Current Techniques in Language-based Security

Steve Zdancewic
University of Pennsylvania
Mobile Code

- Modern languages like Java and C# have been designed for Internet applications and extensible systems

  applet  applet  applet

  web browser

  operating system

- PDAs, Cell Phones, Smart Cards, …
Applet Security Problems

- Protect OS & other valuable resources.
- Applets should not:
  - crash browser or OS
  - execute “rm –rf /”
  - be able to exhaust resources
- Applets should:
  - be able to access *some* system resources (e.g. to display a picture)
  - be isolated from each other

- Principles of least privileges and complete mediation apply
Java and C# Security

- Static Type Systems
  - Memory safety and jump safety
- Run-time checks for
  - Array index bounds
  - Downcasts
  - Access controls
- Virtual Machine / JIT compilation
  - Bytecode verification
  - Enforces encapsulation boundaries (e.g. private field)
- Garbage Collected
  - Eliminates memory management errors
- Library support
  - Cryptography, authentication, ...

These lectures
Access Control for Applets

- What level of granularity?
  - Applets can touch some parts of the file system but not others
  - Applets can make network connections to some locations but not others
- Different code has different levels of trustworthiness
  - www.l33t-hax0rs.com vs. www.java.sun.com
- Trusted code can call untrusted code
  - e.g. to ask an applet to repaint its window
- Untrusted code can call trusted code
  - e.g. the paint routine may load a font
- How is the access control policy specified?
<table>
<thead>
<tr>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Java Security Model</td>
</tr>
<tr>
<td>▪ Stack inspection</td>
</tr>
<tr>
<td>▪ Concrete examples</td>
</tr>
<tr>
<td>▪ Semantics from a PL perspective</td>
</tr>
<tr>
<td>▪ Formalizing stack inspection</td>
</tr>
<tr>
<td>▪ Reasoning about programs that use stack inspection</td>
</tr>
<tr>
<td>▪ Type systems for stack inspection</td>
</tr>
<tr>
<td>▪ Discussion &amp; Related work</td>
</tr>
<tr>
<td>▪ Relate stack inspection to information flow</td>
</tr>
</tbody>
</table>

(C# similar)
Java Security Model

VM Runtime

- a.class
- b.class
- c.class
- d.class
- e.class

Classloader
SecurityManager

Security Policy

- Domain A
- Permissions

- Domain B
- Permissions

http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html
Kinds of Permissions

- `java.security.Permission` Class

  perm = new java.io.FilePermission("/tmp/abc","read");

  java.security.AllPermission
  java.security.SecurityPermission
  java.security.UnresolvedPermission
  java.awt.AWTPermission
  java.io.FilePermission
  java.io.SerializablePermission
  java.lang.reflect.ReflectPermission
  java.lang.RuntimePermission
  java.net.NetPermission
  java.net.SocketPermission
  ...

Zdancewic
Code Trustworthiness

- How does one decide what protection domain the code is in?
  - Source (e.g. local or applet)
  - Digital signatures
  - C# calls this “evidence based”

- How does one decide what permissions a protection domain has?
  - Configurable – administrator file or command line

- Enforced by the classloader
Classloader Resolution

- When loading the first class of an application, a new instance of the URLClassLoader is used.
- When loading the first class of an applet, a new instance of the AppletClassLoader is used.
- When java.lang.Class.forName is directly called, the primordial class loader is used.
- If the request to load a class is triggered by a reference to it from an existing class, the class loader for the existing class is asked to load the class.

- Exceptions and special cases... (e.g. web browser may reuse applet loader)
Example Java Policy

grant codeBase "http://www.l33t-hax0rz.com/*" {
    permission java.io.FilePermission("/tmp/*", "read,write");
}

grant codeBase "file://$JAVA_HOME/lib/ext/*" {
    permission java.security.AllPermission;
}

grant signedBy "trusted-company.com" {
    permission java.net.SocketPermission(...);
    permission java.io.FilePermission("/tmp/*", "read,write");
    ...
}

Policy information stored in:
$JAVA_HOME/lib/security/java.policy
$USER_HOME/.java.policy
(or passed on command line)
Example Trusted Code

Code in the System protection domain

```java
void fileWrite(String filename, String s) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        FilePermission fp = new FilePermission(filename, "write");
        sm.checkPermission(fp);
        /* ... write s to file filename (native code) ... */
    } else {
        throw new SecurityException();
    }
}

public static void main(…) {
    SecurityManager sm = System.getSecurityManager();
    FilePermission fp = new FilePermission("/tmp/*", "write, ...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
```
Example Client

Applet code obtained from http://www.l33t-hax0rz.com/

class UntrustedApplet {
    void run() {
        ...
        s.FileWrite("/tmp/foo.txt", "Hello!");
        ...
        s.FileWrite("/home/stevez/important.tex", "kwijibo");
        ...
    }
}
Stack Inspection

- Stack frames are annotated with their protection domains and any enabled privileges.

