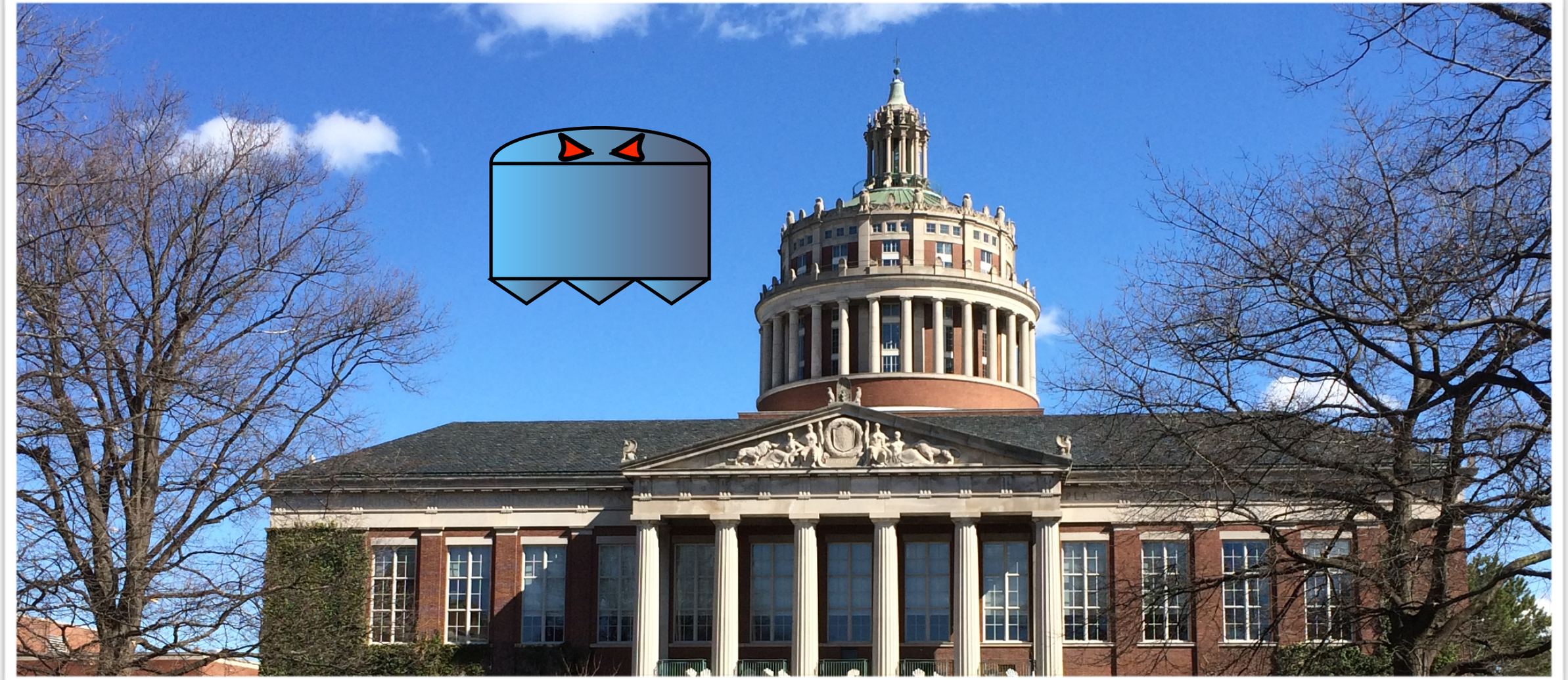


Meliora!

SVA Tutorial

John Criswell
University of Rochester

Introduction

LLVM is Great!

Compilers!

JITs!

Security Hardening Tools!

Formal Verification!

Bug Finding Tools!

Profiling Tools!

What if your program is an operating system kernel?

Operating Systems are Just Programs

- Code has global, stack, and heap like other programs
- Code manipulates data structures like other programs

Operating System is a Special Program

- Privileged hardware configuration
  - Page tables
  - Interrupt Vector Table
- State manipulation
  - Context switching
  - Signal handler dispatch

