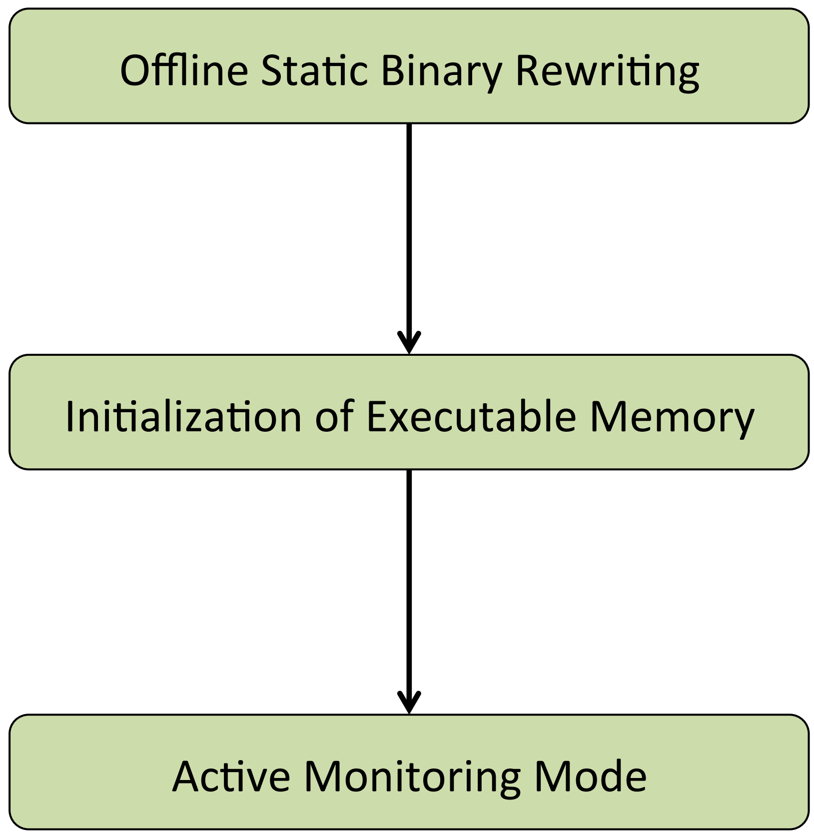


The desired gadget was not executed

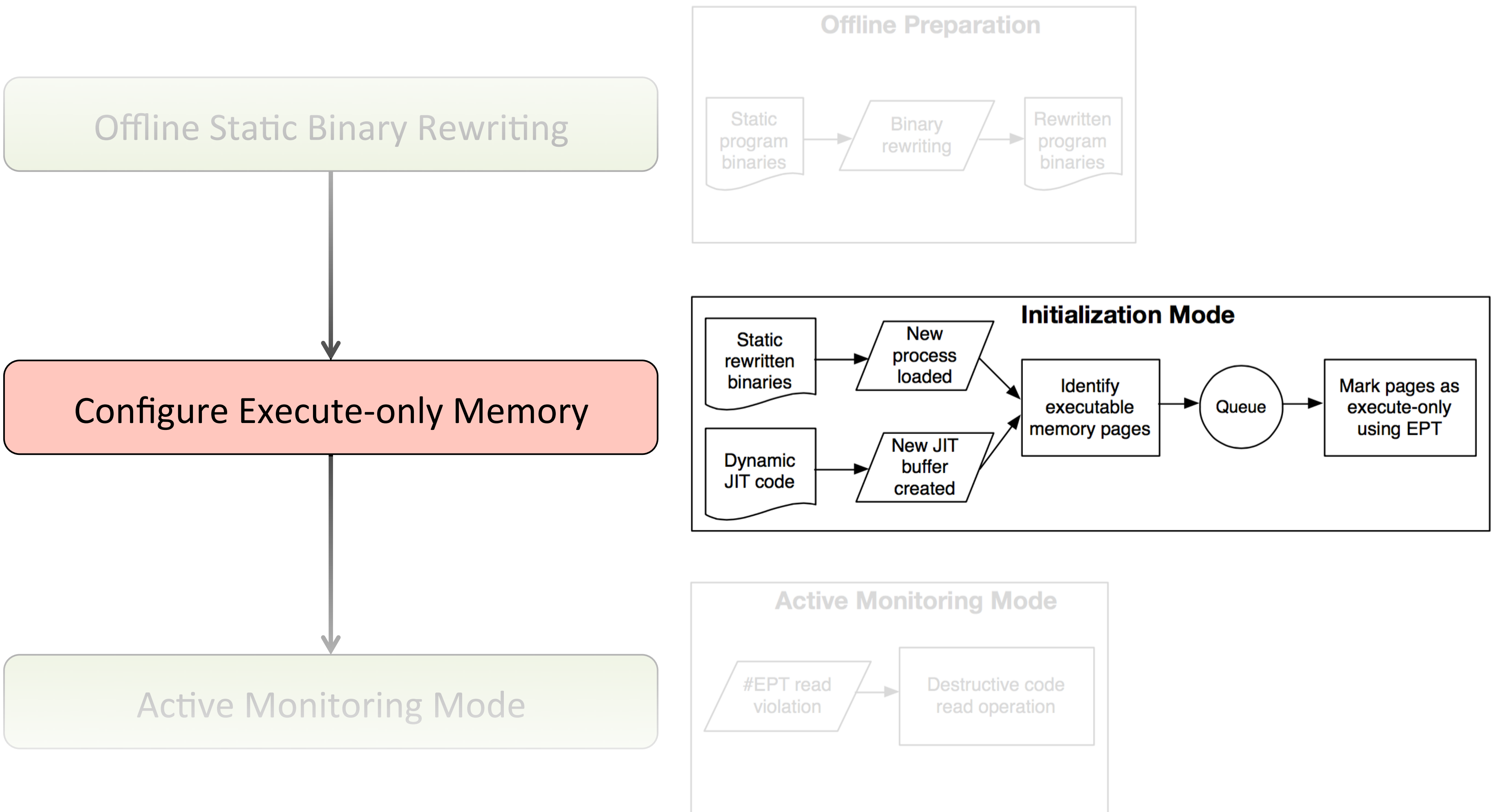
Outline

- Destructive Code Reads
- System Implementation
- Evaluation
- Future Work

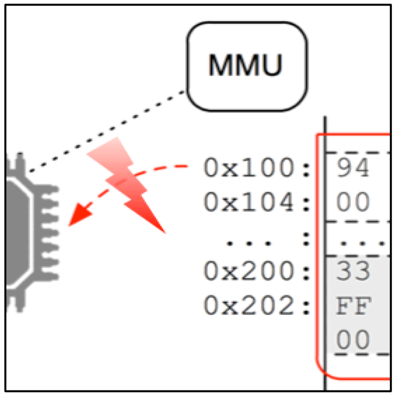
System Implementation



System Implementation

