Secure Virtual Architecture: A Novel Foundation for Operating System Security

Vikram Adve

University of Illinois at Urbana-Champaign

Joint work with:

John Criswell, Dinakar Dhurjati, Sumant Kowshik, Andrew Lenharth, Balpreet Pankaj

Thanks: **NSF, SRC, DARPA, Motorola, Apple**

Acknowledgements

Funding

- NSF (several programs), SRC, DARPA, Motorola, Apple

LLVM, pointer analysis (DSA), Automatic Pool Allocation

- Chris Lattner
- Other LLVM developers, past and present

SVA PrivOps design "consultants"

- Pierre Salverda
- David Raila

Other input and feedback

- Sam King, Roy Campbell
- Many reviewers

The Context - 1

Increasing threats to system software

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

Situation could get dramatically worse

- Incentives are increasing: "*botnets,*" online banking, identity theft
- Generations entering college are *much* more computer-savvy

(A) http://www.us-cert.gov/cas/bulletins/SB2005.html

The Context - 2

Vast majority of security-critical software is in C/C++

Existing Software

- "Commodity" OS: Windows, Linux, MacOS, BSD family, …
- Special-purpose OS: Secure64, QNX, …
- OS services: sshd, xinetd, named, sendmail, …
- Servers: Apache, **FIND OTHER NAMES …**

Existing Solutions

- Only ad-hoc, partial solutions are used in production systems
- E.g., non-executable stack/data, SFI, StackGuard, interface annotations, etc.
- Linux kernel has **2.5M** lines of code in ring 0, Windows has **5M** (B)

(A) "Eliminating Malware and Rootkits," Secure64 White Paper, July 2006

OS on a Safe Runtime: Vision

A *complete OS* in a safe execution environment

Eliminates an important class of security holes

Improves system reliability

Enables novel OS design techniques

- Application-specific extensions
- Run entire user processes in kernel address space
- Typed inter-process communication
- First-class HLL virtual machines
- Well-defined multithreading
- Novel compiler+run-time solutions to high-level security problems
- …

OS on a Safe Runtime: So Far

To date, *all* rely on a safe programming language:

- *Genera and others:* LISP
- *SPIN*: Modula 3
- *JavaOS, KaffeOS, ... :* Java
- *Singularity*: C#
- ...

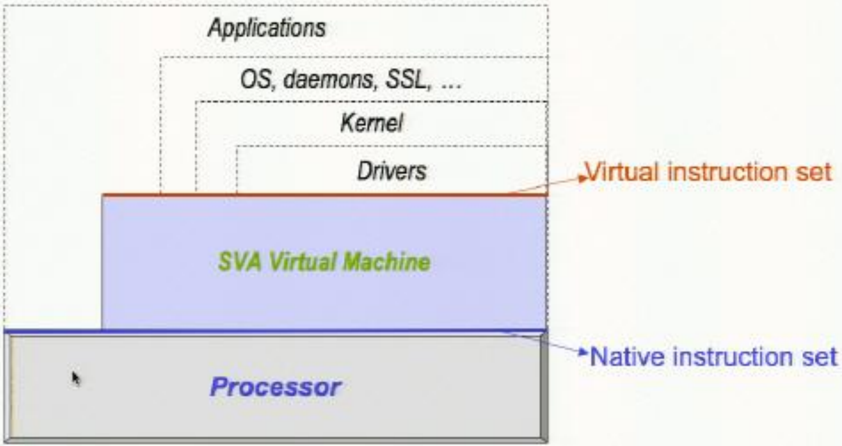
These projects led to many novel OS design techniques.

But what about today's commodity kernels?

Linux, MacOS, BSD family, Windows, ...

