Secure Virtual Architecture

A Framework for Securing And Certifying Software on Commodity Systems

Vikram Adve

University of Illinois at Urbana-Champaign

Overview

- **Problem statement: "Security Inversion"**
  - Commodity OSs, browsers are buggy, vulnerable
  - Untrustworthy foundation for secure applications
- **State of the art is inadequate**
  - Performance; application recovery; DoS; system complexity
- **SVA: Building blocks for better solutions**
  - Safe execution environment for a complete commodity OS
  - Automatic recovery from faults in the core kernel
  - Easier certifying compilation (compared with binary code)
- **Building on SVA to Mitigate the Security Inversion**

Commodity Systems Are Vulnerable

US-CERT: 5198 O.S. vulnerabilities reported in 2005 [A]

- **812** on Windows, **2328** on Unix/Linux, **2058** on multiple systems

Month of Kernel Bugs (Nov. 2006)

- One new bug ***every day***
- Linux: **8**, MacOS: **8**, FreeBSD: **2**, Solaris: **1**, Windows: **1**
- Code injection: **13**, DOS: **13**, Memory corruption: **4**
- Month of Apple Bugs (Jan 2007): 23 new bugs in MacOS!

Browser Attacks

- Overview: See *http://www.cert.org/advisories/CA-2000-02.html*
- https (SSL) doesn't help; e.g., exploit scripts are embedded in legitimate HTML
- Common web server security policies (e.g., "same origin") don't help

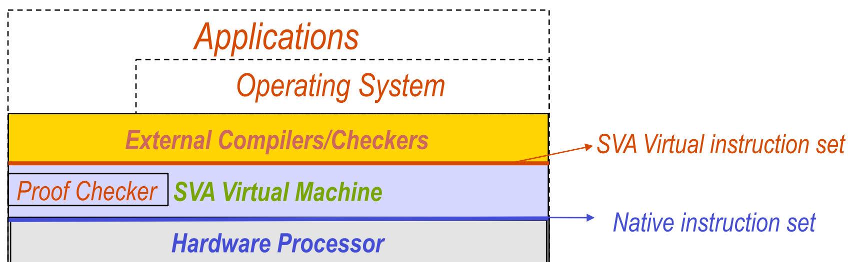
Kernel Sizes and Bug Estimates [B,C]

- ***Windows:*** 40M total, **5M kernel**, 5,000-35,000 bugs
- ***Linux:*** 30M total, **2.5M kernel**, 2,500-17,500 bugs
- Assumes 1-7 errors per 1K lines, as estimated by SEI for commercial software

(A) http://www.us-cert.gov/cas/bulletins/SB2005.html
(B) Bill Worley and Mark Beckett, Eliminating Malware and Rootkits, Secure64 White Paper, July 2006
(C) Tanenbaum, Can We Make Operating Systems Reliable and Secure?, Computer Magazine, Feb. 2006

Secure Virtual Architecture

- ***Compiler-based*** *VM* ***below*** *operating system*
  - *Compiler-based* : think **JVM**, i.e., using (LLVM) bytecode
  - *Below OS* : think **Hypervisor**
- Supports type-unsafe languages (e.g., C/C++)
- "Safe execution environment" for complete OS (e.g., Linux)