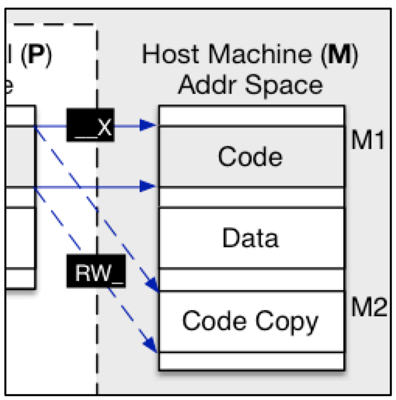


Key Requirements for Destructive Code Reads



"When" to mediate?

Detect read operations into executable memory



"How" to mediate?

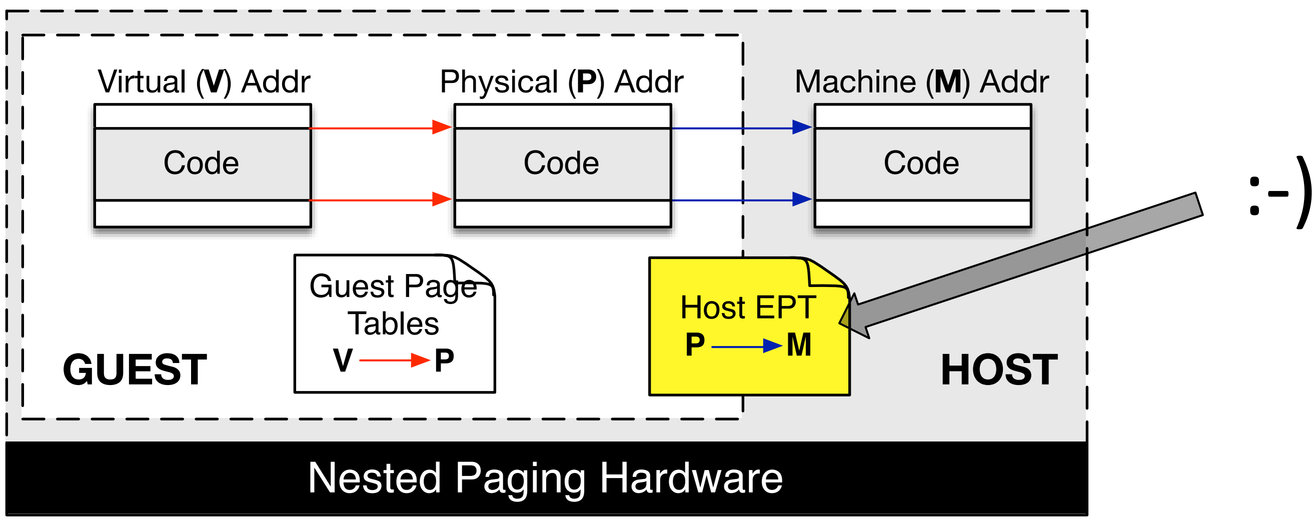
Maintain separate code/data views for same (virtual) memory address