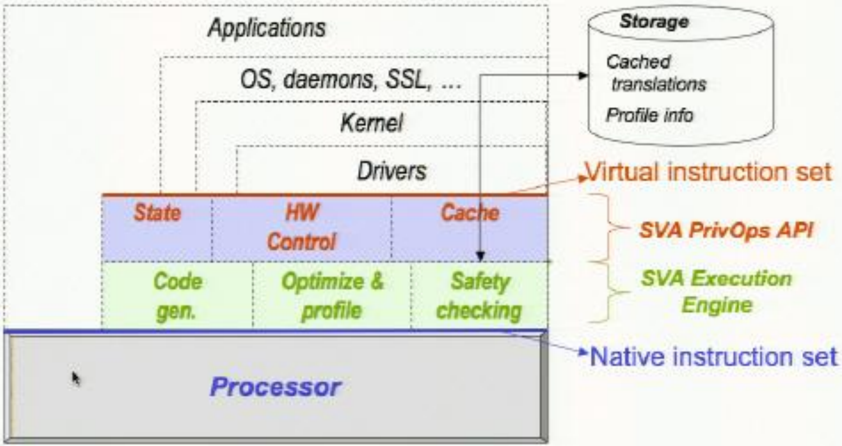
SVA: Secure Virtual Architecture

Designed to work with <u>legacy</u> kernels, e.g., Linux



SVA: Secure Virtual Architecture

Designed to work with legacy kernels, e.g., Linux



Categories of Security Problems

Solved with a safe execution env.

Buffer overflows

Dangling pointers

Format string errors

Uninitialized pointers

Categories of Security Problems

New solutions enabled by a compiler-based VM

Excessive s/w privilege

Application data secrecy

Detecting rootkits

Information flow

Incorrect security checks

DOS by resource exhaustion

Dynamic code insertion

Race conditions

…

Categories of Security Problems

Need application-level solutions

- Configuration errors
- Excessive admin privilege
- Cross-site scripting
- Network-level DOS
- Email worms
- Phishing attacks
- Spam
- …

Categories of Security Problems

Solved with a safe execution env.

Buffer overflows

Dangling pointers

Format string errors

Uninitialized pointers

Categories of Security Problems

New solutions enabled by a compiler-based VM

Excessive s/w privilege
Application data secrecy
Detecting rootkits
Information flow
Incorrect security checks
DOS by resource exhaustion
Dynamic code insertion
Race conditions
...

Categories of Security Problems

Solved with a safe execution env.

Buffer overflows

Dangling pointers

Format string errors

Uninitialized pointers

Categories of Security Problems

New solutions enabled by a compiler-based VM

Excessive s/w privilege

Application data secrecy

Detecting rootkits

Information flow

Incorrect security checks

DOS by resource exhaustion

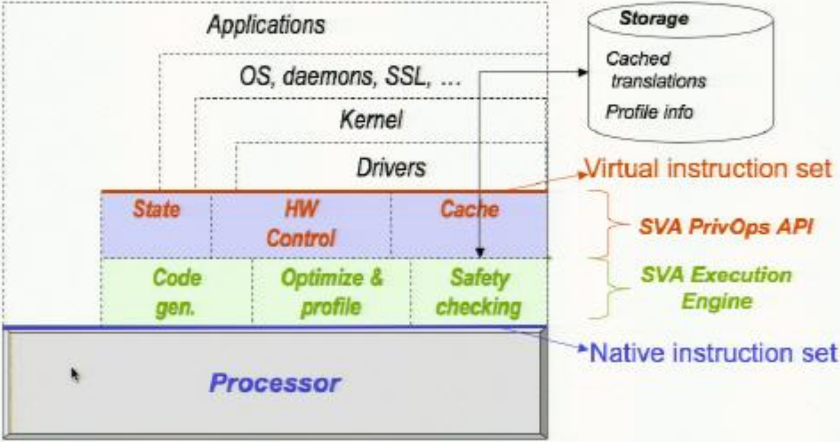
Dynamic code insertion

Race conditions

...

