

# In-lined Reference Monitoring (IRM)

CS 6V81-002: Language-based  
Security

February 25, 2008

# Introduction

- RM Definition: “A *reference monitor* observes execution of a *target system* and halts that system whenever it is about to violate some security policy of concern.”
- For example:
  - An OS can monitor access to files, I/O devices, etc.
  - Control transfers when system calls are invoked (An OS *or* virtual machine can act as an RM here)

# Introduction

- But those examples are external to the executing program
  - Performance cost: Untrusted program and RM run in separate address spaces (Context switches needed)
  - Expressiveness cost: Limitations on policy scope
    - What if we care about more than just system calls?

# Introduction

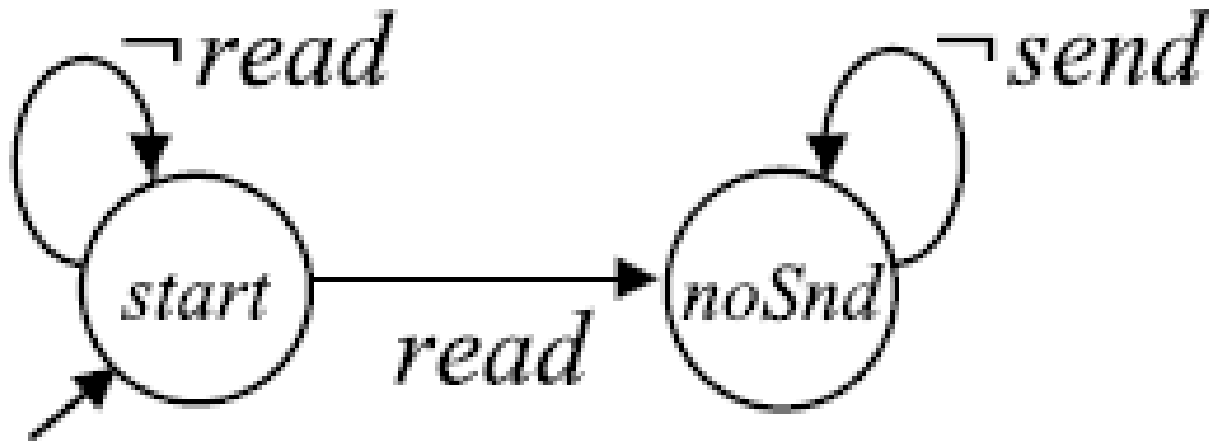
- A solution: *In-lined* Reference Monitors
- Inject the RM directly into the untrusted program
  - Performance cost is minimal (no context switches)
  - Policy scope is limited only by the ingenuity of the rewriting program
- SFI is a kind of IRM that specifically handles memory access policies
- **SASI** – “**S**ecurity **A**utomata **S**F**I** Implementation”

# Security Automata

- Models security policies
- Example:
  - “No messages may be sent after reading a file.”
- Three components:
  - Set of possible states
  - Input alphabet
    - read file, send message, etc.
  - Transition relation
    - Given a current state and an input symbol, to what new state does the automaton move?

# Security Automata

- Example automaton:
  - “No messages may be sent after reading a file.”