- During inspection, stack frames are searched from most to least recent:
  - fail if a frame belonging to someone not authorized for privilege is encountered
  - succeed if activated privilege is found in frame
Stack Inspection Example

```java
main(...){
    fp = new FilePermission("/tmp/*","write,...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
```
Stack Inspection Example

```java
main(...){
    fp = new FilePermission("/tmp/*","write,...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
```
Stack Inspection Example

```java
void run() {
    ...
    s.FileWrite("/tmp/foo.txt", "Hello!");
    ...
}

main(...){
    fp = new FilePermission("/tmp/*", "write,...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
```
Stack Inspection Example

```java
void fileWrite("/tmp/foo.txt", "Hello!") {
    fp = new FilePermission("/tmp/foo.txt", "write");
    sm.checkPermission(fp);
    /* ... write s to file filename ... */
}

void run() {
    ...
    s.FileWrite("/tmp/foo.txt", "Hello!");
    ...
}

main(...){
    fp = new FilePermission("/tmp/*", "write,...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
```
Stack Inspection Example

```java
void fileWrite("/tmp/foo.txt", "Hello!") {
    fp = new FilePermission("/tmp/foo.txt", "write");
    sm.checkPermission(fp);
    /* ... write s to file filename ... */
}

void run() {
    ... 
    s.FileWrite("/tmp/foo.txt", "Hello!");
    ...
}

main(...){
    fp = new FilePermission("/tmp/*", "write,...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
```
void run() {
    ...
    s.FileWrite("/home/stevez/important.tex", "kwijibo");
}

main(...){
    fp = new FilePermission("/tmp/*", "write, ...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
void fileWrite(".../important.txt", "kwijibo") {
    fp = new FilePermission("important.txt", "write");
    sm.checkPermission(fp);
}

void run() {
    ...
    s.FileWrite("/home/stevez/important.tex", "kwijibo");
}

main(...){
    fp = new FilePermission("/tmp/*","write,...");
    sm.enablePrivilege(fp);
    UntrustedApplet.run();
}
Other Possibilities

- The `fileWrite` method could enable the write permission itself
  - Potentially dangerous, should not base the file to write on data from the applet
  - ... but no enforcement in Java (information flow would help here)

- A trusted piece of code could *disable* a previously granted permission
  - Terminate the stack inspection early
Stack Inspection Algorithm

checkPermission(T) {
    // loop newest to oldest stack frame
    foreach stackFrame {
        if (local policy forbids access to T by class executing in
            stack frame) throw ForbiddenException;

        if (stackFrame has enabled privilege for T) return;  // allow access

        if (stackFrame has disabled privilege for T)
            throw ForbiddenException;
    }

    // end of stack
    if (Netscape || ...) throw ForbiddenException;
    if (MS IE4.0 || JDK 1.2 || ...) return;
}
Two Implementations

- On demand –
  - On a checkPermission invocation, actually crawl down the stack, checking on the way
  - Used in practice

- Eagerly –
  - Keep track of the current set of available permissions during execution (security-passing style Wallach & Felten)
    - more apparent (could print current perms.)
    - more expensive (checkPermission occurs infrequently)
Stack Inspection

- Stack inspection seems appealing:
  - Fine grained, flexible, configurable policies
  - Distinguishes between code of varying degrees of trust

- But...
  - How do we understand what the policy is?
  - Semantics tied to the operational behavior of the program (defined in terms of stacks!)
  - How do we compare implementations
  - Changing the program (e.g. optimizing it) may change the security policy
  - Policy is distributed throughout the software, and is not apparent from the program interfaces.
  - Is it any good?
Stack Inspection Literature

- A Systematic Approach to Static Access Control
  François Pottier, Christian Skalka, Scott Smith

- Stack Inspection: Theory and Variants
  Cédric Fournet and Andrew D. Gordon

- Understanding Java Stack Inspection
  Dan S. Wallach and Edward W. Felten
  - Formalize Java Stack Inspection using ABLP logic
Formalizing Stack Inspection

Steve Zdancewic
University of Pennsylvania
Stack Inspection

- Stack frames are annotated with their protection domains and any enabled privileges.