Hardware-Assisted Nested Paging is a key enabler

Hardware-Assisted Nested Paging

Hardware feature to improve virtualization performance:
  Translate guest to host addresses in hardware

Different implementations:
  Intel EPT*
  AMD RVI

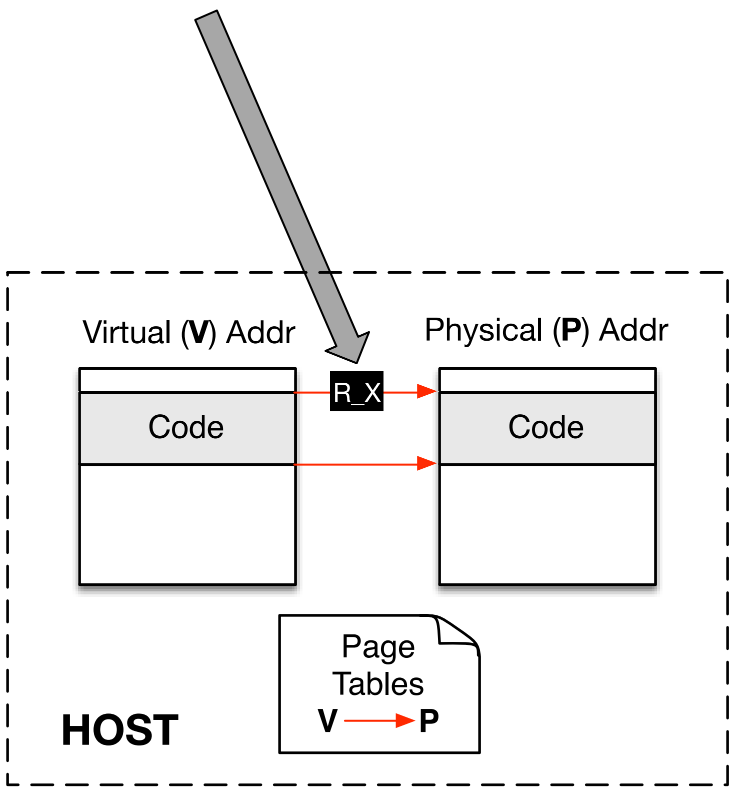


** EPT: Extended Page Tables*
RVI: Rapid Virtualization Indexing

When to Mediate

(1) Efficient detection of reads into executable memory

