

# Run-time Program Monitors: Theory and **Practice**

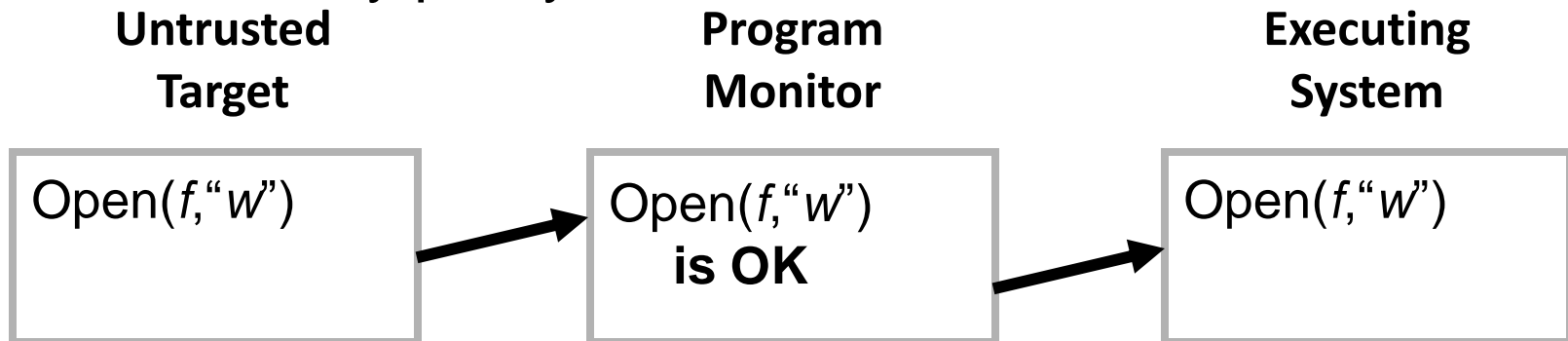
Lujo Bauer

18-732

Fall 2010

# Program Monitors

- Software often behaves unexpectedly
  - Bugs
  - Malicious design (malware)
- Run-time program monitors
  - Ensure that software dynamically adheres to a security policy



# Policies and Properties

- Policies:

$$\mathcal{P}(\Sigma)$$

- Properties:

$$\mathcal{P}(\Sigma) : (\forall \sigma \in \Sigma : \hat{\mathcal{P}}(\sigma))$$

where  $\sigma$  is a (trace of a) single execution and  $\Sigma$  is the set of executions that define a program