- During inspection, stack frames are searched from most to least recent:
  - fail if a frame belonging to someone not authorized for privilege is encountered
  - succeed if activated privilege is found in frame
Stack Inspection Literature

- A Systematic Approach to Static Access Control
  François Pottier, Christian Skalka, Scott Smith

- Stack Inspection: Theory and Variants
  Cédric Fournet and Andrew D. Gordon

- Understanding Java Stack Inspection
  Dan S. Wallach and Edward W. Felten
  - Formalize Java Stack Inspection using ABLP logic
Abstract Stack Inspection

- Abstract permissions

\[ p, q \in P \quad \text{Set of all permissions} \]
\[ R, S \subseteq P \quad \text{Principals (sets of permissions)} \]

- Hide the details of classloading, etc.

- Examples:
  System = \{fileWrite("f1"), fileWrite("f2"),...\}
  Applet = \{fileWrite("f1")\}
**\( \lambda \text{sec Syntax} \)**

- **Language syntax:**

  \[
  e, f ::= \quad \text{expressions}
  \]
  \[
  x \quad \text{variable}
  \]
  \[
  \lambda x. e \quad \text{function}
  \]
  \[
  e \; f \quad \text{application}
  \]
  \[
  R\{ e \} \quad \text{framed expr}
  \]
  \[
  \text{enable p in e} \quad \text{enable}
  \]
  \[
  \text{test p then e else f} \quad \text{check perm.}
  \]
  \[
  \text{fail} \quad \text{failure}
  \]

  \[
  v ::= x \mid \lambda x. e \quad \text{values}
  \]
  \[
  o ::= v \mid \text{fail} \quad \text{outcome}
  \]
Framing a Term

- Models the Classloader that marks the (unframed) code with its protection domain:

\[
R[x] = x \\
R[\lambda x.\, e] = \lambda x.\, R[R[e]] \\
R[e\; f] = R[e]\; R[f] \\
R[\text{enable}\; p\; \text{in}\; e] = \text{enable}\; p\; \text{in}\; R[e] \\
R[\text{test}\; p\; \text{then}\; e\; \text{else}\; f] = \\
\quad \text{test}\; p\; \text{then}\; R[e]\; \text{else}\; R[f] \\
R[\text{fail}] = \text{fail}
\]
Example

```plaintext
readFile =
  λfileName.System{
    test fileWrite(fileName) then
        ...
    else fail
  }

Applet{readFile "f2"} ⇓ fail
System{readFile "f2"} ⇓ <f2 contents>
```
\[ \text{\texttt{\textlambda}sec Operational Semantics} \]

- Evaluation contexts:

  \[
  E ::= \\
  \begin{array}{ll}
  [] & \text{Hole} \\
  E \, e & \text{Eval. Function} \\
  \nu \, E & \text{Eval. Arg.} \\
  \text{enable } p \text{ in } E & \text{Tagged frame} \\
  \text{R}\{E\} & \text{Frame}
  \end{array}
  \]

- \(E\) models the control stack
\textbf{\textit{\lambda}sec Operational Semantics}

\[
E[(\lambda x. e) \; v] \rightarrow E[e\{v/x\}]
\]

\[
E[\text{enable} \; p \; \text{in} \; v] \rightarrow E[v]
\]

\[
E[R\{v\}] \rightarrow E[v]
\]

\[
E[\text{fail}] \rightarrow \text{fail}
\]

\[
E[\text{test} \; p \; \text{then} \; e \; \text{else} \; f] \rightarrow E[e]
\]

\quad \text{if Stack}(E) \models p

\[
E[\text{test} \; p \; \text{then} \; e \; \text{else} \; f] \rightarrow E[f]
\]

\quad \text{if } \neg(\text{Stack}(E) \models p)

\[
e \downarrow o \quad \text{iff} \quad e \rightarrow^* o
\]

Stack Inspection
Example Evaluation Context

Applet{readFile “f2”}

E = Applet{[]} 
 r = readfile “f2”
Example Evaluation Context

\[
E = \text{Applet}[]
\]

\[
r = (\lambda \text{fileName}. \text{System}\{
    \text{test fileWrite(fileName) then}
    \ldots \quad \text{// primitive file IO (native code)}
    \text{else fail}
\}
)