Virtual Instruction Set

SVA-Core

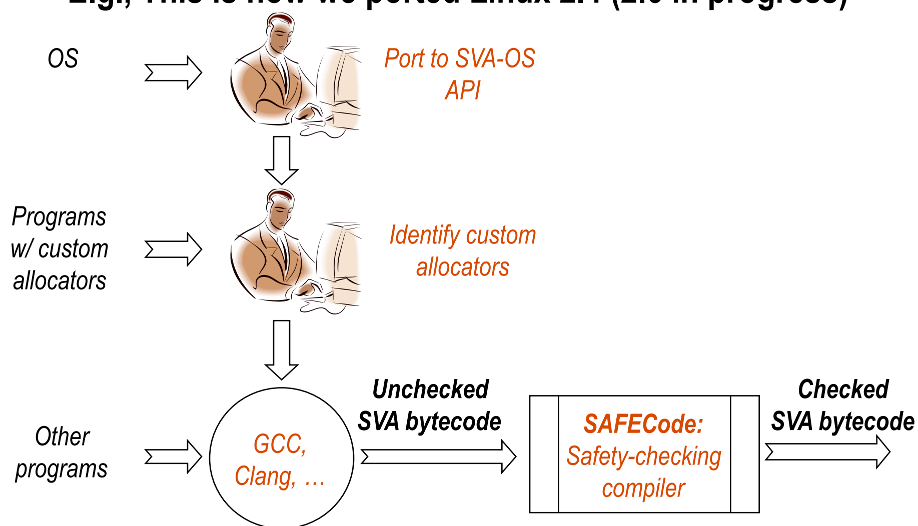
- Based on LLVM
- Typed; explicit CFG; explicit SSA form
- Sophisticated interprocedural compiler techniques
  - Pointer analysis, Automatic Pool Allocation, Dead arg. elimination; …
- Easier certified compilation than “down to” native binary code

SVA-OS

- OS-neutral instructions support commodity OSs
- Removes difficult to analyze assembly code
- Encapsulates privileged operations
- Like porting to a new hardware architecture

Porting Software to SVA

E.g., This is how we ported Linux 2.4 (2.6 in progress)



SVA Safety Guarantees

Strong Guarantees

- **Memory safety:** E.g., no uninitialized pointer uses, no array overflows
- **Type safety** for a *subset* of objects
- **Control flow integrity:** only follow compiler-predicted execution paths
- **Sound operational semantics** ⇒ sound static analysis

Primary Weakness

- **Dangling pointer errors may occur**: avoid need for GC
- ***They do not invalidate the above guarantees!***

Practical

- Completely automatic for user space: no wrappers, no GC
- Minimal porting effort for kernel code
- Works for arbitrary C programs
- Relatively low overhead for C programs

Low-level Hardware-Software Interactions

Even *type-safe* OS code can invalidate safety guarantees

- ☹ Could overwrite to processor state in memory
- ☹ Could wrap in mismatched <CPU state, stack>
- ☹ Could violate memory-mapped I/O requirements
- ☹ Could insert illegal MMU mappings
- ☹ *Cannot* use self-modifying code, but would like to (in limited ways)

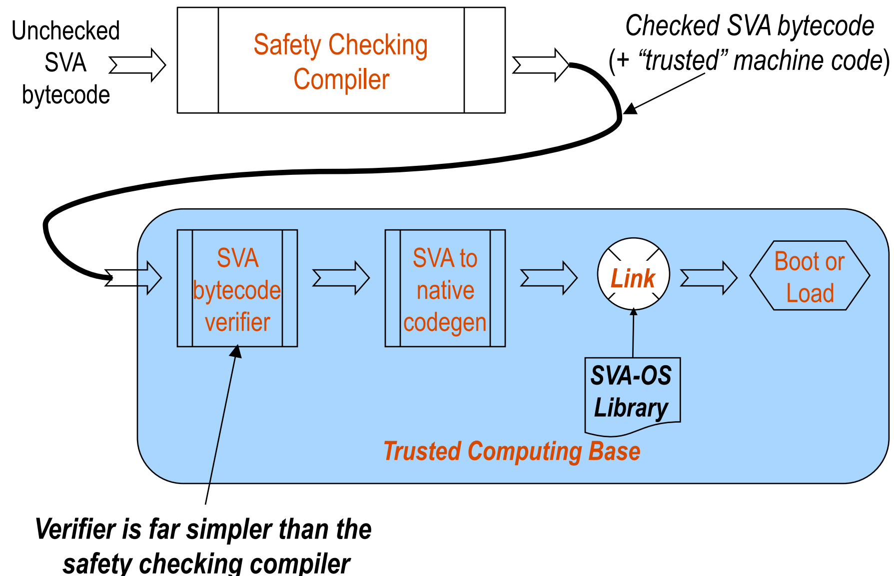
SVA-OS: Memory Safety with Hardware-Software Interactions