Challenge 1: Assembly Code

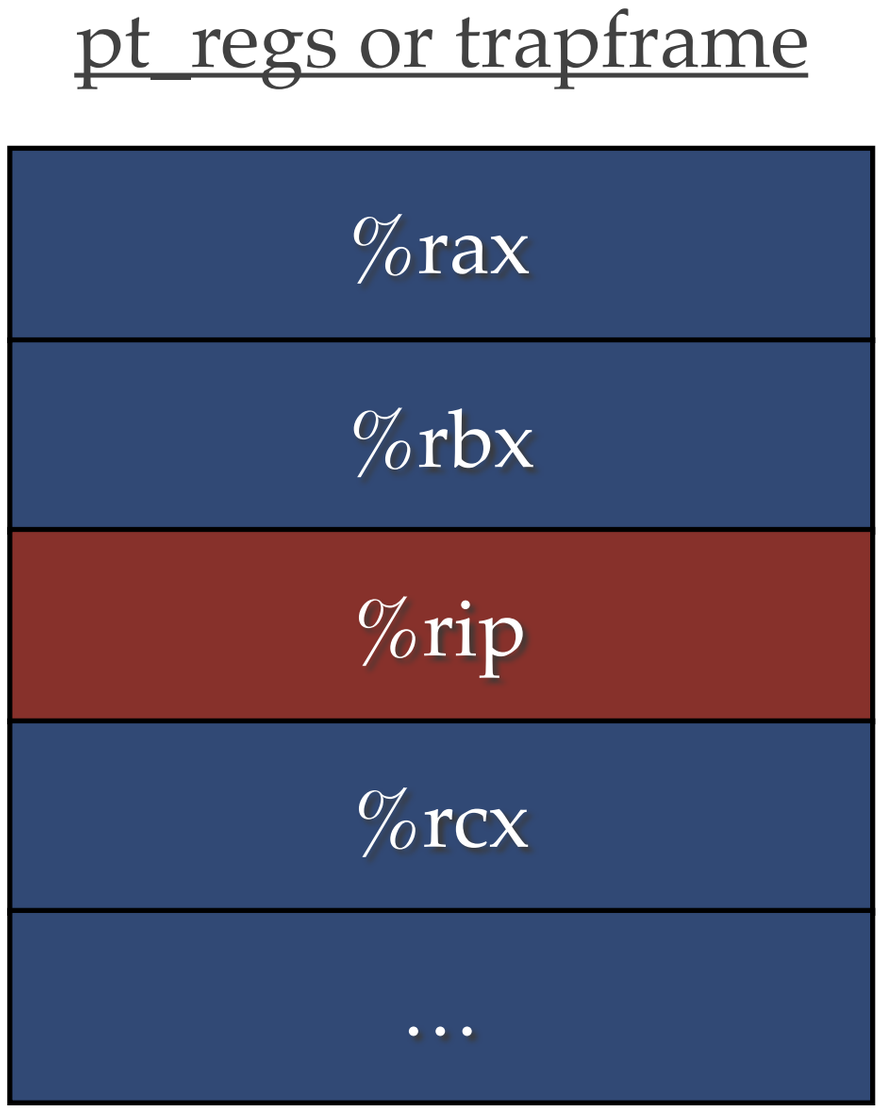
- Assembly code is hand-written
- Very difficult to analyze
- The value of LLVM is that you avoid binary analysis

Challenge 2: Memory with Side Effects

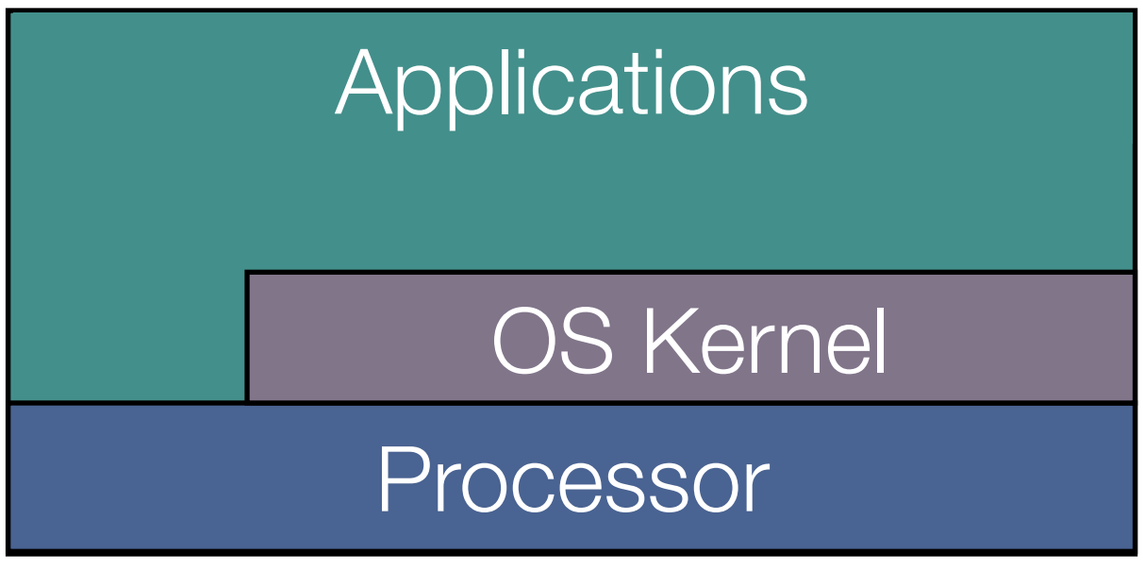
- Loads and stores access the following
  - Memory objects (global, stack, heap)
  - Page tables
  - Interrupt vector tables
  - Interrupted Program State
  - State saved on context switch

Stores to Memory

- Does store to memory modify control flow?