# Security Automata

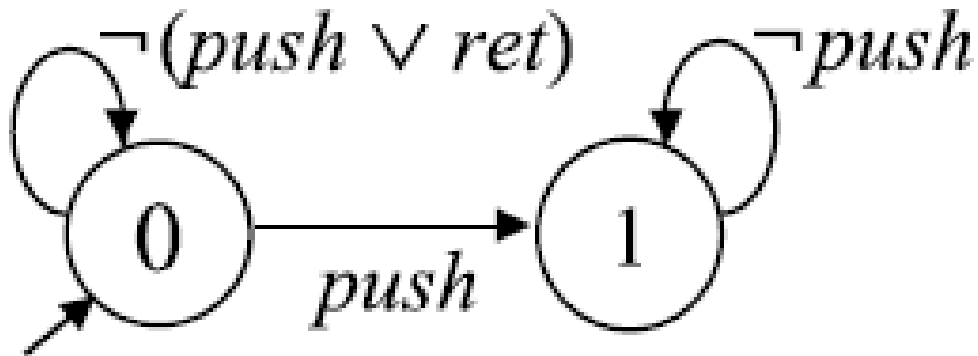
- Traditional approach:
  - System is OK as long as there are transitions available for the given inputs
  - If there is no transition for a given input, that corresponds to a policy violation
  - If an illegal operation occurs, halt the target system
- Alternatives (Not discussed in the paper):
  - Have a specific “bad state” (moving there means there is a policy violation)
    - So non-existing transitions mean nothing
  - Don’t actually *halt* system, but take some other action

# Merging-in an Automaton

- Strategically insert checks into the target code
- SASI uses four phases:
  1. Insert security automata
    - Place a copy of the automaton before each instruction
  2. Evaluate transitions
    - For each instruction's preceding copy of the automaton, evaluate all possible transitions to *true* or *false*
  3. Simplify automata
    - Eliminate all transitions that evaluated to *false*
  4. Compile automata
    - Translate resulting automata into code for injection into target program

# Merging-in an Automaton

- Apply this automaton...
  - “Push once, and only once, before returning.”

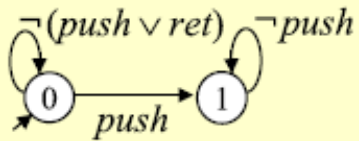


- ...to this code:

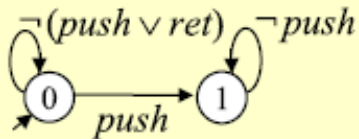
```
mul r1, r0, r0
push r1
ret
```

# Merging-in an Automaton

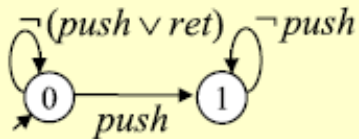
Insert security automata



**mul r1,r0,r0**

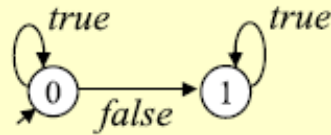


**push r1**

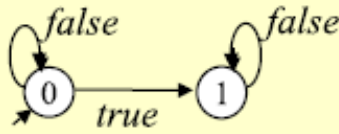


**ret**

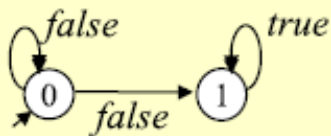
Evaluate transitions



**mul r1,r0,r0**

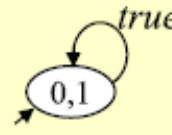


**push r1**

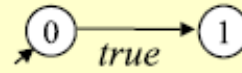


**ret**

Simplify automata



**mul r1,r0,r0**



**push r1**



**ret**

Compile automata

```
mul r1,r0,r0
if state==0
then state:=1
else ABORT
push r1
```