Problem: OS native paging cannot mark memory as execute-only

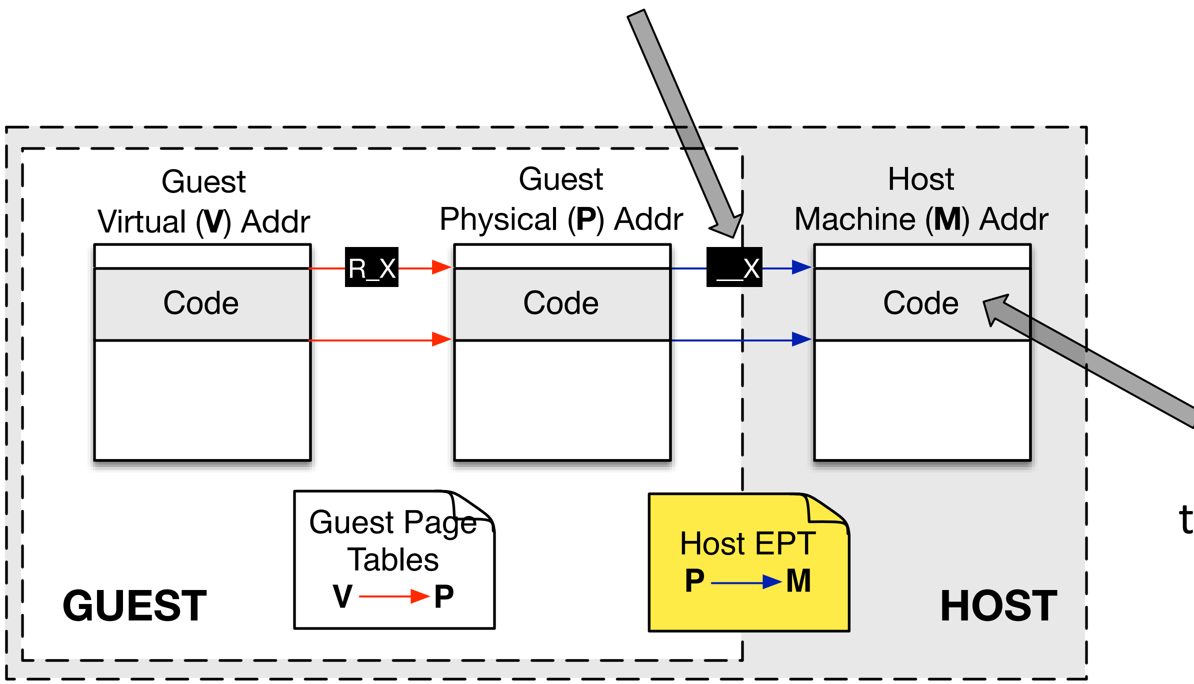


When to Mediate

(1) Efficient detection of reads into executable memory

Problem: OS native paging cannot mark memory as execute-only

Solution: Virtualize the host and use Intel EPT to mark execute-only



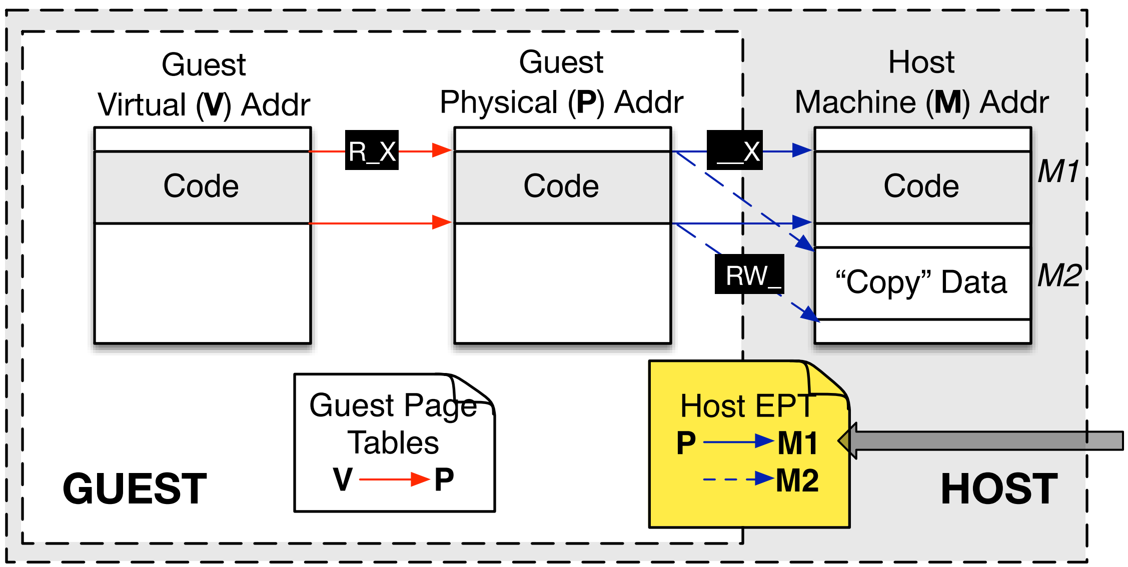
Reads into this page will trap into hypervisor