- Redesign SVA-OS interface to secure all low-level interactions
- Use pointer analysis to reduce run-time checks

Transparent Recovery from Kernel Faults

- **Problem**
  - Faults, exploits of kernel: hurts availability; denial of service
  - Device drivers not enough: Numerous errors are in *core* kernel
- **Solution**
  - **Recovery Domains:** Principle for recoverable OS kernels
  - Key insight: Treat error as property of a *request*, not subsystem
  - Able to recover from 97% of injected faults *in core Linux 2.4*
  - **First system to recover from unexpected faults in *core* kernel**

Certifying Compilation

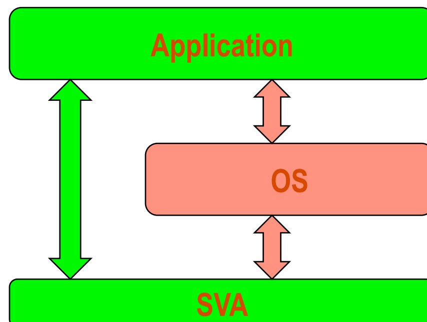


Building On The Building Blocks

Examples: Can we extend SVA to enable:

- Secure application memory?
- Secure inter-process information flow?
- Defenses against Denial-of-Service by OS?

Example: Secure Application Memory



Similar to Overshadow but with:

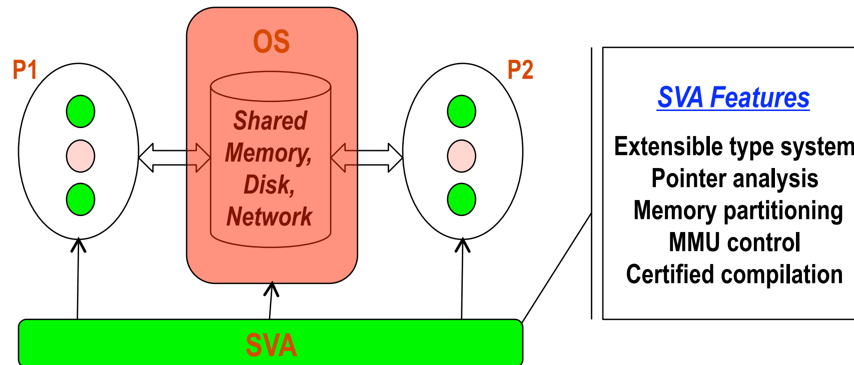
- Recovery domains
- Improved DoS protection
- Much simpler VM
- Lower overhead

SVA can *enforce* private address space via MMU control

SVA can intercept all system calls to encrypt/decrypt data

SVA does *not* need to kill application on detected exploit

Example: End-to-end Information Flow



SVA can *separately* analyze, transform P1, P2, OS:

- Can we enforce run-time labels at sender and receiver?
- Can we optimize run-time checks via memory partitioning?

Example: Denial of Service

- **Make DoS harder by combining:**
  - **Recovery domains** to avoid killing application(s) on exploits
  - **QoS scheduler** added to SVA
  - Hardware **performance isolation**
  - Supervision of **MMU mappings**
  - Standard VMM-like techniques for **file system protection**
- **But there are many hard issues here …**
  - OS can arbitrarily return error codes on system calls
  - OS can *ignore* network I/O?
  - OS mediates DNS lookups: DNS spoofing?

Extending The Building Blocks

Can we combine SVA with:

- Binary analysis and instrumentation techniques?
- Hardware support for isolation; performance isolation?
- Certified compilation from SVA-Core to native machine code
- Automatic verification of TCB?
- Secure web browser support?

Questions?