SVA: Secure Virtual Architecture

Designed to work with legacy kernels, e.g., Linux



Outline

SVA: Secure Virtual Architecture

- ☛SVA Overview
- ☛SVA Virtual Instruction Set and Compiler System
- ☛SVA Safety Guarantees
- ☛Future Work: Security Applications of SVA

Porting an OS to SVA

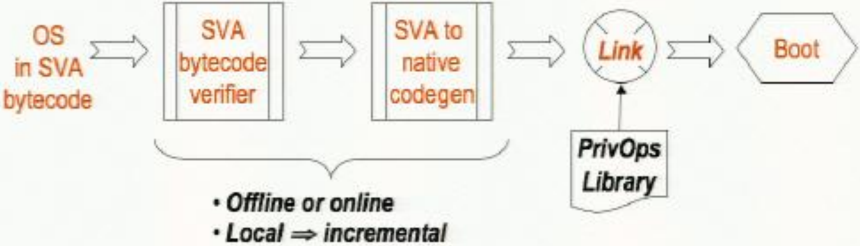
E.g., This is how we ported Linux 2.4.22

Existing OS in C/C++ ⇒

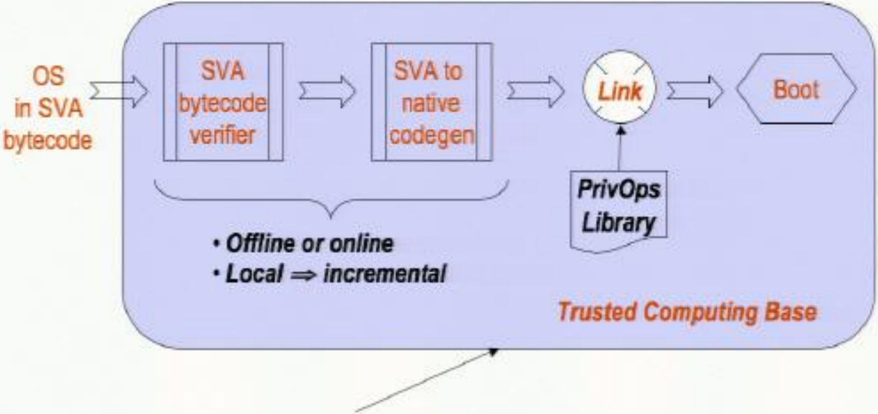
⇒ Safety-checking C/C++ compiler ⇒ OS in SVA bytecode

- *Port kernel to PrivOps API*
- *Port kernel allocators to checker API*

Running an OS on SVA



Running an OS on SVA



- TCB components are far simpler than the safety checking compiler
- Incremental ⇒ dynamic loading is easy

SVA Design Features

Comprehensive: All security-sensitive software can be protected

- Kernel, OS extensions, device drivers, daemons, SSL

‡*Capability exists but not yet used*

SVA Design Features

Comprehensive: All security-sensitive software can be protected

- Kernel, OS extensions, device drivers, daemons, SSL

Incontrovertible: Security criteria cannot be bypassed

- VM and OS can enforce safety policies at install time or load-time

‡Capability exists but not yet used

SVA Design Features

Comprehensive: All security-sensitive software can be protected

- Kernel, OS extensions, device drivers, daemons, SSL

Incontrovertible: Security criteria cannot be bypassed

- VM and OS can enforce safety policies at install time or load-time

Robust: Reduce trusted computing base (TCB):

- Complex safety-checking compiler is outside the TCB
- Extend to other security problems: Information flow, security automata, …

‡*Capability exists but not yet used*

SVA Design Features

Comprehensive: All security-sensitive software can be protected

- Kernel, OS extensions, device drivers, daemons, SSL

Incontrovertible: Security criteria cannot be bypassed

- VM and OS can enforce safety policies at install time or load-time

Robust: Reduce trusted computing base (TCB):

- Complex safety-checking compiler is outside the TCB
- Extend to other security problems: Information flow, security automata, …

Whole-system ‡: Security analysis, transforms across programs

- Can perform analysis, transformations *across application/kernel boundary*

‡Capability exists but not yet used

SVA Design Features

Comprehensive: All security-sensitive software can be protected

- Kernel, OS extensions, device drivers, daemons, SSL

Incontrovertible: Security criteria cannot be bypassed

- VM and OS can enforce safety policies at install time or load-time

Robust: Reduce trusted computing base (TCB):