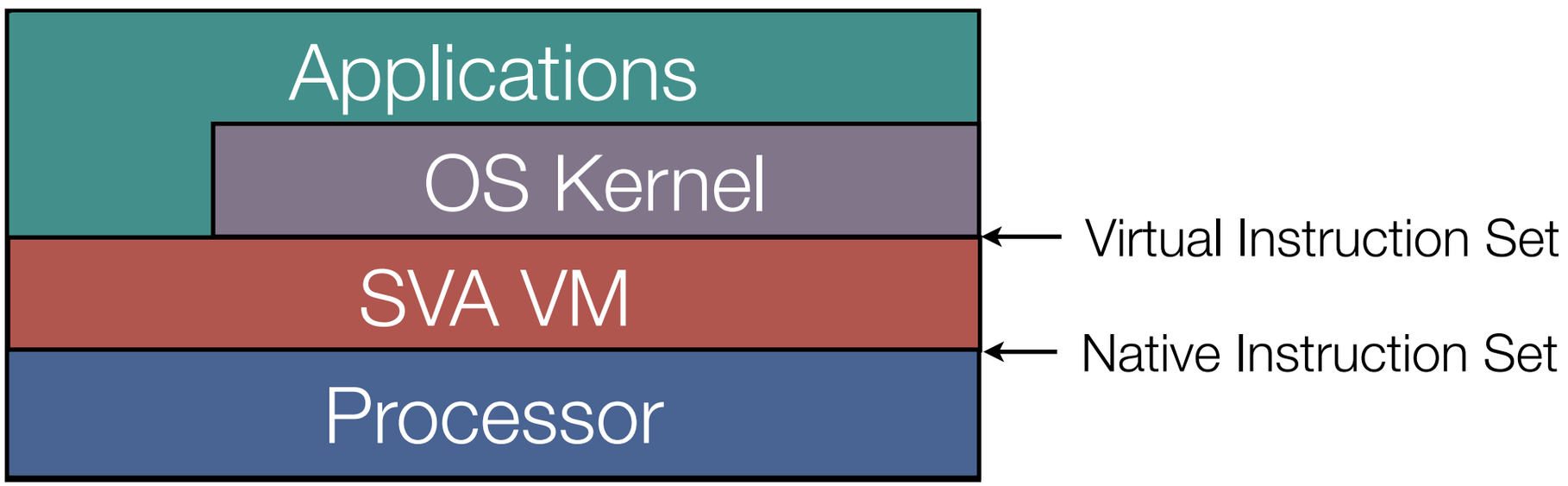


Secure Virtual Architecture



- OS compiled to virtual instruction set
  - Designed to be easy to analyze and instrument
  - Low-level instructions (SVA-OS) replace assembly code
- Translate ahead-of-time, boot-time, or run-time