\quad \text{“f2”}
\]
Example Evaluation Context

Applet{readFile "f2"}

E = Applet{[]}

r = System{
    test fwrite("f2") then
    ... // primitive file I/O (native code)
    else fail
}

Zdancewic
Example Evaluation Context

Applet{System{
    test fileWrite("f2") then
    ...
    // primitive file IO (native code)
    else fail
}}
Example Evaluation Context

Applet{System{
    test fileWrite("f2") then
    ... // primitive file IO (native code)
    else fail
}}

E' = Applet{System{[]}}

r' = test fileWrite("f2") then
    ... // primitive file IO (native code)
    else fail
Formal Stack Inspection

\[ E' = \text{Applet\{System\{[]\}\}} \]

\[ r' = \text{test fileWrite(“f2”) then} \]
\[ \quad \ldots \ // \text{primitive file I/O (native code)} \]
\[ \quad \text{else fail} \]

When does stack \( E' \) allow permission fileWrite(“f2”)?

\[ \text{Stack}(E') \vdash \text{fileWrite(“f2”)} \]
Stack of an Eval. Context

- \( \text{Stack}([],) = . \)
- \( \text{Stack}(E \ e) = \text{Stack}(E) \)
- \( \text{Stack}(v \ E) = \text{Stack}(E) \)
- \( \text{Stack}(\text{enable} \ p \ \text{in} \ E) = \text{enable}(p).\text{Stack}(E) \)
- \( \text{Stack}(R\{E\}) = R.\text{Stack}(E) \)

\[
\text{Stack}(E') \\
= \text{Stack}(\text{Applet}\{\text{System}\{[]\}\}) \\
= \text{Applet.\text{Stack}}(\text{System}\{[]\}) \\
= \text{Applet.\text{System.\text{Stack}}([[]])} \\
= \text{Applet.\text{System.}}
\]
Abstract Stack Inspection

. \vdash p \quad \text{empty stack axiom}

\[ \frac{x \vdash p \quad p \in R}{x.R \vdash p} \quad \text{protection domain check} \]

\[ \frac{x \vdash p}{x.\text{enable}(q) \vdash p} \quad \text{irrelevant enable} \]

\[ \frac{x \vdash p}{x.\text{enable}(p) \vdash p} \quad \text{check enable} \]
Abstract Stack Inspection

\[ \vdash p \quad \text{empty stack enables all} \]

\[ p \in R \quad \frac{}{x.R \vdash p} \quad \text{enable succeeds*} \]

\[ \frac{x \vdash p}{x.\text{enable}(q) \vdash p} \quad \text{irrelevant enable} \]

* Enables should occur only in trusted code
Equational Reasoning

e \downarrow \iff \text{there exists } o \text{ such that } e \downarrow o

Let C[\cdot] be an arbitrary program context.

Say that \( e \simeq e' \) iff for all C[], if C[e] and C[e'] are closed then
\( C[e] \downarrow \iff C[e'] \downarrow. \)
Example Inequality

let $x = e$ in $e' = (\lambda x.e') e$

$ok = \lambda x.x$

$loop = (\lambda x.x \ x)(\lambda x.x \ x)$  (note: loop $\Downarrow$)

$f = \lambda x. \text{let } z = x \text{ ok in } \lambda _z$

$g = \lambda x. \text{let } z = x \text{ ok in } \lambda _z(x \text{ ok})$

Claim: $f \not\simeq g$

Proof:
Let $C[] = \emptyset[[] \lambda _.\text{test } p \text{ then loop else ok}] \text{ ok}$
Example Continued

\[
C[f] = \emptyset \{f \lambda_.\text{test } p \text{ then loop else ok} \} \text{ ok}
\]

- \[
\rightarrow \emptyset \{\text{let } z = \lambda_.\text{test } p \text{ then loop else ok} \} \text{ ok}
\]
- \[
\rightarrow \emptyset \{\text{let } z = \text{ test } p \text{ then loop else ok} \text{ in } \lambda_.z \} \text{ ok}
\]
- \[
\rightarrow \emptyset \{\text{let } z = \text{ ok in } \lambda_.z \} \text{ ok}
\]
- \[
\rightarrow \emptyset \{\lambda_.\text{ok} \} \text{ ok}
\]
- \[
\rightarrow (\lambda_.\text{ok}) \text{ ok}
\]
- \[
\rightarrow \text{ ok}
\]
Example Continued

\[ C[g] = \emptyset \{ g \, \lambda_. \text{test p then loop else ok} \} \, \text{ok} \]

