

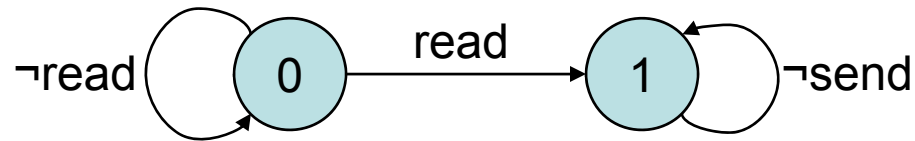
# Certified In-lined Reference Monitoring

CS6v81: Language-based Security

Spring 2008

# Last Week: IRM's

- History-based security policies
  - Example: No network-sends after file-reads
  - Security automaton:



- Implementation:

```
if state==0 then state:=1  
else if state==1 then state:=1;  
call System_read;
```

```
if state==0 then state:=0  
else if state==1 then halt;  
call System_send;
```

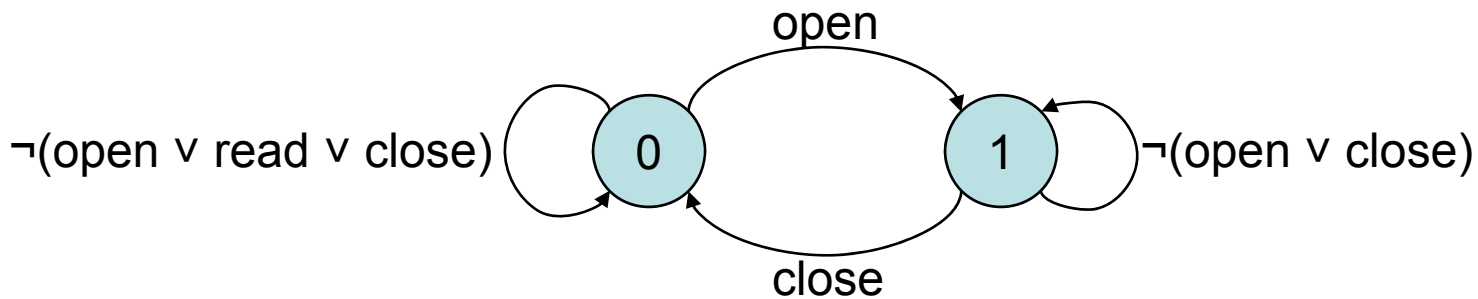
- Protect **state** from tampering (SFI), make **guard instructions** non-circumventable (CFI), etc.

# Two New Challenges

- Challenge #1: Per-object policies
  - Ex: Every file must be opened before read
- Main Challenge: Certifying IRM's
  - TAL approach: compiler produces typed assembly
  - Mobile approach: rewriter produces typed .NET bytecode
  - Type system proves *policy-adherence* (not just memory and control-flow safety)

# Challenge #1: Per-Object Policies

- Policy: Each file must be opened before read or closed.
- First implementation attempt:



```
if state==0 then state:=1  
else if state==1 then halt;  
call System_fopen;
```

```
if state==0 then halt  
else if state==1 then state:=0;  
call System_fclose;
```

```
if state==0 then halt;  
call System_fread;
```

# Challenge #1: Per-Object Policies

- Need separate **state** variable for *each* File object
- ...but File objects are created and destroyed dynamically at runtime!
- ...and there might be an unbounded number of them!
- Need to create and destroy security state variables on the fly

# Solution: Packages

- Package = two-field object

```
class Package {  
    void *obj;  
    int state;  
}
```

- Two operations: pack and unpack

```
void Pack(Package *p, void *o, int s) {  
    p->obj = o;  
    p->state = s;  
}
```

```
void *Unpack(Package *p, int *s) {  
    void *obj = p->obj;  
    p->obj = null;  
    *s = p->state;  
    return obj;  
}
```

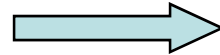
# Rewriting With Packages

```
File *f = new File(...);
```



```
Package *p = new Package();  
Pack(p, new File(...), 0);
```

```
myfunction(..., f, ...);
```



```
myfunction(..., p, ...);
```

```
fopen(f, ...);
```



```
int state;  
File *f = Unpack(p, &state);  
if state==0 then state=1;  
else if state==1 then halt;  
fopen(f, ...);  
Pack(p, f, state);
```