- Practical monitors implement/enforce  $\hat{\mathcal{P}}$  on  $\sigma$

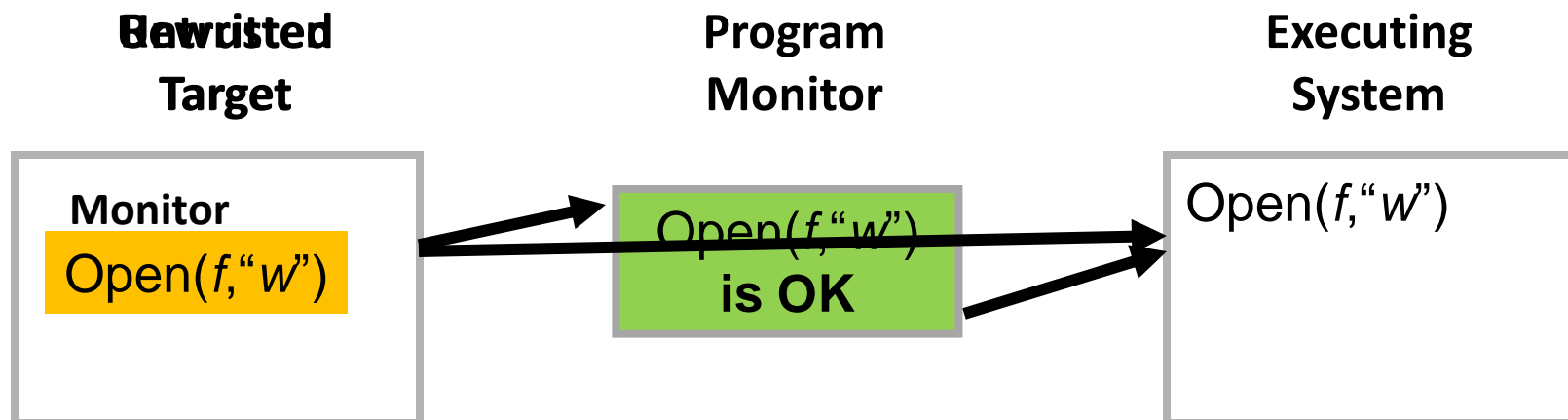
# Common Monitor Examples

- File access control
- Firewalls
- Stack inspection
- Bounds checks on input values
- Security logging
- Displaying security warnings
- Operating systems and virtual machines
- ...

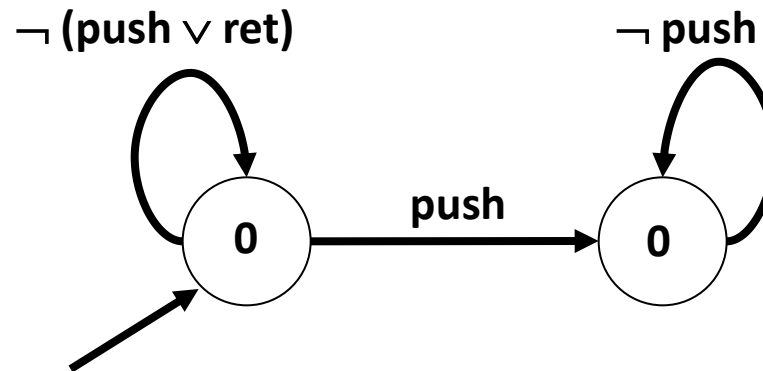
# Security Automata SFI Implementation

[Erlingsson & Schneider '99]

- Monitor is specified as a security automaton
- Monitor is inlined into code
  - Checks whether operation is allowed
- Java & C implementations
- Operates on object code
  - Java bytecode, x86 assembly language



# Inlining an Automaton



# Inlining an Automaton

Insert security  
automata

Evaluate  
transitions

Simplify  
automata

Compile  
automata

(Figure from Erlingsson & Schneider, SASI Enforcement of Security Policies, 1999.)

# Specifying SASI Policies

- Specification tied to object code
  - Java is high level
    - `new java.net.URLConnection`
    - Application-level abstractions
  - C (x86 assembly) is low level
    - `mov ... , jne ...`
    - Must create own abstractions

# SASI Policies: Java

```
MethodCall(name) ::= op=="invokevirtual" && param[1]==name  
FileRead() ::= MethodCall("java/io/FileInputStream/read()I");  
Send() ::= MethodCall("java/net/SocketOutputStream/write(I)V");
```

...

```
start ::=  
    !FileRead() -> start  
    FileRead() -> hasRead  
;  
hasRead ::=  
    !Send() -> hasRead  
;
```

# Anti-circumvention Policies

- Monitor must enforce *complete mediation*
  - Must prevent *all* ways of reaching “bad” action
- Must prevent target from:
  - modifying security automaton state
  - circumventing automaton code
  - modifying its own code or jumping to code other than its own
- Can be enforced using SASI itself
- Observation: In Java much of this comes for free!

# SASI Retrospective

- Automata make a poor specification language
  - E.g., implementing counters
- Acceptable performance
  - Run-time overhead is low
  - Not quite as good as special-purpose mechanisms, but close
- x86 assembly poor for specifying policies
  - Very expressive
  - Far from application-level abstractions

# Sources

- Ú. Erlingsson & F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, 1999.