- \[ \rightarrow \emptyset \{ \text{let } z = \]
  \[ (\lambda_. \text{test p then loop else ok} \) \, \text{ok} \]
  \[ \text{in } \lambda_. ( (\lambda_. \text{test p then loop else ok} \) \, \text{ok}) \} \, \text{ok} \]
- \[ \rightarrow \emptyset \{ \text{let } z = \text{test p then loop else ok} \]
  \[ \text{in } \lambda_. ( (\lambda_. \text{test p then loop else ok} \) \, \text{ok}) \} \, \text{ok} \]
- \[ \rightarrow \emptyset \{ \text{let } z = \text{ok} \]
  \[ \text{in } \lambda_. ( (\lambda_. \text{test p then loop else ok} \) \, \text{ok}) \} \, \text{ok} \]
- \[ \rightarrow \emptyset \{ \lambda_. ( (\lambda_. \text{test p then loop else ok} \) \, \text{ok}) \} \, \text{ok} \]
- \[ \rightarrow (\lambda_. ( (\lambda_. \text{test p then loop else ok} \) \, \text{ok})) \, \text{ok} \]
- \[ \rightarrow (\lambda_. \text{test p then loop else ok}) \, \text{ok} \]
- \[ \rightarrow \text{test p then loop else ok} \]
- \[ \rightarrow \text{loop} \rightarrow \text{loop} \rightarrow \text{loop} \rightarrow \text{loop} \rightarrow \text{loop} \rightarrow \text{...} \]
Example Applications

Eliminate redundant annotations:
\[ \lambda x.R\{\lambda y.R\{e\}\} \approx \lambda x.\lambda y.R\{e\} \]

Decrease stack inspection costs:
\[ e \approx \text{test } p \text{ then (enable } p \text{ in } e \text{) else } e \]
Axiomatic Equivalence

Can give a sound set of equations $\equiv$ that characterize $\simeq$. Example axioms:

- $\equiv$ is a congruence (preserved by contexts)
- $(\lambda x. e) v \equiv e\{v/x\}$ (beta equivalence)
- $x \not\in \text{fv}(v) \Rightarrow \lambda x. v \equiv v$
- $\text{enable } p \text{ in } o \equiv o$
- $\text{enable } p \text{ in } (\text{enable } q \text{ in } e) \equiv$
  $\text{enable } q \text{ in } (\text{enable } p \text{ in } e)$
- $R \supseteq S \Rightarrow R\{S\{e\}\} \equiv S\{e\}$
- $R\{S\{\text{enable } p \text{ in } e\}\} \equiv$
  $R \cup \{p\}\{S\{\text{enable } p \text{ in } e\}\}$
- ... many, many more

$\equiv \text{ Implies } \simeq$
Example: Tail Calls

Ordinary evaluation:
\[ R\{(\lambda x. S\{e\}) \ v\} \rightarrow R\{S\{e\{v/x\}\}\} \]

Tail-call eliminated evaluation:
\[ R\{(\lambda x. S\{e\}) \ v\} \rightarrow S\{e\{v/x\}\} \]

Not sound in general!

But OK in special cases.
Example: Tail Calls

Suppose $R \supseteq S$. Then:

$$R\{(\lambda x. S\{e\})\ v\} \equiv R\{S\{e\{v/x\}\}\} \equiv S\{e\{v/x\}\} \equiv S\{e\}\{v/x\} \equiv (\lambda x. S\{e\})\ v$$

In particular, code within a protection domain can safely make tail calls to other code in that domain.
Example: Higher-order Code

main = System [ λh.(h ok ok)]

fileHandler =
    System[λs.λc.λ_.c (readFile s)]

leak = Applet[λs.output s]

main(λ_.Applet{fileHandler “f2” leak})
Example: Higher-order Code

- main(\_.Applet\{fileHandler "f2" leak\})
- \rightarrow* System\{Applet\{fileHandler "f2" leak\} okS\}
- \rightarrow* System\{Applet\{System\{System\{
    \_.System\{leak (readFile "f2")\}\}\}\}\}\ okS\}
- \rightarrow* System\{\_.System\{leak (readFile "f2")\} okS\}
- \rightarrow* System\{System\{leak <f2 contents>\}\}\}
- \rightarrow* System\{System\{Applet\{output <f2 contents>\}\}\}\}
- \rightarrow* System\{System\{Applet\{ok\}\}\}
- \rightarrow* ok
Next Time