- Complex safety-checking compiler is outside the TCB
- Extend to other security problems: Information flow, security automata, …

Whole-system ‡: Security analysis, transforms across programs

- Can perform analysis, transformations *across application/kernel boundary*

Hardware assisted ‡: Can exploit upcoming hardware support

- **TPM** (secure boot, secure PKI), **IOMMU** (secure DMA)

‡Capability exists but not yet used

SVA Prototype

SVA PrivOps Library

- C and i386 assembly code for Pentium 3
- Compiled to native code library ahead of time

Safety Principles

- Same safety guarantees for standalone programs and kernel
- Untrusted safety-checking compiler, trusted type checker

Linux 2.4.22 port to SVA

- Like a port to a new architecture
- Assembly code replaced with SVA operations
- ***kmem_cache_alloc***: type homogeneous (TH) pools
- ***kmalloc***, others: non-TH pools

Outline

SVA: The Secure Virtual Architecture Project

- SVA Overview
- SVA Virtual Instruction Set
- SVA Safety Principles
- Future Security Applications of SVA

SVA Virtual Instruction Set

Unprivileged operations:

- Derived from *IR* of the LLVM compiler system
- Extended with security annotations:
  - *Current:* type system for safety properties

Privileged operations (PrivOps) API:

- API for kernel-hardware interactions
- Mechanisms, not policy

LLVM Provides Compiler Foundation

Framework for "lifelong compilation"

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

Commercial-quality compiler infrastructure

- **JIT:** Apple (MacOS 10.5), Adobe, Hue AS
- **Static back end:** Cray, Ageia, Ascenium, Wind River
- **Silicon compilation:** AutoESL
- ***Unknown***: Aerospace, Microchip Tech, Wind River
- Also many academic, open-source, users; many contributors