# Are Packages Practical?

- Memory overhead
  - adds 8 bytes per security-relevant object
- Runtime efficiency
  - 1 method call becomes 3 method calls
  - worst case: ~3x overhead
  - optimization strategy: lift pack/unpack out of loops
- TCB Bloat
  - pack/unpack is *very small* trusted library
  - can reuse exact same library for every application

# Global State

- If we implement packages, we get global-state IRM policies for free!
  - Create a `global_package` at program start
  - `Unpack/repack` it at each security-relevant operation
  - Optimization: leave it unpacked throughout execution
- Conclusion: Packages *generalize* the traditional IRM approach by allowing dynamic creation of state variables.

# Challenge #2: Shrink the TCB

- No proof of correctness for IRM's
  - Entire rewriter is trusted
  - Rewriter becomes huge when we start to add optimizing phases to the analysis
- How to prove *policy-adherence* for per-object, history-based policies...?
- Two possible approaches:
  - PCC approach: express policies as logical predicates
  - TAL approach: design a type system that can express security automata

# The Big Idea

- IRM Rewriter yields ***typed*** code (like TAL) instead of plain code
- Well-typedness can be verified by a (relatively) small ***type-checker***
- ***Soundness theorem***: Well-typed code is always policy adherent
- Typing annotations can be ***erased*** after type-checking, yielding an executable binary
- Note: We only guarantee policy-adherence, ***NOT*** that the rewriter is behavior-preserving

# Design Constraints

- No change to .NET VM
  - IRM types are refinements of existing .NET types
  - IRM types implemented as .NET XML annotations
- Must allow rewriter optimizations
  - Inefficient to require a specific instruction sequence surrounding every security-relevant event
- Support (most) “real-world” .NET features
  - dynamic memory allocation, exceptions, concurrency, finalizers, non-termination, ...
  - only Reflection not supported (yet)

# Mobile-0

- Introduce “effect types” (class + security state):  
 $\tau ::= \dots \mid C\langle\text{state}\rangle$
- Security-relevant operations have restrictive types

Read: File<open>  $\rightarrow$  string

- Type-checking example:

File *f = new File(...);	{ f : File<closed> }
f.Open();	{ f : File<open> }
string s = f.Read(...);	{ f : File<open> }
f.Close();	{ f : File<closed> }
s = f.Read(...);	{ does not type-check }

- Problem: Does not support join points well

# Join points

- Join points – nodes in the control flow graph where two flows merge
- Example:

f.Open();	{ f : File<open> }
if (x=0) f.Close();	{ f : File<??> }

# Mobile-1

- Use regular expressions as security states:

$$\tau ::= \dots \mid C\langle H \rangle$$
$$H ::= \varepsilon \mid e \mid H_1 H_2 \mid H_1 \cup H_2 \mid H^\omega$$
$$\text{Read: File}\langle e_{\text{open}} (e_{\text{read}})^\omega \rangle \rightarrow \text{string}$$

- Type-checking decides regexp subset:

File *f = new File(...);	{ f : File< $\varepsilon$ > }
f.Open();	{ f : File< $e_{\text{open}}$ > }
while(...) f.Read(...);	{ f : File< $e_{\text{open}} e_{\text{read}}^\omega$ > }
f.Close();	{ f : File< $e_{\text{open}} e_{\text{read}}^\omega e_{\text{close}}$ > }
s = f.Read(...);	{ does not type-check }

- Problem: Aliasing

# Aliasing Problem

- To infer correct history types  $C\langle H \rangle$ , type-checker must decide aliasing

- Example:

```
File *f1 = new File(...); { f1 : File< $\epsilon$ > }  
File *f2 = f1;           { f1 : File< $\epsilon$ >, f2 : File< $\epsilon$ > }  
f1.Open();              { f1 : File< $e_{\text{open}}$ >, f2 : File<??> }
```

- Aliasing is (in general) undecidable

# Mobile-2

- Simple aliasing occurs very frequently:
  - Aliasing between local variables (registers)
  - Can support it with no runtime overhead
  - Add a level of indirection to type system

$$\tau ::= \dots \mid C\langle \ell \rangle$$
$$\Psi : \ell \rightarrow H$$
$$H ::= \varepsilon \mid e \mid H_1 H_2 \mid H_1 \cup H_2 \mid H^\omega$$