How to Mediate

(2) Efficient maintenance of separate code/data views

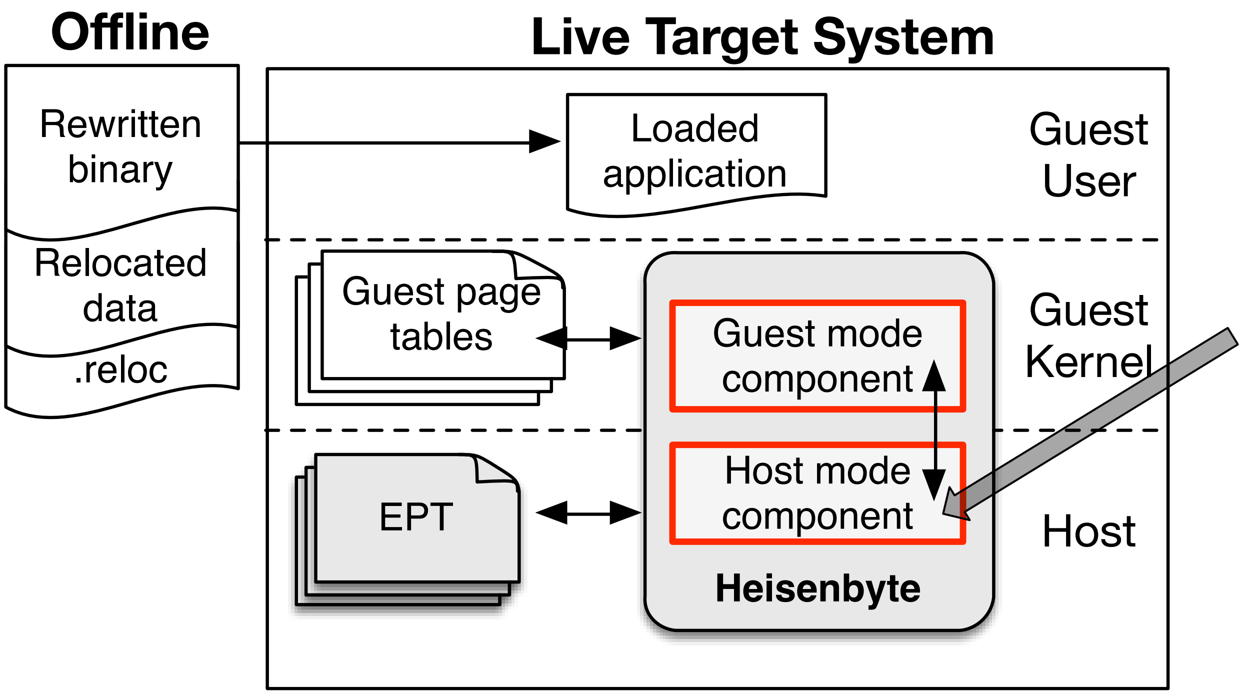
Goal: Induce different program behavior at the same virtual address depending on read or execute operation

Solution: Manipulate EPT to redirect memory translation at runtime



Change translation at runtime

Architecture (Para-virtualized)