- Static analysis for stack inspection
  - Type system for stack inspection

- Connections to information-flow analysis
Stack Inspection: Translation & Static Analysis

Steve Zdancewic
University of Pennsylvania
Types for Stack Inspection

- Want to do static checking of $\lambda_{sec}$ code
  - Statically detect security failures.
  - Eliminate redundant checks.
  - Example of nonstandard type system for enforcing security properties.

- Type system based on work by Pottier, Skalka, and Smith:
  - “A Systematic Approach to Static Access Control”

- Explain the type system by taking a detour through “security-passing” style.
  - See Wallach’s & Felten’s
\text{\textlambda sec Syntax}

- Language syntax:

\[
e, f ::= \text{expressions} \\
\quad x \quad \text{variable} \\
\quad \lambda x.e \quad \text{function} \\
\quad e f \quad \text{application} \\
\quad \text{R}\{e\} \quad \text{framed expr} \\
\quad \text{enable p in e} \quad \text{enable} \\
\quad \text{test p then e else f} \quad \text{check perm.} \\
\quad \text{let x = e in f} \quad \text{local decl.}
\]

- Restrict the use of “fail” in the source language
Adding Static Checking

- New expression form:
  
  \[
  \text{check } p \text{ then } e
  \]

- Operationally, equivalent to:
  
  \[
  \text{test } p \text{ then } e \text{ else fail}
  \]

- But, the type system will ensure that the check always succeeds.
Security-passing Style

- Basic idea: Convert the “stack-crawling” form of stack inspection into a “permission-set passing style”
  - Compute the set of current permissions at any point in the code.
  - Make the set of permissions explicit as an extra parameter to functions (hence “security-passing style”)

- Target language is untyped lambda calculus with a primitive datatype of sets.
Yet another formalization of stack inspection:

Compute the set \( T \) of permissions granted by stack \( x \) given starting with static permissions \( R \) and dynamic permissions \( S \).

Change to \( \emptyset; \emptyset; x \) for the “least privileges” version

\[
P; P; x \vdash T \quad p \in T
\]

\[
x \vdash p
\]
"Eager" Stack Inspection

\[
R; S; . \vdash S
\]
Bottom of the stack

\[
\frac{R'; S \cap R'; x \vdash T}{R; S; R'.x \vdash T}
\]
New prot. Domain.

\[
\frac{R; S \cup \{p\} \cap R); x \vdash T}{R; S; \text{enable}(p).x \vdash T}
\]
Enabled permission.
Lemma: $\text{Stack}(E) \vdash p$ in the first formulation iff $\text{Stack}(E) \vdash p$ in the eager formulation.
Target Language: $\lambda$set

- **Language syntax:**
  
  $e, f ::= 
  \begin{align*}
  &x &\text{variable} \\
  &\lambda x. e &\text{function} \\
  &e \ f &\text{application} \\
  &\text{fail} &\text{failure} \\
  &\text{let } x = e \text{ in } f &\text{local decl.} \\
  &\text{if } p \in s e \text{ then } e \text{ else } f &\text{member test} \\
  &s e &\text{set expr.}
  \end{align*}$

- $s e ::= 
  \begin{align*}
  &S &\text{perm. set} \\
  &s e \cup s e &\text{union} \\
  &s e \cap s e &\text{intersection} \\
  &x &
  \end{align*}$
Translation: $\lambda$sec to $\lambda$set

- $[[e]]_R = \text{“translation of } e \text{ in domain } R\text{”}$

- $[[x]]_R = x$

- $[[\lambda x.e]]_R = \lambda x.\lambda s. [[e]]_R$

- $[[e f]]_R = [[e]]_R [[f]]_R \ s$

- $[[\text{let } x = e \text{ in } f]]_R = \text{let } x = [[e]]_R \text{ in } [[f]]_R$

- $[[\text{enable } p \text{ in } e]]_R = \text{let } s = s \cup \{p\} \cap R \text{ in } [[e]]_R$

- $[[R\{e\}]]_R = \text{let } s = s \cap R' \text{ in } [[e]]_R'$

- $[[\text{check } r \text{ then } e]]_R = \text{if } r \in s \text{ then } [[e]]_R \text{ else } \text{fail}$

