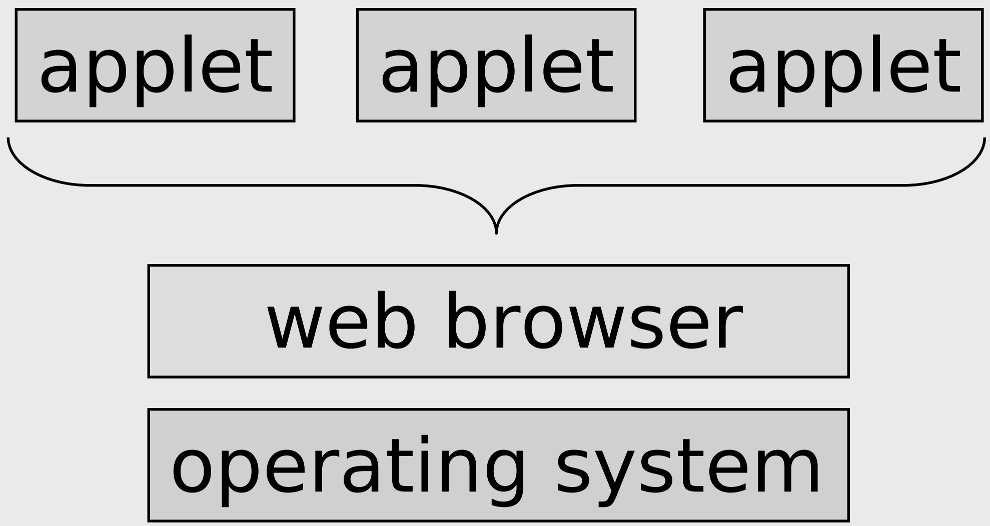
Current Techniques in Language-based Security