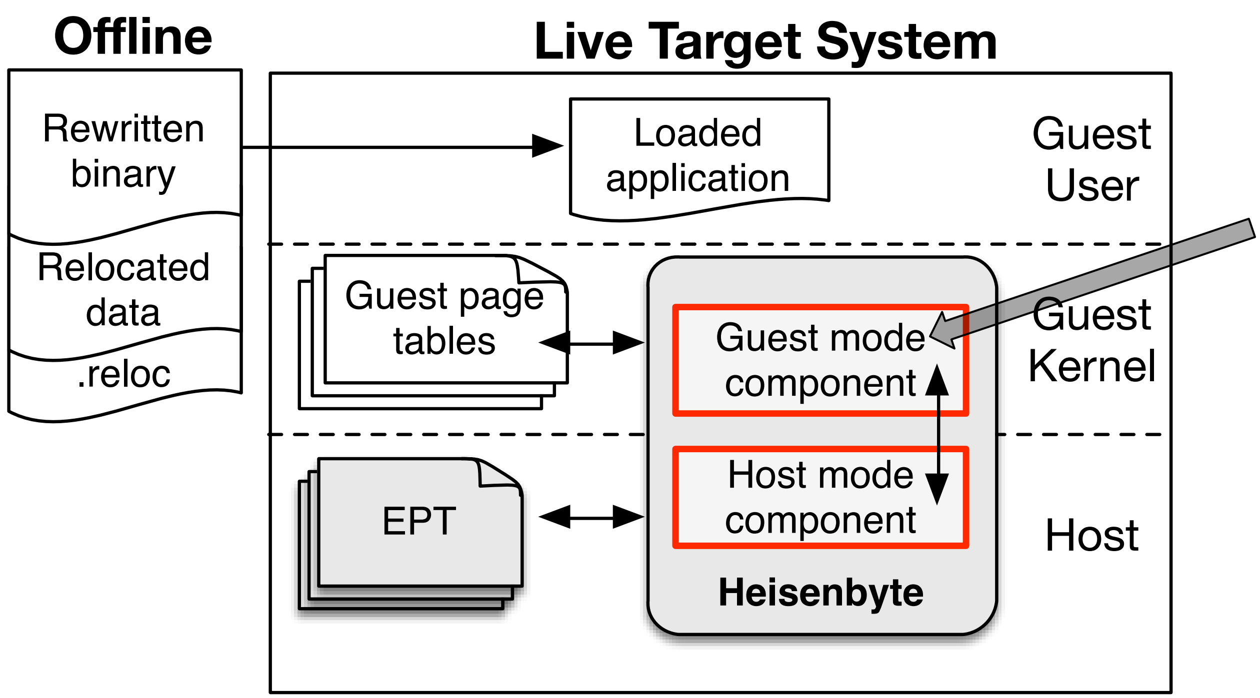


Configure execute-only mem

When and how to mediate

Destroy code when read

Architecture (Para-virtualized)



Identify executable mem

Induce COW

Tracking Runtime Executable Memory

How to identify executable memory we want to protect?

(1) Static program binaries
- Windows OS-provided runtime callbacks for
  - New/exiting processes
  - Loaded libraries

(2) Dynamic JIT code
- Inline hooking of Windows memory management APIs
- Perform hypercalls to hypervisor when
  - Exec buffer → Non-exec
  - Non-exec buffer → Exec
  - Exec buffer → Freed

[More in paper …] Optimizations and Windows-specific implementation details

Tracking Runtime Executable Memory

Challenges

Challenge 1: Shared physical memory pages across processes

Solution: Induce Copy-On-Write (COW) on pages with 1-byte identity write operation to each page

Challenge 2: Demand paging – pages could be paged out

Solution: Make pages resident in physical memory using `MmProbeAndLockPages()` kernel API

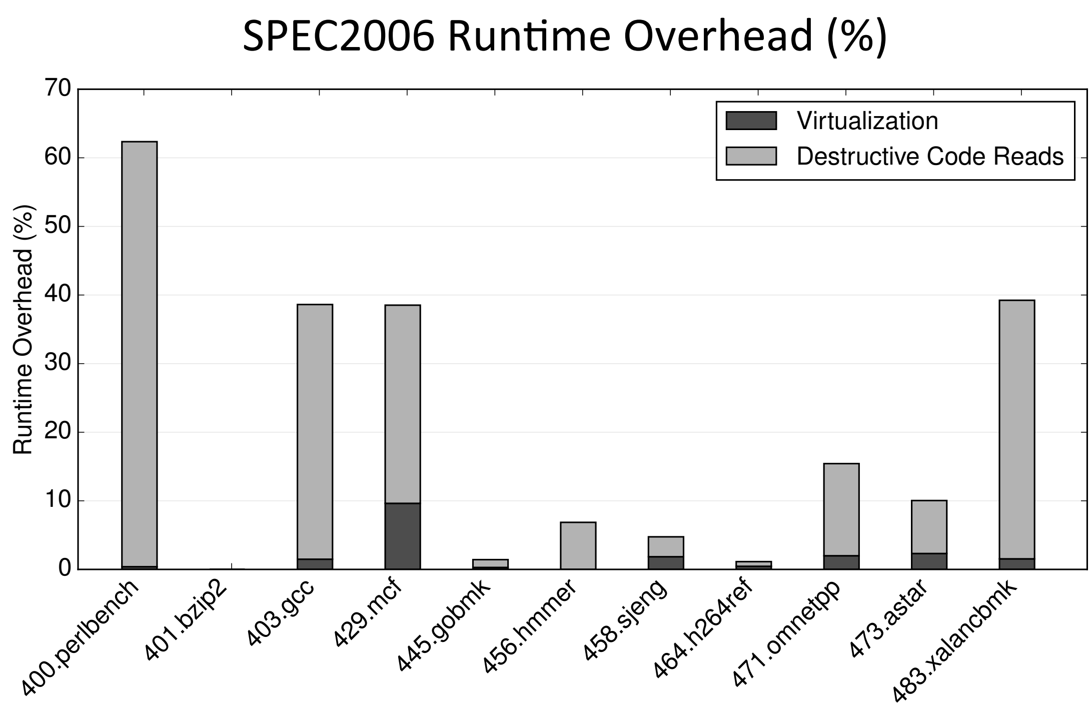
Some Caveats to HEISENBYTE

- Cannot handle code that reads/writes to *itself*
  - Eg. Self-modifying code
- Cannot mitigate attacks that reveal contents of memory *without directly reading* executable memory
  - Eg. Fault-based side-channel attacks (Blind-ROP)
- Need support for *fine-grained* ASLR
  - Eg. Instruction-level in-place code randomization
- *One-byte* code "destruction" regardless of operand size of read operation