- Example:

File \*f1 = new File(...);    { f1 : File< $\ell_1$ > } {  $\ell_1 = \varepsilon$  }

File \*f2 = f1;                    { f1 : File< $\ell_1$ >, f2 : File< $\ell_1$ > } {  $\ell_1 = \varepsilon$  }

f1.Open();                        { f1 : File< $\ell_1$ >, f2 : File< $\ell_1$ > } {  $\ell_1 = e_{\text{open}}$  }

# Heap Aliasing

- Aliases that leak to the heap are not statically trackable
  - Example: `a[3]=f { f:File<eopen>, a:File<?> array }`
- Solution: No unpacked objects may leak
  - Before leaking a security-relevant object to the heap, pack it
  - Once packed, no way to get the object back except by unpacking its package
  - Pack/unpack implemented as “swap” operations, preventing untracked aliases
  - Example: `a[3]=p { p:Package, a:Package array }`

# Type-checking Pack/Unpack

$o : \text{class File}\langle \ell \rangle, \{(\ell = \text{open})\}$

$s : \text{State}\langle \text{open read}^\omega \rangle$

$p := \text{pack}(o, s)$

$\text{open} \subseteq \text{open read}^\omega ?$

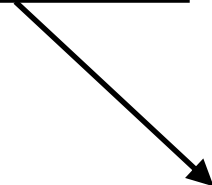
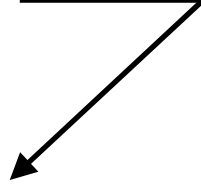
$p : \text{package File}, \{\}$

# Type-checking Pack/Unpack

$p : \text{package File}, \{\}$



$o2 := \text{unpack}(p)$



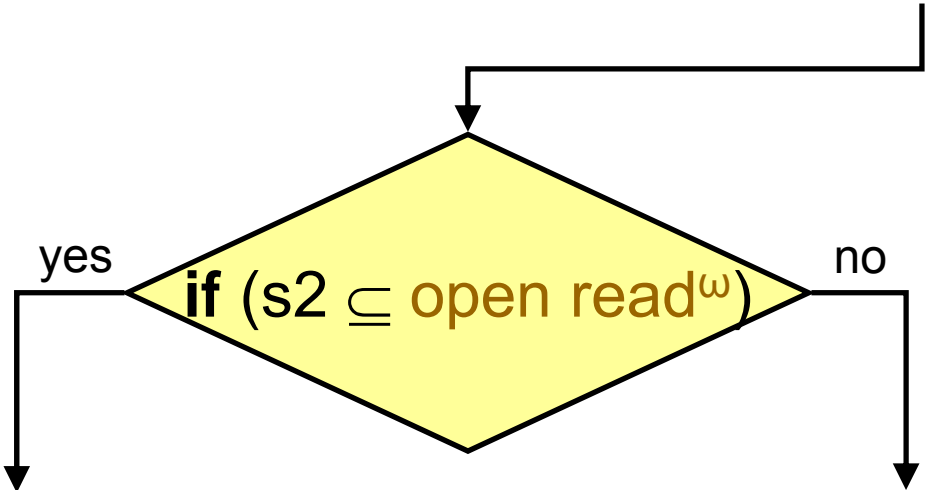
$o2 : \text{class File} \langle \ell_2 \rangle, \{(\ell_2 \rightarrow \theta)\}$

$s2 : \text{State} \langle \theta \rangle, \{(\ell_2 \rightarrow \theta)\}$

**o2 := unpack(p)**

**o2 : class File** $\langle \ell_2 \rangle, \{(\ell_2 \rightarrow \theta)\}$

**s2 : State** $\langle \theta \rangle, \{(\ell_2 \rightarrow \theta)\}$



$\{(\ell_2 \rightarrow \theta \cap \text{open read}^\omega)\}$

$\{(\ell_2 \rightarrow \theta \cap \text{open read}^\omega)\}$

# Concurrency!

- Implement pack and unpack atomically
  - Pack is like lock-acquire
  - Unpack is like lock-release
- Packages can alias, so they can be shared between threads
- Unpacked objects cannot alias across threads
  - Only one thread has an unpacked version of the object at any given time
- This suffices to prove policy-adherence for multi-threaded applications!
- Note: Does not prevent deadlock; only prevents policy violations in multi-threaded code