Steve Zdancewic
University of Pennsylvania

Mobile Code

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



- 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

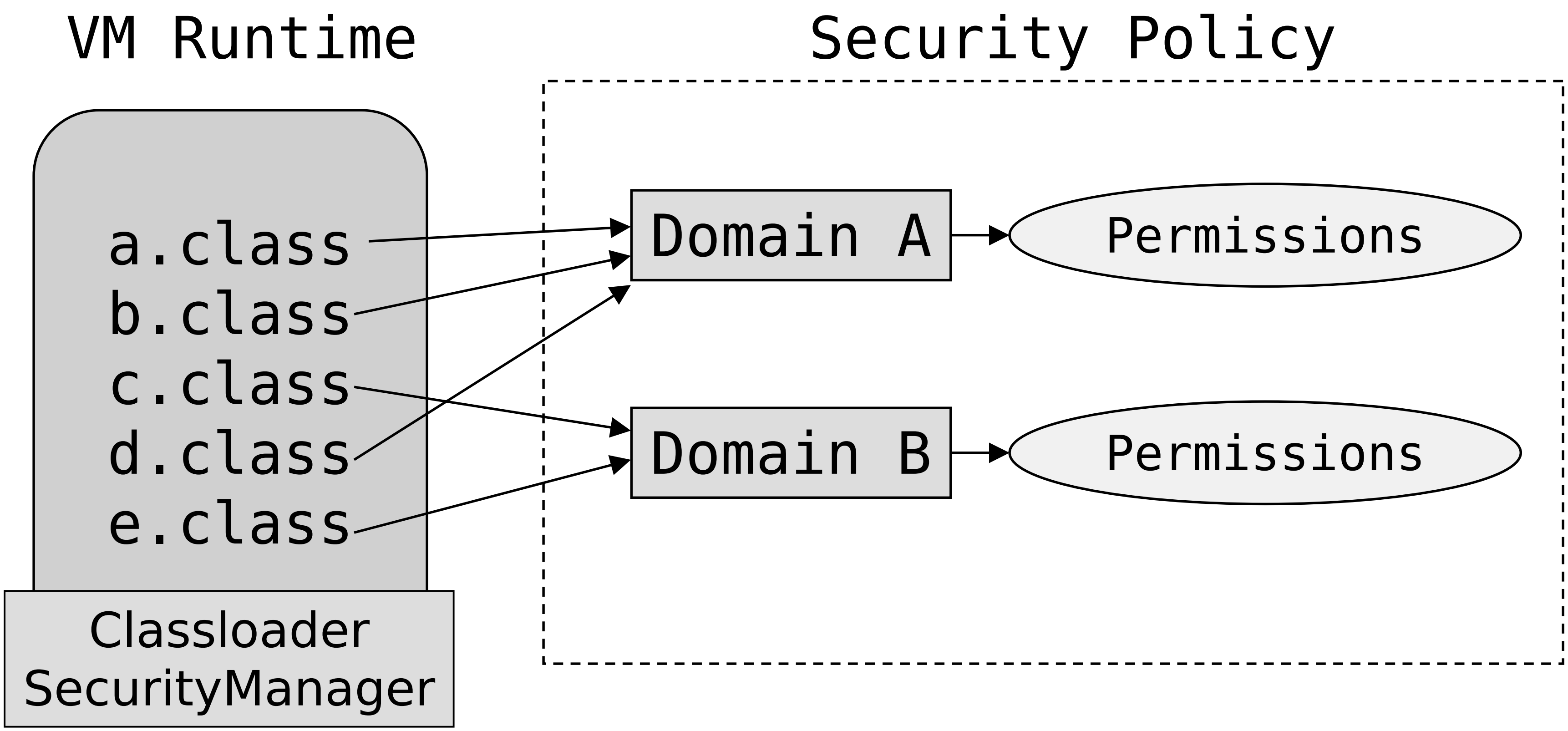
Access Control for Applets

- 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?

Outline

- Java Security Model (C# similar)
- Stack inspection
  - Concrete examples
- Semantics from a PL perspective
  - Formalizing stack inspection
  - Reasoning about programs that use stack inspection
  - Type systems for stack inspection
- Discussion & Related work
  - Relate stack inspection to information flow

Java Security Model



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

Kinds of Permissions

- java.security.Permission Class

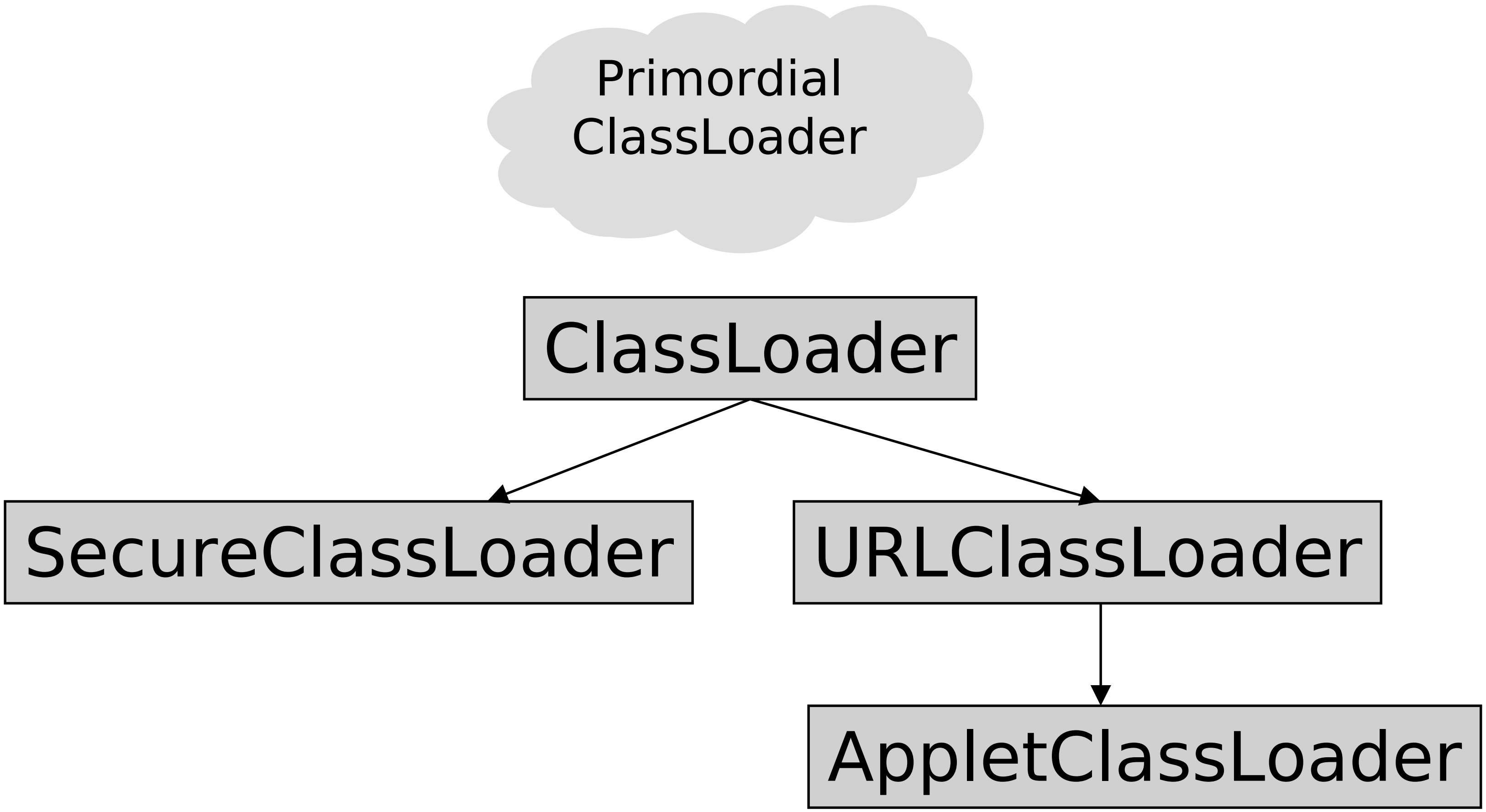
```
perm = new java.io.FilePermission("/tmp/abc","read");

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
…
```

