Program Monitoring

Based on Enforcing Security through Execution Monitoring
Úlfar Erlingsson, 2004 Summer School on Software Security

2004 SIGPL 여름학교
2004. 08. 13
부산대학교

창 병모
숙명여대 컴퓨터과학과

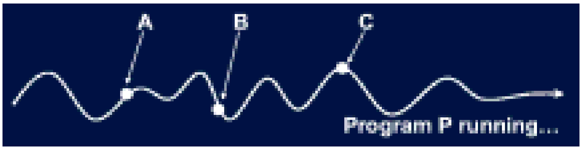
Outline

- Execution Monitoring
- Security Policy as Security Automata
- Inlined Reference Monitors
- Further works

Execution Monitoring

Execution Monitoring

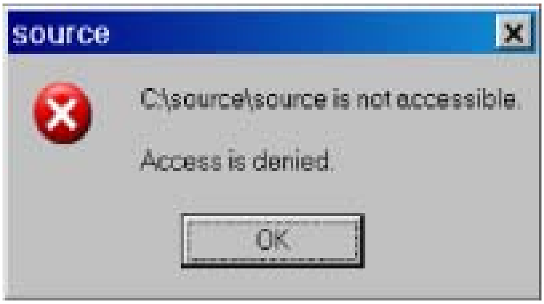
- Observe program execution
  - Look at a program's execution on a given input as a sequence of runtime events
    (e.g., the A, B, and C below)
  - Possibly do "something" on each event



What is EM good for?

- Debugging, tracing, breakpoints, etc.
- Auditing and Logging
- Software testing
  - memory leaks,
  - out-of-bounds array accesses,
  - race conditions, atomicity, etc.
- Security
  - buffer overflow prevention etc.

In particular

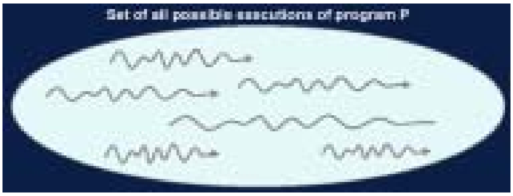


C:\source\source is not accessible.

Access is denied.

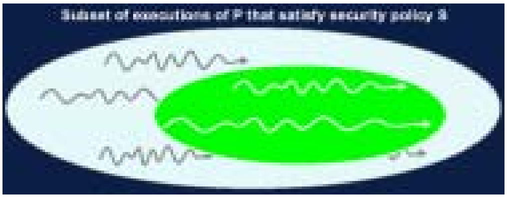
Program as Sets of Execution Traces

- View a program as defining an (infinite) set of (possibly infinite) execution traces
- All executions on all possible inputs



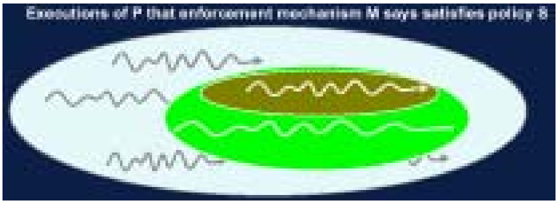
Security Policies as Traces

- Define security policies as a subset of possible program execution traces
- Security policy set defines a predicate S



Enforcing Security Policies

- Allows some traces that satisfy security policy
- Enforcement mechanism M is a concrete implementation that defines a subset of S

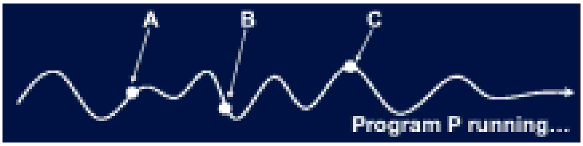

Executions of P that enforcement mechanism M says satisfies policy S

Desirable Security Mechanisms

- Don’t want enforcement to be vacuous
  (e.g. defining the empty set or disallowing all)
- Want enforcement to be exact (M == S)

Execution Monitoring

- Focusing on one Execution Trace
- Easy to do (just observe and constrain)
- EM can often approximate desired policy
- EM closely related to safety properties



EM Security Policies [Schneider00]

- Define acceptable/unacceptable execution
  - EM observes execution (and truncates it)
  - EM-enforceable part of safety properties

- Safety property
  - access control
  - integrity

- Not Safety Property
  - information flow
  - liveness
  - availability

Characteristics of EM

- EM enforcement mechanism
  - Analyzes the single (current) execution
  - Must truncate execution as soon as prefix violates policy
  - Must detect violations after a finite time

- EM Enforceable policy implies safety property
  - EM ⊂ safety properties

- Why EM ≠ Safety Properties
  - EM can only use bounded memory
  - Safety properties can use infinite state

What EM can & can’t do

- EM *can* do access control
  - DAC, MAC, MLS, …

- EM can’t do information flow
  - InfoFlow depends on other traces

- EM can’t do Liveness/Availability

Security Policy as Security Automata

Specifying Security Policies

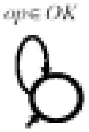
One way is as **Security Automata**

- Formalism expresses the right properties
  - SA $\equiv$ safety properties $\supset$ EM-enforceable
- Simple to specify, interpret, and compile
- Good for analysis, emulation, testing