- $[[\text{test } r \text{ then } e1 \text{ else } e2]]_R$
  
  = \text{if } r \in s \text{ then } [[e1]]_R \text{ else } [[e2]]_R$

- Top level translation: $[[e]] = [[e]]_{P\{P/s\}}$
Example Translation

System = {“f1, “f2”, “f3”}
Applet = {“f1”}

h = System{enable “f1” in
    Applet{((λx.
        System{check “f1” then write x}]
    “kwijibo”} }
Example Translation

[[h]] =

(* System *)
let s = P ∩ \{“f1”, “f2”, “f3”\} in

(* enable “f1” *)
let s = s ∪ (\{“f1”\} ∩ \{“f1”, “f2”, “f3”\}) in

(* Applet *)
let s = s ∩ \{“f1”\} in

(λx.λs.

(* System *)
let s = s ∩ \{“f1”, “f2”, “f3”\} in
if “f1” ∈ s then write x else fail)

“kwijibo” s
“Administrative” Evaluation

(1) \( \text{let } s = e \text{ in } f \rightarrow_a f\{R/s\} \text{ if } e \rightarrow^* R \)

(2) \( E[e] \rightarrow_a E[e'] \text{ if } e \rightarrow_a e' \)

For example:
\[
[[h]] \rightarrow_a *
(\lambda x.\lambda s. \\
(* \text{System } *)
\text{let } s = s \cap \{"f1", "f2", "f3"\} \text{ in }
\text{if } "f1" \in s \text{ then write } x \text{ else } ()
"kwijibo" \{"f1"\}
\]
Stack Inspection Lemma

Lemma:

- Suppose \( R; S; \text{Stack}(E) \vdash T \).
  Then there exist \( E' \) and \( R' \) such that for all (source) \( e \):

\[
[[E[e]]]R\{S/s\} \xrightarrow{a^*} E'[[[e]]R'{T/s}] 
\]

Proof (sketch): By induction on structure of \( E \).
Translation Correctness (1)

Lemma:

- If $e \rightarrow e'$ then there is an $f$ such that $[[e]] \rightarrow^* f$ and $[[e']] \rightarrow_a^* f$

- Furthermore, if $e \rightarrow e'$ is a beta step, then $[[e]] \rightarrow^* f$ includes at least one beta step.

Proof (sketch): Induction on the evaluation step taken. Uses the stack inspection lemma.
Translation Correctness

Theorem:

- If \( e \rightarrow^* v \) then \([e] \rightarrow^* [v]\)
- If \( e \rightarrow^* \text{fail} \) then \([e] \rightarrow^* \text{fail}\)
- Furthermore, if \( e \) diverges, so does \([e]\).

Proof (sketch): Use the lemma on the previous slide.
Stepping Back

- Have two formulations of stack inspection: “original” and “eager”

- Have a translation to a language that manipulates sets of permissions explicitly.
  - Includes the “administrative” reductions that just compute sets of permissions.
  - Similar computations can be done statically!
Deriving a Type System

- Eager stack inspection judgment:

\[ R; S; \text{Stack}(E) \vdash T \]

- Statically track the current protection domain
- Statically track the currently enabled permissions
- Use the expression instead of Stack(E)
Typing Judgments

\[ R;S;\Gamma \vdash e : t \]

- **Variable context**
- **Current protection domain**
- **Current permission set**
- **Term**
- **Type**
Form of types

- Only interesting (non administrative) change during compilation was for functions:
  \[ [[\lambda x.e]]R = \lambda x.\lambda s.[[e]]R \]

- Source type: \( t \rightarrow u \)
- Target type: \( t \rightarrow s \rightarrow u \)
- The 2\textsuperscript{nd} argument, is always a set, so we “specialize” the type to:
  \( t \rightarrow \{S\} \rightarrow u \)
Types

- Types:

  \[ t ::= \]
  \[ \text{int, string, ...} \]
  \[ t \rightarrow \{S\} \rightarrow t \]

  types
  base types
  functions
Simple Typing Rules

Variables:

\[ R;S;\Gamma \vdash x : \Gamma(x) \]

Abstraction:

\[ R;S';\Gamma,x:t1 \vdash e : t2 \quad \Rightarrow \quad R;S;\Gamma \vdash \lambda x.e : t1 \rightarrow \{S'\} \rightarrow t2 \]
More Simple Typing Rules