Outline

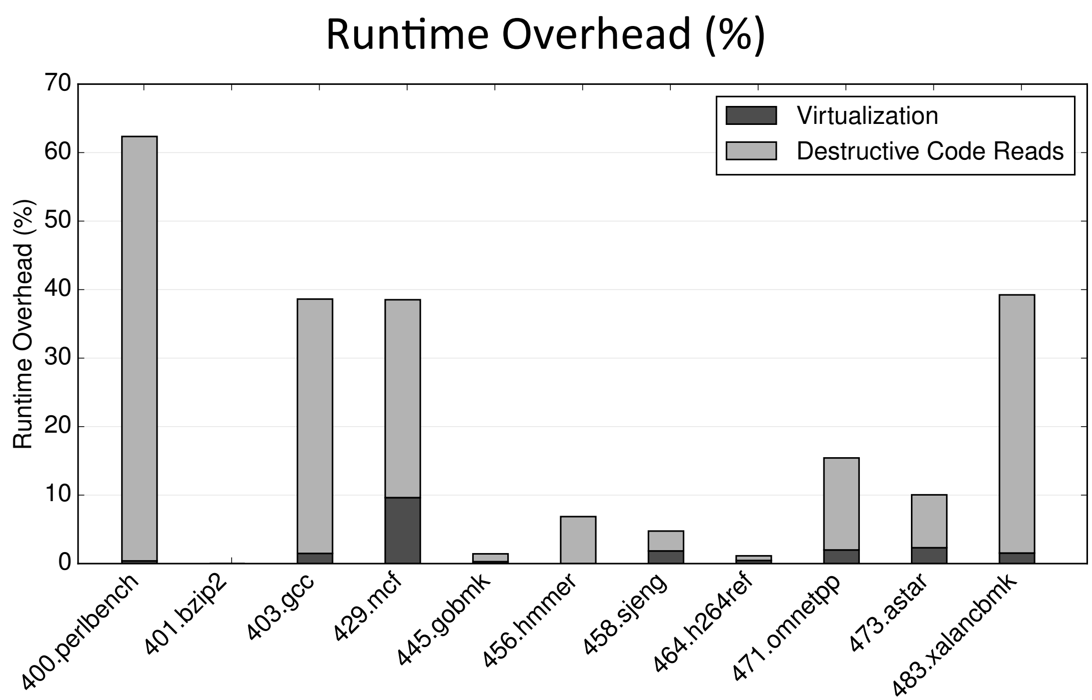
- Destructive Code Reads
- System Implementation
- Evaluation
- Future Work

Evaluation – Execution Overhead



"Destructive code reads" overhead depends on how imperfect the separation of data from code in executable sections