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;
}
```

```
;; SVA 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
}
```

- Architecture-neutral
- Low-level operations
- SSA representation
- Typed
- Mid-level type info

[*CGO 2004, MICRO 2003*]

SVA Privileged Operations API

Hardware Control

Syscall, interrupt, trap handlers

- all OS entries are monitored

Page table entries

- so are page mapping events

I/O operations

- ditto

State Manipulation

Context-switching

- save/restore *native* state

Signal delivery

Interrupt Context

- exploit hardware for fast interrupts

[*WIOSCA 2006*]

Outline

SVA: The Secure Virtual Architecture Project

- SVA Overview
- SVA Virtual Instruction Set
- **SVA Safety Principles**
  - Guarantees
  - Standalone programs
  - Complete kernel
- **Future Security Applications of SVA**

Safety Guarantees

Strong Guarantees: Close to a safe language, but not equal

- **Memory safety:** No uninitialized pointer uses, no array overflows
- **Type safety** for a *subset* of objects
- **Control flow integrity:** only follow compiler-predicted paths
- **Sound operational semantics** $\Rightarrow$ sound static analysis

[TECS 05, PLDI 06, ICSE 06, DSN 06]

Safety Guarantees

Strong Guarantees: Close to a safe language, but not equal

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

Primary Weakness: By Design

- **Tolerate but not eliminate dangling pointer errors**: avoid need for GC
- *Tolerate ⇒ Do not invalidate the above guarantees!*
- Option: detect *all* dangling pointer uses: low overhead for some programs

[TECS 05, PLDI 06, ICSE 06, DSN 06]

Safety Guarantees

Strong Guarantees: Close to a safe language, but not equal

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

Primary Weakness: By Design

- **Tolerate but not eliminate dangling pointer errors**: avoid need for GC
- *Tolerate ⇒ Do not invalidate the above guarantees!*
- Option: detect *all* dangling pointer uses: low overhead for some programs

Practical

- Completely automatic, no wrappers, no GC
- Works for arbitrary C programs
- Very low overhead for C programs

[TECS 05, PLDI 06, ICSE 06, DSN 06]

Underlying Principles

Idea 1: Pool allocation ⇔ static points-to graph

- Generate "pool checks" on (some) loads, stores
- ***Enforces partial type safety, sound pointer analysis***

Idea 2: Exploit type-homogeneous pools

- ***Eliminate all checks except array-bounds checks***
- ***Dangling pointers harmless*** **(with careful object alignment)**

Idea 3: Efficient array bounds checks without metadata

- Jones & Kelly: Maintain lookup tables of allocated objects
- Use 1 table per pool: greatly reduces overhead
- ***Avoid metadata on pointers***

Evaluation: Run-time Overhead

Olden, *Ptrdist*, 3 system daemons [Full list in PLDI06]

Evaluation: Run-time Overhead

Olden, *Ptrdist*, 3 system daemons [Full list in PLDI06]

No source changes necessary

Detected all attacks from Zitser's suite.

Evaluation: Run-time Overhead

1.0 ≡ no pool allocation + no SAFECode passes

Olden, *Ptrdist*, 3 system daemons [Full list in PLDI06]

No source changes necessary

Detected all attacks from Zitser's suite.

Program	SAFECode: No array checks	SAFECode with array checks
bh	3%	3%
bisort	0	-4
em3d	27	91
treeadd	-1	0
tsp	-1	1
Yacr2	30	30
Ks	12	12
anagram	23	23
ftpd	0	4
fingerd	3	7
ghttpd	7	-10

Evaluation: Run-time Overhead

1.0 ≡ no pool allocation + no SAFECode passes

Olden, *Ptrdist*, 3 system daemons [Full list in PLDI06]

No source changes necessary

Detected all attacks from Zitser's suite.

Compare CCured (*Olden*, *Ptrdist*)

- **Much higher time overheads except em3d**
- **Much higher space overheads (1x-4x)**
- **Significant porting effort**

Program	SAFECode: No array checks	SAFECode with array checks
bh	3%	3%
bisort	0	-4
em3d	27	91
treeadd	-1	0
tsp	-1	1
Yacr2	30	30