Application:

\[
\begin{align*}
R;S;\Gamma & \vdash e : t \rightarrow \{S\} \rightarrow t' \\
R;S;\Gamma & \vdash f : t \\
R;S;\Gamma & \vdash e \ f : t'
\end{align*}
\]

Let:

\[
\begin{align*}
R;S;\Gamma & \vdash e : u \\
R;S;\Gamma,x:u & \vdash f : t \\
R;S;\Gamma & \vdash \text{let } x = e \text{ in } f : t
\end{align*}
\]
Typing Rules for Enable

Enable fail: \[ \frac{R;S;\Gamma \vdash e : t \quad p \notin R}{R;S;\Gamma \vdash \text{enable } p \text{ in } e : t} \]

Enable succeed: \[ \frac{R;S \cup \{p\};\Gamma \vdash e : t \quad p \in R}{R;S;\Gamma \vdash \text{enable } p \text{ in } e : t} \]
Rule for Check

Note that this typing rule requires that the permission p is statically known to be available.

\[
\begin{align*}
R; S \cup \{p\}; \Gamma & \vdash e : t \\
R; S \cup \{p\}; \Gamma & \vdash \text{check } p \text{ then } e : t
\end{align*}
\]
Rule for Test

Check the first branch under assumption that \( p \) is present, check the else branch under assumption that \( p \) is absent.

\[
\begin{align*}
R; S \cup \{p\}; \Gamma &\vdash e : t \\
R; S - \{p\}; \Gamma &\vdash f : t \\
\hline
R; S; \Gamma &\vdash \text{test } p \text{ then } e \text{ else } f : t
\end{align*}
\]
Rule for Protection Domains

Intersect the permissions in the static protection domain with the current permission set.

\[ S' ; S \cap S' ; \Gamma \vdash e : t \]

\[ R ; S ; \Gamma \vdash S' \{ e \} : t \]
Weakening (Subsumption)

It is always safe to “forget” permissions.

\[
R; S'; \Gamma \vdash e : t \quad S' \subseteq S
\]

\[
R; S; \Gamma \vdash e : t
\]
Type Safety

- Theorem:
  If \( P;P;\emptyset \vdash e : t \) then either \( e \rightarrow^* v \) or \( e \) diverges.

- In particular: \( e \) never fails. (i.e. check always succeeds)

- Proof:
  Preservation & Progress.
Example: Good Code

\[
\text{h} = \text{System}\{\text{enable } "f1" \text{ in} \\
\hspace{1cm} \text{Applet}\{(\lambda x. \\
\hspace{2cm} \text{System}\{\text{check } "f1" \text{ then write } x\} ) \\
\hspace{2cm} "kwijibo"\} \}
\]

Then \[P;S;\emptyset \models h : \text{unit} \quad \text{for any } S\]
Example: Bad Code

g = System{enable “f1” in
    Applet{((\lambda x.  
        System{check (“f2” then write x})
    “kwijibo”})}

Then R;S;\emptyset \vdash g : t is not derivable for any R,S, and t.
Static vs. Dynamic Checks

Calling this function requires the *static* permission $p$:

$$\emptyset;\emptyset;\emptyset \vdash \lambda x.\text{check } p \text{ in } x : \text{int} \rightarrow \text{int}$$

Only way to call it (assuming initial perms. are empty) is to put it in the scope of a *dynamic* test:

```
  test p then ...can call it here...
  else ...may not call it here...
```
Expressiveness

- This type system is very simple
  - No subtyping
  - No polymorphism
  - Not algorithmic
  - Hard to do inference
- Can add all of these features...
- See François’ paper for a nice example.
  - Uses Rémy’s row types to describe the sets of permission.
  - Uses HM(X) – Hindley Milner with constraints
  - Also shows how to derive a type system for the source language from the translation!
Discussion

- Problem: Applets returning closures that circumvent stack inspection.
- Possible solution:
  - Values of the form: $R\{v\}$ (i.e. keep track of the protection domain of the source)
  - Similarly, one could have closures capture their current security context
  - Integrity analysis (i.e. where data comes from)
- Fournet & Gordon prove some properties of strengthened versions of stack inspection.
Stack Inspection ++

- Stack inspection enforces a form of integrity policy
- Can combine stack inspection with information-flow policies:
  - Banerjee & Naumann – Using Access Control for Secure Information Flow in a Java-like Language (CSFW’03)