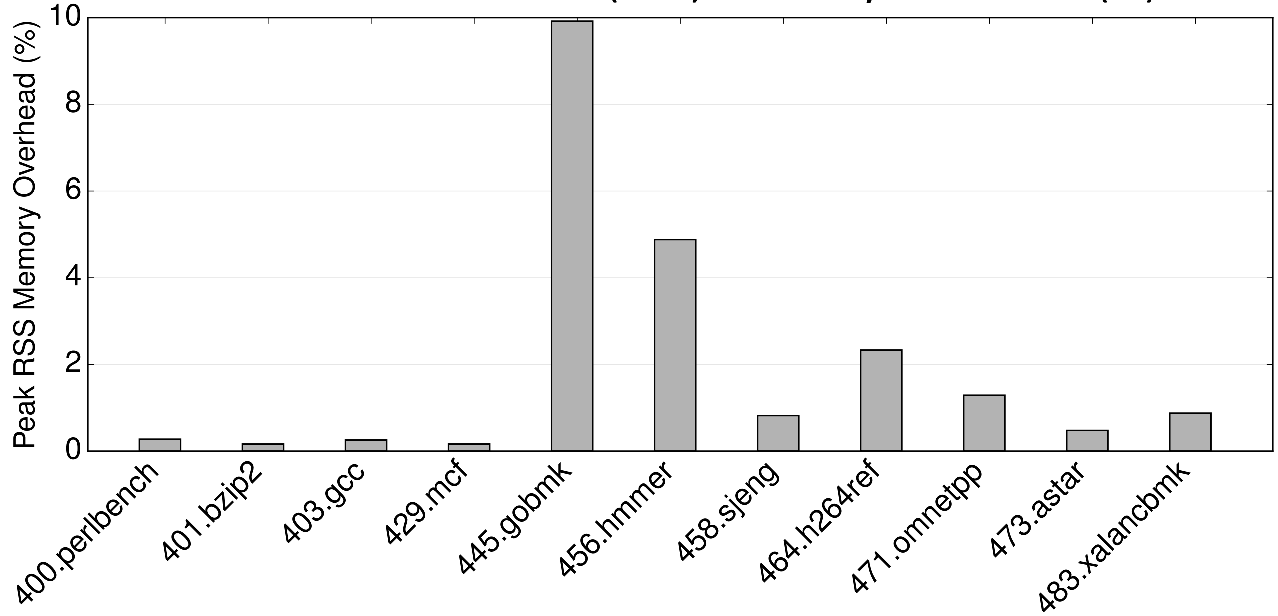
Evaluation – Execution Overhead



Virtualization avg overhead: ~1.8%
Destructive code reads avg overhead: ~16.5%

Evaluation – Memory Overhead

Peak Resident Set Size (RSS) Memory Overhead (%)



Peak RSS memory avg overhead: ~0.8%

Evaluation - Security

HEISENBYTE corrupts code with debug trap code `0xCC`

Crafted dynamic code reuse exploits and monitor for invoked debug trap

(1) Dynamic code

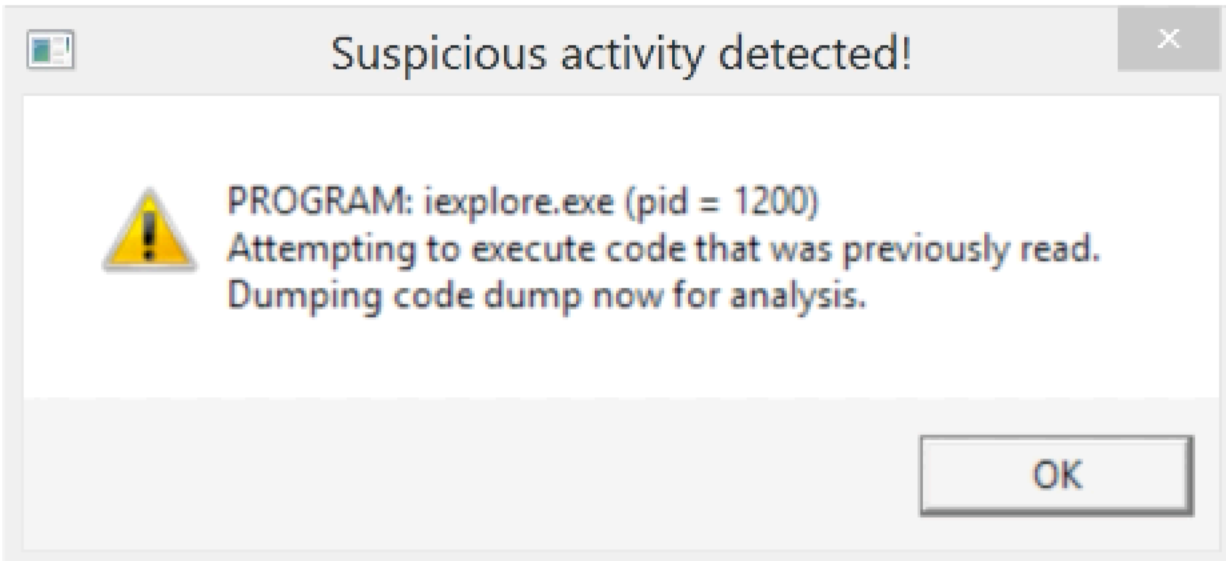
- Self-injected bug in toy program that mimics the creation of a JIT code buffer

(2) Static code

- CVE-2013-2551: Internet Explorer Bug

Exploits on both static programs and dynamic JIT code triggered debug traps

Evaluation – Demo on Win8 / IE10



Outline

- Destructive Code Reads
- System Implementation
- Evaluation
- Future Work

Future Work

- Improve code/data separation task in disassembly for Windows COTS binaries
  - Record read operations into executable memory to guide disassembly and binary rewriting
- Lower overhead of destructive code reads
  - Use new virtualization-based hardware features in Haswell+ processors (Eg. New #VE exception)
- Explore value of destructive *data* reads

Conclusions

Key Idea: Make exec. mem. indeterminate after it has been read

- New security concept: "Destructive code reads"
- One application: Mitigate memory disclosure attacks
- Heisenbyte is a practical solution
  - Works with imperfect disassembly on COTS binaries
  - No instrumentation on the binaries
  - JIT code works too

Thank you!

Adrian Tang • @0x0atang