LLVM Provides Compiler Foundation

Framework for "lifelong compilation"

- Compile-time, link-time, install-time, run-time, "idle"-time
- IR: Compact, persistent, designed for effective optimization

Commercial quality compiler infrastructure

- **JIT:** Adobe, AMD, Apple, Intel, Mono, nVidia, RapidMind, …
- **Link-time Optimizer:** Apple, Cray, …
- **Static back end:** Apple, Cray, Aerospace, Microchip Tech, …
- **Silicon compilation:** AutoESL

Available at llvm.org

LLVM IR = Core Unprivileged Operations

```
/* C Source Code */
int SumArray(int Array[],
             int Num)
{
  int i, sum = 0;
  for (i = 0; i < Num; ++i)
    sum += Array[i];
  return sum;
}
```

- Architecture-neutral
- Low-level operations
- SSA representation
- Strictly-typed
- Mid-level type info

```
;; SVM Code
int %SumArray(int* %Array, int %Num)
{
bb1:
    %cond = setgt int %Num, 0
    br bool %cond, label %bb2, label %bb3
bb2:
    %sum0  = phi  int [%tmp10, %bb2], [0, %bb1]
    %i0    = phi  int [%inc, %bb2], [0, %bb1]
    %tmp7  = cast int %i0 to long
    %tmp8  = getelementptr int* %Array, long %tmp7
    %tmp9  = load int* %tmp8
    %tmp10 = add  int %tmp9, %sum0
    %inc   = add  int %i0, 1
    %cond2 = setlt int %inc, %Num
    br bool %cond2, label %bb2, label %bb3
bb3:
    %sum1  = phi  int [0, %bb1], [%tmp10,%bb2]
    ret int %sum1
}
```

SVA-OS Privileged Operations

Hardware Control

Interrupt, trap handlers

Page table entries

I/O operations

State Manipulation

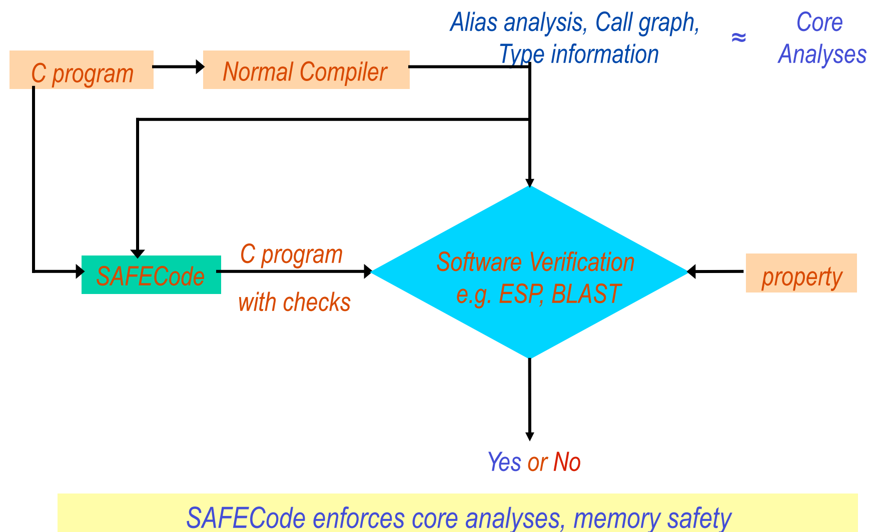
Virtual vs. Native State
- *Native state accessible but opaque*

Context-switching

Signal delivery

Interrupt Context
- *Exploit hardware for fast interrupts*

Sound Analysis with SVA



E.g., Static Analysis Using SVA

e.g., Client analyses in ESP, BLAST

Flow-sensitive clients: Only 2 changes:

- malloc can return non-fresh memory *in same points-to set*

Flow insensitive clients:

- Don't require any changes

Sound analyses for C are now possible (based on flow-insensitive, unification-based alias analysis).