Ks	12	12
anagram	23	23
ftpd	0	4
fingerd	3	7
ghttpd	7	-10

Safety Challenges in the OS

Kernels have many custom pool allocators

- *Retain all existing kernel allocators*
- Define clean interface between allocator and checker: needs manual port
- Compiler: Infer mapping of pools to points-to graph nodes

Kernels have many external entry points bringing in pointers

- Can still use array indexing checks for externally-accessible pools

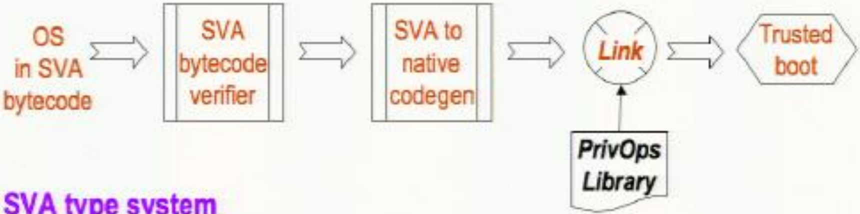
Kernel allocators not initialized early in boot sequence

- Reserve memory to use for metadata for early objects

Extensive use of non-type-safe code

- More aggressive static bounds-checking, run-time tuning

Verifying Security of SVA Bytecode



SVA type system

- Stack of "regions" (logical pools); every object registered in a pool
- Region annotation on every pointer

Bytecode Verifier

- Simple local, type checker
- Inserts run-time checks
- Local ⇒ easy to support dynamically loaded modules

This strategy can be used for any type system, e.g., security automata [Walker, POPL2000], information flow [Myers, POPL99]

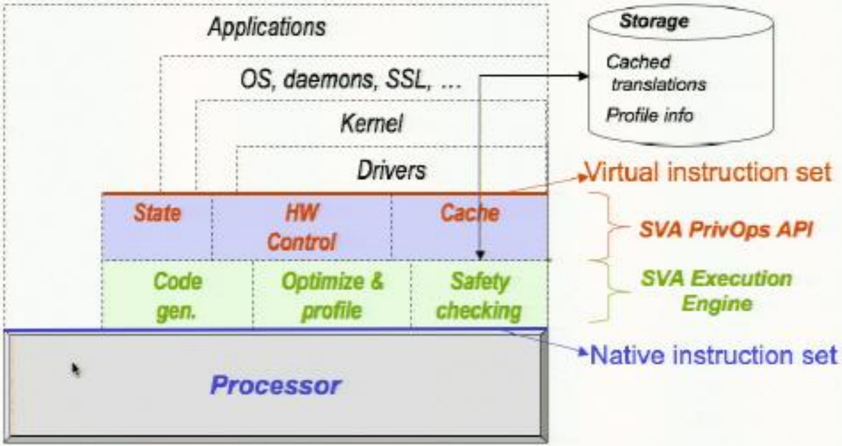
Outline

SVA: The Secure Virtual Architecture Project

- SVA Overview
- SVA Virtual Instruction Set
- SVA Safety Principles
- **Future Security Applications of SVA**
  - Application data secrecy
  - Reducing privilege
  - Monitoring kernel-mode rootkits

SVA: Secure Virtual Architecture

Compiler + privileged run-time + type checker



Some Security Applications of SVA

Minimizing Privilege for Root Programs

- Compiler ⇒ Privilege bracketing + secure system call dispatch
- Compiler ⇒ "Authenticated system call" policies
- *Incontrovertible*: transformations at install-time or load-time

Protecting Application Data From OS

- Higher privilege VM ⇒ Private physical memory
- Higher privilege VM ⇒ Monitor all page mappings, I/O operations

Monitoring Kernel-mode Rootkits (with Sam King)

- Higher privilege VM ⇒ monitoring of kernel cannot be subverted
- Compiler ⇒ fine-grain monitoring of kernel operations

Summary

SVA: Secure Virtual Architecture

Safe environment for programs

- Fully automatic

Safe environment for entire OS

- Low porting effort

Higher-level security capabilities

- Wide-open research area
- Collaborations welcome!

Solved with a safe execution env.

Buffer overflows

Dangling pointers

Format string errors

Uninitialized pointers

Questions?

Summary

SVA: Secure Virtual Architecture

Safe environment for programs

- Fully automatic

Safe environment for entire OS

- Low porting effort

Higher-level security capabilities

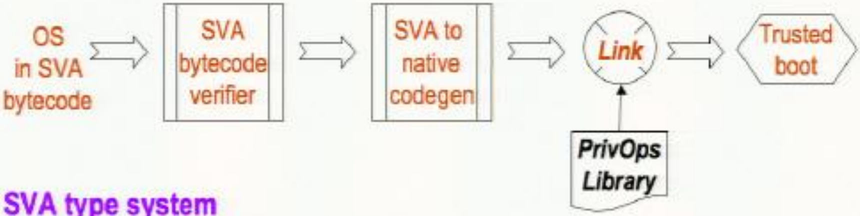
- Wide-open research area
- Collaborations welcome!

New solutions enabled by a compiler-based VM

Excessive s/w privilege
Application data secrecy
Detecting rootkits
Information flow
Incorrect security checks
DOS by resource exhaustion
Dynamic code insertion
Race conditions
...

Questions?

Verifying Security of SVA Bytecode



SVA type system

- Stack of "regions" (logical pools); every object registered in a pool
- Region annotation on every pointer

Bytecode Verifier

- Simple local, type checker
- Inserts run-time checks
- Local ⇒ easy to support dynamically loaded modules

This strategy can be used for any type system, e.g., security automata [Walker, POPL2000], information flow [Myers, POPL99]

Summary

SVA: Secure Virtual Architecture

Safe environment for programs

- Fully automatic

Safe environment for entire OS

- Low porting effort

Higher-level security capabilities

- Wide-open research area
- Collaborations welcome!

New solutions enabled by a compiler-based VM

Excessive s/w privilege
Application data secrecy
Detecting rootkits
Information flow
Incorrect security checks
DOS by resource exhaustion
Dynamic code insertion
Race conditions
...

Questions?