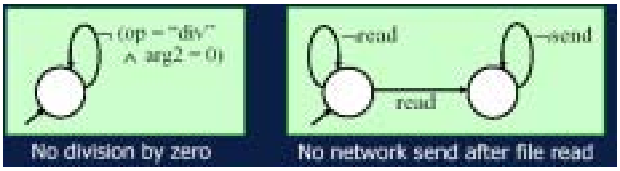
Security Automata

- Buchi Automata with all states accepting
- Fail if no transition is possible
- Can accept infinite inputs

- Simple Example: simple access control



Examples



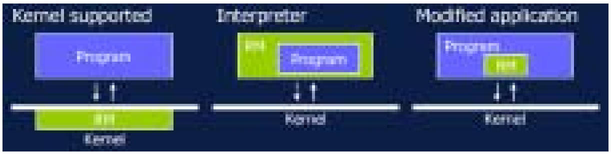
No division by zero

No network send after file read

Inlined Reference Monitor

Reference Monitors [Anderson72]

- Execution monitor that forwards events to security-policy-specific validity checks
- Implementing RMs
  - Capture *all* policy-relevant events
  - Protect RM from subversion

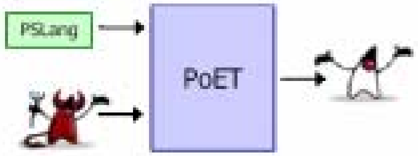


Validity Checks

- Triggered by RM on each event
- Encodes the security policy
- Perform arbitrary computation to decide whether to allow event or halt
  - Can have side effects? (Not if EM)
  - Can change program flow? (Not if EM)

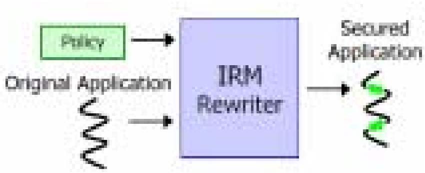
Inlined Reference Monitors
[Erlingsson Schneider 99]

- IDEA: Use 3rd type of RM implementations
  - Use Security Automata to specify security policy
  - Policy specifies both RM and Validity Checks
  - Permanently embed security into application



IRM Implementation

- Implement RMs by program modification



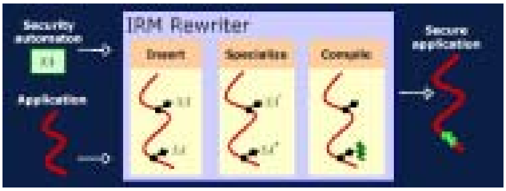
- IRMs have access to program abstractions
  - Capture all potentially security-relevant events
  - Rewriter works on machine language programs
- Issues
  - How to capture all relevant events
  - Prevent application subverting inserted RM
  - Preserve application behavior

IRM Enforcement Advantages

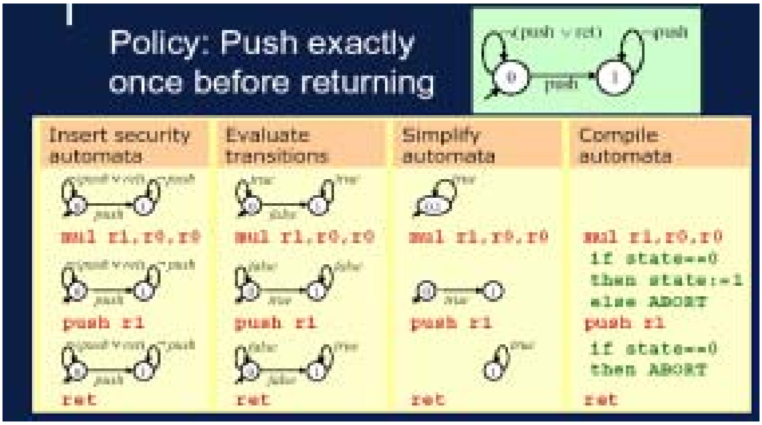
- Can enforce policies on application abstractions
  - E.g., Restrict MSWord macros and documents
- Mechanism is simple and efficient
  - Rewrites machine code
  - Kernel is unaware of security enforcement
  - No enforcement overhead from context switches

Efficient IRM Enforcement

- Evaluate SA policy at every point in program
- Often no need to check at a machine instruction
  - "No div zero": Only check before "div" instructions
- Simplify SA by partial evaluation
  - Insert security policy checking code before every instruction
  - Use static knowledge of insertion point to simplify the check



Example IRM Rewriting


Policy: Push exactly once before returning

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
```

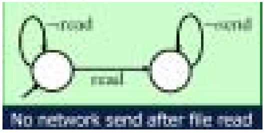
Example:
No message sent after reading a file

- SAL specification for “No messages sent after reading a file.”

```
/* Macro definitions */
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() -> noSnd
;
noSnd ::=
     !Send() -> noSnd
;
```

Security Automata



No network send after file read

- JVML enforcement of “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 state = start goto SUCCEED
invokestatic SASIJVML/ABORT()V                         ; else violation
SUCCEED:
invokevirtual java/net/SocketOutputStream/write(I)V    ; send msg
...
```

Further Works

- Monitoring Machine Code Execution
  - Software Fault Isolation
  - Buffer Overflows and Mitigations
- Advanced IRMs
  - Low-level Actions
  - Event Synthesis
  - Static Analysis