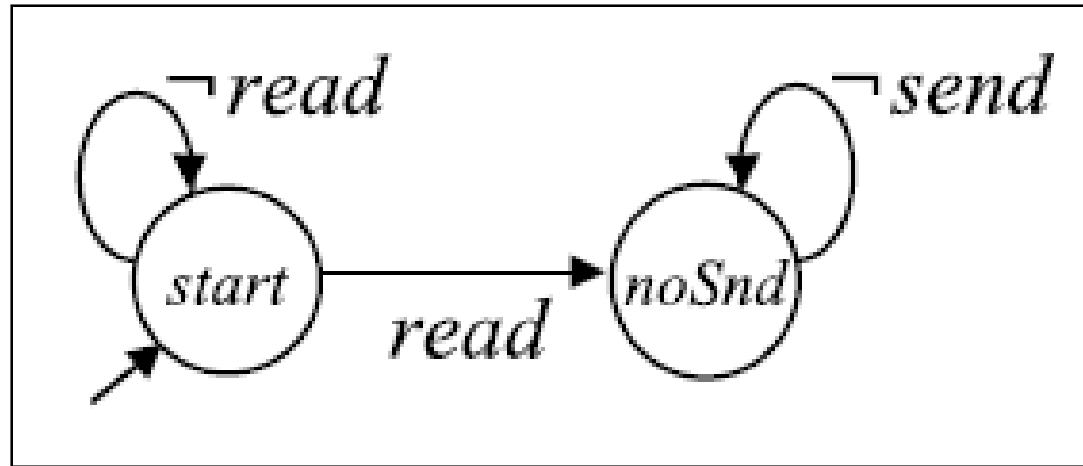
```
if state==0
then ABORT
ret
```

# SAL

- “**S**ecurity **A**utomaton **L**anguage”
  - A policy specification language for SASI
  - Models the security automaton graph
- Different versions for x86 and JVMML implementations
  - The paper gives an example for JVMML

# SAL

```
/* Macros */
MethodCall(name) ::= op=="invokevirtual" && param[1]==name;
FileRead() ::= MethodCall("java/io/FileInputStream/read()I");
Send() ::= MethodCall("java/net/SocketOutputStream/write(I)V");
/*
** The Security Automaton
*/
start ::=
    !FileRead() -> start
    FileRead() -> hasRead
;
hasRead ::=
    !Send() -> hasRead
;
```