Secure Virtual Architecture

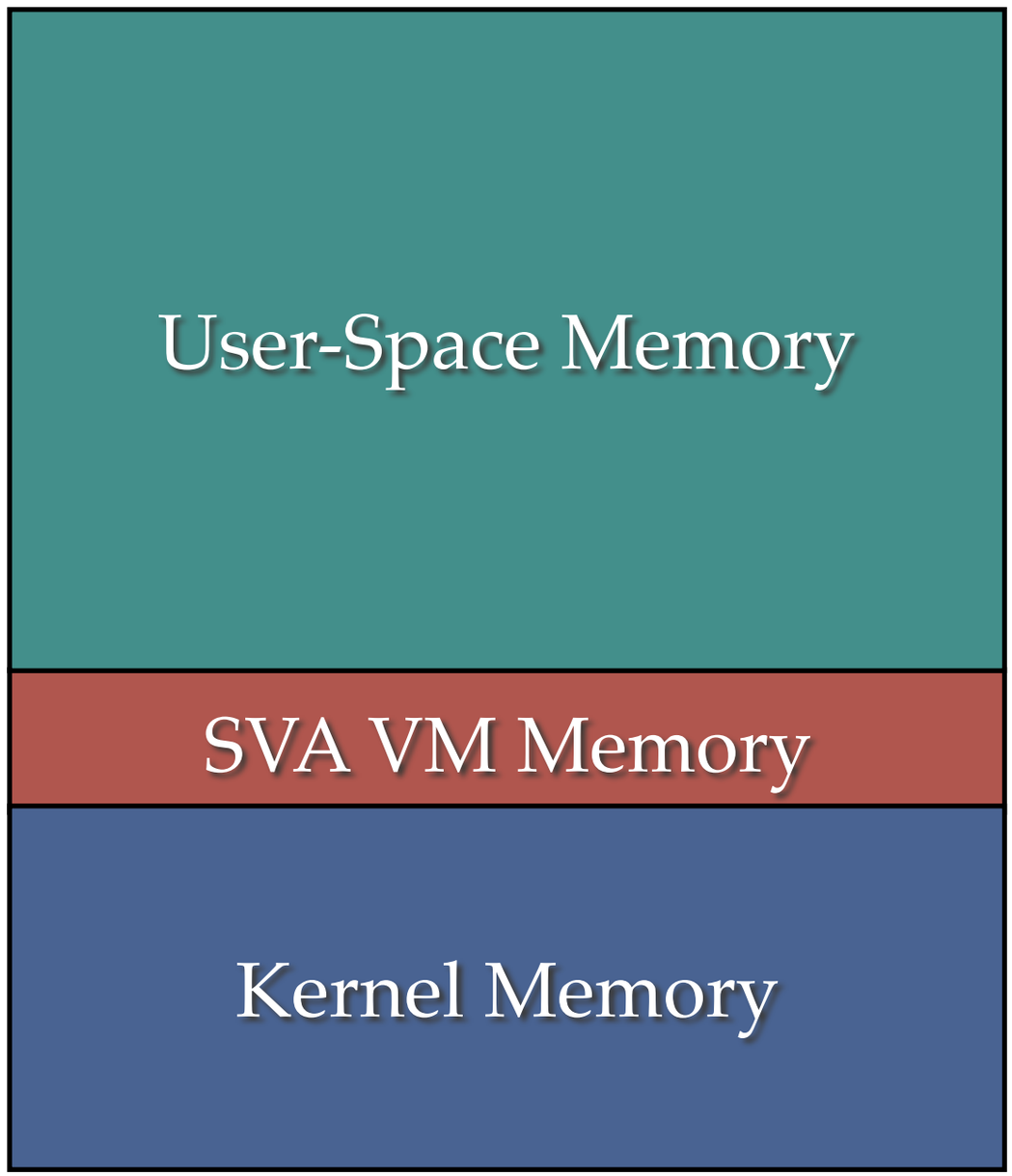


- OS compiled to virtual instruction set
  - Designed to be easy to analyze and instrument
  - Low-level instructions (SVA-OS) replace assembly code
- Translate ahead-of-time, boot-time, or run-time

SVA Virtual Instruction Set

- SVA-Core: Compiler Instrumentation
  - Based on LLVM IR: Typed, Explicit SSA form
- SVA-OS: SVA Runtime
  - OS-neutral instructions to support a commodity OS
  - Encapsulates & controls hardware and state manipulation

SVA Virtual Address Space Layout



- Store opaque state in SVA VM memory
- Protect SVA VM memory from corruption
  - Memory safety
  - SFI

State Saved in SVA VM Memory

- Privileged hardware state
  - Interrupt vector table
  - System call table
  - Page table pages
- Program state
  - State saved on context switch
  - State saved on interrupt, trap, system call

Opaque Native State

State	Purpose
Integer State	Represents the native state on the CPU
Interrupt Context	Represents native state saved on an interrupt, trap, or system call

SVA-OS Instructions

Handler Registration

Function	Description
sva.register.interrupt()	Register interrupt handler
sva.register.trap()	Register trap handler
sva.register.syscall()	Register system call handler

MMU Configuration

Function	Description
sva.declare.ptp()	Mark memory as page table page
sva.update.mapping()	Modify entry in a page table page
sva.release.ptp()	Remove page from page table page
sva.mm.load.pagetable()	Load page table pointer on to CPU
sva.mm.save.pagetable()	Save copy of page table pointer on CPU

State Manipulation Instructions

Function	Description
sva.swap.integer()	Context switch to a new thread
sva.new.thread()	Create new thread
sva.icontext.save()	Save interrupt context on to thread interrupt context stack
sva.icontext.load()	Load interrupt context on thread interrupt stack

Further Reading

- SVA (SOSP 2007, Usenix Security 2009)
- KCoFI (IEEE S&P 2014)
- Virtual Ghost (ASPLOS 2014)
- Apparition (Usenix Security 2018)
- **Shade (VEE 2019) (that's today!)**
- Criswell Ph.D. Disseration
  - Appendix A describes SVA-OS instructions