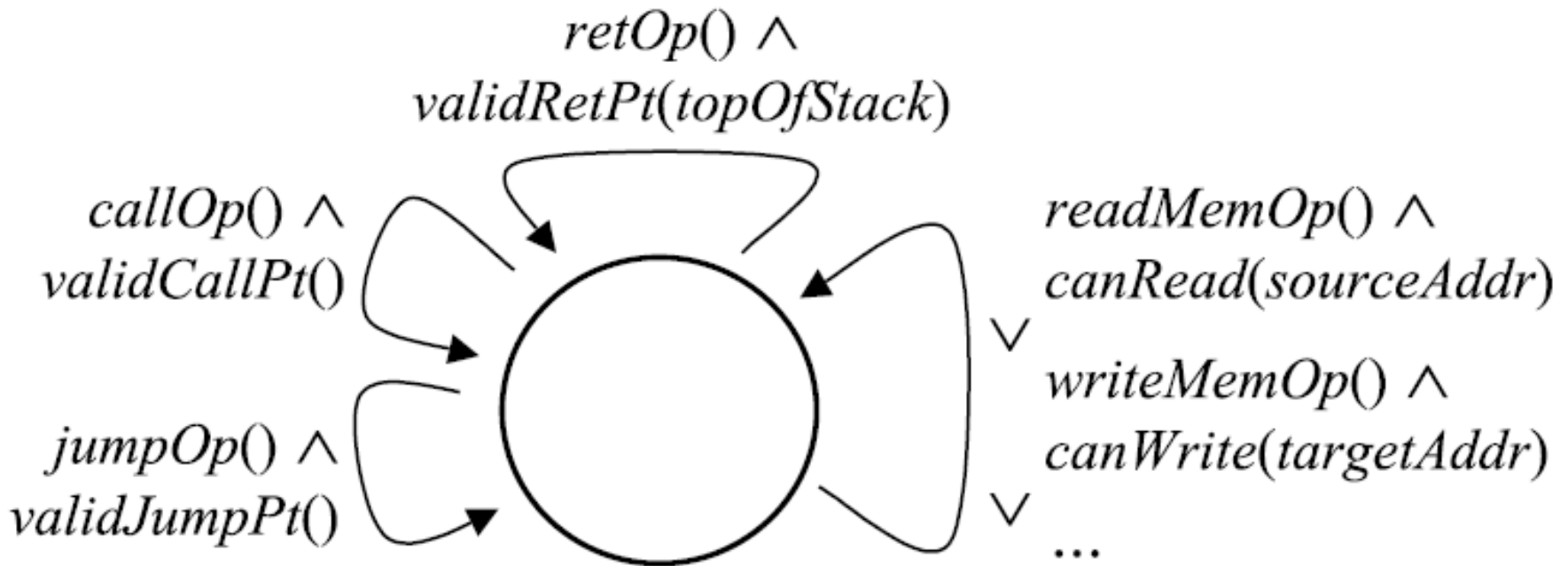
# SAL

- For x86:
  - Evaluates opcode and operands of instructions
  - Defines address ranges for reads and writes
  - Defines addresses that can be used by branches, calls, and returns
- For JVMIL
  - Evaluates class name, method name, method type signature, opcode, instruction operands, and JVM state in which an instruction will be executed

# x86 Prototype

- Designed to work with gcc assembly output
  - Allows for the following assumptions:
    - Maintains particular register-usage conventions
    - Isn't self-modifying
    - Program behavior is not altered by “stutter-steps” like nops
    - Variables and branch-targets can only use labels assigned by gcc at compile time
  - Basically, it makes it easier to protect the IRM from being circumvented
- Using these assumptions, SASI prepends a SAL description of the following automaton...

# x86 Prototype



- Provides SFI-esque protection
  - All jumps, calls, returns, and memory reads and writes must be legal

# x86 Prototype

- Example: Copying `%edx` contents to `dirty` array position designated by `%eax`

```
pushl    %ebx
leal    dirty(,%eax,4), %ebx
andl    segmentMask, %ebx
cmpl    writeSegment, %ebx
jne     SASIx86_FAIL
popl    %ebx
movl    %edx, dirty(,%eax,4)
```

SASI x86 SFI

# JVML Prototype

- JVML itself provides some guarantees:
  - Implies the SFI-esque policy from the earlier slide
  - Type safety!
- JVML also provides lots of assistance:
  - Prevents jumps to unlabeled instructions
    - SASI's injected automaton code does not contain labels
  - Provides information about classes, objects, methods, threads, and types
    - Allows for the use of more abstract, application-specific policies (think object-oriented)

# JVML Prototype

- “No messages sent after reading a file”

```
...
ldc 1 ; noSnd state number
putstatic SASIJVML/state ; change state to noSnd
invokevirtual java/io/FileInputStream/read()I ; read file
...
getstatic SASIJVML/state ; get current state number
ifeq SUCCEED ; if start = state goto SUCCEED
    invokestatic SASIJVML/FAIL()V ; else violation
SUCCEED:
invokevirtual java/net/SocketOutputStream/write(I)V ; send msg
```

# SASI in Retrospect

- Strengths:
  - Broader scope than SFI
  - Security automata allow for complex policies
  - Changes do not affect program correctness
- Weaknesses:
  - Some possibly questionable expansions of TCB:
    - x86 SASI's reliance on gcc conventions
    - JVMML SASI's reliance on JVMML itself
  - SAL becomes awkward for some kinds of policies
    - e.g., tracking use of a particular variable
    - x86 assembly lacks explicit “function” and “object” semantics
  - What about reflection?

# SASI in Retrospect

- Possible suggested improvements:
  - Using a policy specification language which tracks specific typed variables
  - Using TAL or some other variant of x86 assembly
  - Altering SASI to modify high-level code (e.g, C++) instead of object code
    - Greatly increases the TCB, and thus unattractive
- Second paper talks about a much more complex policy language: PSLang

# References

Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proceedings of the New Security Paradigms Workshop*, Caledon Hills, ON, September 1999.

# Discussion Questions

- How flexible is the security automaton policy model?
  - What are some interesting policy scenarios?
  - How could the model be improved?
- What's the best way to tackle x86 rewriting?
  - How do we handle types, functions, objects?
  - Switch focus to TAL, PCC?
- Is there any way to deal with reflection?