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 Hierarchy



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

```
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

Policy Database

```
main(…){
 fp = new FilePermission(“/tmp/*”,“write,…”);
 sm.enablePrivilege(fp);
 UntrustedApplet.run();
}
```

Stack Inspection Example

```
main(…){
 fp = new FilePermission(“/tmp/*”,“write,…”);
 sm.enablePrivilege(fp);
 UntrustedApplet.run();
}
```

fp

Policy Database

Stack Inspection Example

Policy Database

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

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

fp

Stack Inspection Example

```
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();
}
```

fp

Policy Database

Stack Inspection Example

Stack Inspection Example

```
void run() {
  …
  s.FileWrite(“/home/stevez/important.tex”,
              “kwijibo”);
}
```

```
main(…){
 fp = new FilePermission(“/tmp/*”,“write,…”);
 sm.enablePrivilege(fp);
 UntrustedApplet.run();
}
```

fp

Policy Database

Stack Inspection Example

```
void fileWrite("…/important.txt", "kwijibo") {
 fp = new FilePermission("important.txt",
                         "write");
 sm.checkPermission(fp);
```

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

Fail

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

fp

Policy Database

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

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

- Hide the details of classloading, etc.
- Examples:

  System = {fileWrite("f1"), fileWrite("f2"),…}
  Applet = {fileWrite("f1")}

λ_{sec} Syntax

- Language syntax:

```
e,f ::=                            expressions
  x                                variable
  λx.e                             function
  e f                              application
  R{e}                             framed expr
  enable p in e                    enable
  test p then e else f             check perm.
  fail                             failure

v ::= x  |  λx.e                   values
o ::= v  |  fail                   outcome
```

Framing a Term

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

```
R⟦x⟧      =  x
R⟦λx.e⟧  =  λx.R{R⟦e⟧}
R⟦e f⟧   =  R⟦e⟧ R⟦f⟧
R⟦enable p in e⟧ = enable p in R⟦e⟧
R⟦test p then e else f⟧ =
  test p then R⟦e⟧ else R⟦f⟧
R⟦fail⟧ =  fail
```

Example

```
readFile =
  λfileName.System{
    test fileWrite(fileName) then
     … // primitive file IO (native code)
    else fail
  }
```

Applet{readFile “f2”} $\Downarrow$ fail

System{readFile “f2”} $\Downarrow$ <f2 contents>

λ_{sec} Operational Semantics

- Evaluation contexts:

```
E ::=
   []                   Hole
   E e                  Eval. Function
   v E                  Eval. Arg.
   enable p in E        Tagged frame
   R{E}                 Frame
```

- E models the control stack

λ_{sec} Operational Semantics

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

$E[\texttt{enable p in v}] \rightarrow E[v]$

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

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

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

if $\text{Stack}(E) \vdash p$

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

if $\neg(\text{Stack}(E) \vdash p)$

Stack Inspection

$e \Downarrow o \quad \text{iff} \quad e \rightarrow^* o$

Example Evaluation Context

```
Applet{readFile "f2"}

E = Applet{[]}
r = readfile "f2"
```

Example Evaluation Context

```
Applet{readFile “f2”}

E = Applet{[]}
r = (λfileName.System{
      test fileWrite(fileName) then
       … // primitive file IO (native code)
      else fail
      })
    “f2”
```

Example Evaluation Context

```
Applet{readFile “f2”}

E = Applet{[]}
r = 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
     }}
```

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' = Applet{System{[]}}

r' = `test fileWrite("f2") then`

```
        … // primitive file IO (native code)
        else fail
```

When does stack E' allow permission fileWrite("f2")?

$$\text{Stack}(E') \vdash \text{fileWrite}(\text{“f2”})$$

Stack of an Eval. Context

```
Stack([])              = .
Stack(E e)             = Stack(E)
Stack(v E)             = Stack(E)
Stack(enable p in E) = enable(p).Stack(E)
Stack(R{E})            = R.Stack(E)
```

Stack(E’)
= Stack(Applet{System{[]}})
= Applet.Stack(System{[]})
= Applet.System.Stack([])
= Applet.System.

Abstract Stack Inspection

$$. \vdash p$$

empty stack axiom

$$\frac{x \vdash p \quad p \in R}{x.R \vdash p}$$

protection domain check

$$\frac{x \vdash p}{x.\mathrm{enable}(q) \vdash p} \quad p \neq q$$

irrelevant enable

$$\frac{x \models p}{x.\mathrm{enable}(p) \vdash p}$$

check enable

Abstract Stack Inspection

$$. \models p$$

empty stack enables all

$$\frac{p \in R}{x.R \models p}$$

enable succeeds*

$$\frac{x \models p}{x.\mathrm{enable}(q) \models p}$$

irrelevant enable

* Enables should occur only in trusted code

Equational Reasoning

e$\Downarrow$ iff there exists o such that e $\Downarrow$ o

Let C[] be an arbitrary program context.

Say that e $\approx$ 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 $\not\Downarrow$)

f = $\lambda x.$ let z = x ok in $\lambda\_.z$

g = $\lambda x.$ let z = x ok in $\lambda\_.(x\ ok)$

Claim: f $\not\approx$ g

Proof:

Let C[] = $\varnothing$\{[] $\lambda\_$.test p then loop else ok\} ok

Example Continued

C[f] = ∅{f λ_.test p then loop else ok} ok

- → ∅{let z =
  (λ_.test p then loop else ok) ok
  in λ_.z} ok
- → ∅{let z = test p then loop else ok
  in λ_.z} ok
- → ∅{let z = ok in λ_.z} ok
- → ∅{λ_.ok} ok
- → (λ_.ok) ok
- → ok

Example Continued

C[g] = ∅{g λ_.test p then loop else ok} ok

- → ∅{let z =
  (λ_.test p then loop else ok) ok
  in λ_.((λ_.test p then loop else ok) ok)} ok
- → ∅{let z = test p then loop else ok
  in λ_. ((λ_.test p then loop else ok) ok)} ok
- → ∅{let z = ok
  in λ_. ((λ_.test p then loop else ok) ok)} ok
- → ∅{λ_. ((λ_.test p then loop else ok) ok)} ok
- → (λ_. ((λ_.test p then loop else ok) ok)) ok
- → (λ_.test p then loop else ok) ok
- → test p then loop else ok
- → loop → loop → loop → loop → …

Example Applications

Eliminate redundant annotations:

$\lambda x.R\{\lambda y.R\{e\}\} \approx \lambda x.\lambda y.R\{e\}$

Decrease stack inspection costs:

$e \approx$ test p then (enable p in e) else e

Axiomatic Equivalence

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

- $\equiv$ is a congruence (preserved by contexts)
- $(\lambda x.e)\ v \equiv e\{v/x\}$ (beta equivalence)
- $x \notin fv(v) \Rightarrow \lambda x.v \equiv v$
- enable p in o $\equiv$ o
- enable p in (enable q in e) $\equiv$ enable q in (enable p in e)
- $R \supseteq S \Rightarrow R\{S\{e\}\} \equiv S\{e\}$
- R{S{enable p in e}} $\equiv$ R$\cup${p}{S{enable p in e}}
- … many, many more

$\equiv$ Implies $\approx$

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:

$$
\begin{aligned}
& R\{(\lambda x.S\{e\})\ v\} \\
\equiv\ & R\{S\{e\{v/x\}\}\} \\
\equiv\ & S\{e\{v/x\}\} \\
\equiv\ & S\{e\}\{v/x\} \\
& (\lambda x.S\{e\})\ v
\end{aligned}
$$

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(λ_.Applet{fileHanler “f2” leak})
- $\rightarrow^*$ System{Applet{fileHandler “f2” leak} okS}
- $\rightarrow^*$ System{Applet{System{System{
  λ_.System{leak (readFile “f2”)}}}}} okS}
- $\rightarrow^*$ System{λ_.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 λ_{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

λ_{sec} Syntax

- Language syntax:

```
e,f ::=                              expressions
  x                                  variable
  λx.e                               function
  e f                                application
  R{e}                               framed expr
  enable p in e                      enable
  test p then e else f               check perm.
  let x = e in f                     local decl.
```

- Restrict the use of “fail” in the source language

Adding Static Checking

- New expression form:

```
check p then e
```

- Operationally, equivalent to:

```
test p then e 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.

YAFOSI

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 ∅;∅;x for the “least privileges” version

Compute ... the order the st ... as built).

$$\frac{P; P; x \vdash T \quad p \in T}{x \vdash p}$$

"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.

Inspection Correspondence

Lemma: Stack(E) $\vdash$ p in the first formulation iff Stack(E) $\vdash$ p in the eager formulation.

Target Language: λset

- Language syntax:

```
e,f ::=                               expressions
    x                                 variable
    λx.e                              function
    e f                               application
    fail                              failure
    let x = e in f                    local decl.
    if p∈se then e else f             member test
    se                                set expr.
```

- se ::=

```
    S                                 perm. set
    se ∪ se                           union
    se ∩ se                           intersection
    x
```

Translation: λsec to λset

- [[e]]R = "translation of e in domain R"

- [[x]]R = x
- [[λx.e]]R = λx.λs.[[e]]R
- [[e f]]R = [[e]]R [[f]]R s
- [[let x = e in f]]R = let x = [[e]]R in [[f]R
- [[enable p in e]]R = let s = s ∪ ({p} ∩ R) in [[e]]R
- [[R'{e}]]R = let s = s ∩ R' in [[e]]R'
- [[check r then e]]R = if r ∈ s then [[e]]R else fail
- [[test r then e1 else e2]]R
  = if r ∈ s then [[e1]]R 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) let s = e in f $\rightarrow_a$ f{R/s} if e $\rightarrow^*$ R

(2) E[e] $\rightarrow_a$ E[e’] if e $\rightarrow_a$ e’

For example:

[[h]] $\rightarrow_a^*$

(λx.λs.
  (System *)*
  let s = s ∩ {“f1”, “f2”, “f3”} in
  if “f1” ∈ s then write x else ())
“kwijibo” {“f1”}

Stack Inspection Lemma

Lemma:

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

$$[[E[e]]]R\{S/s\} \rightarrow_a^* E'[[[e]]R'\{T/s\}]$$

Proof (sketch): By induction on structure of E.

Translation Correctness (1)

Lemma:

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

- Furthermore, if e→e' is a beta step, then [[e]] →* 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 →* v then [[e]] →* [[v]]
- If e →* fail then [[e]] →* 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!

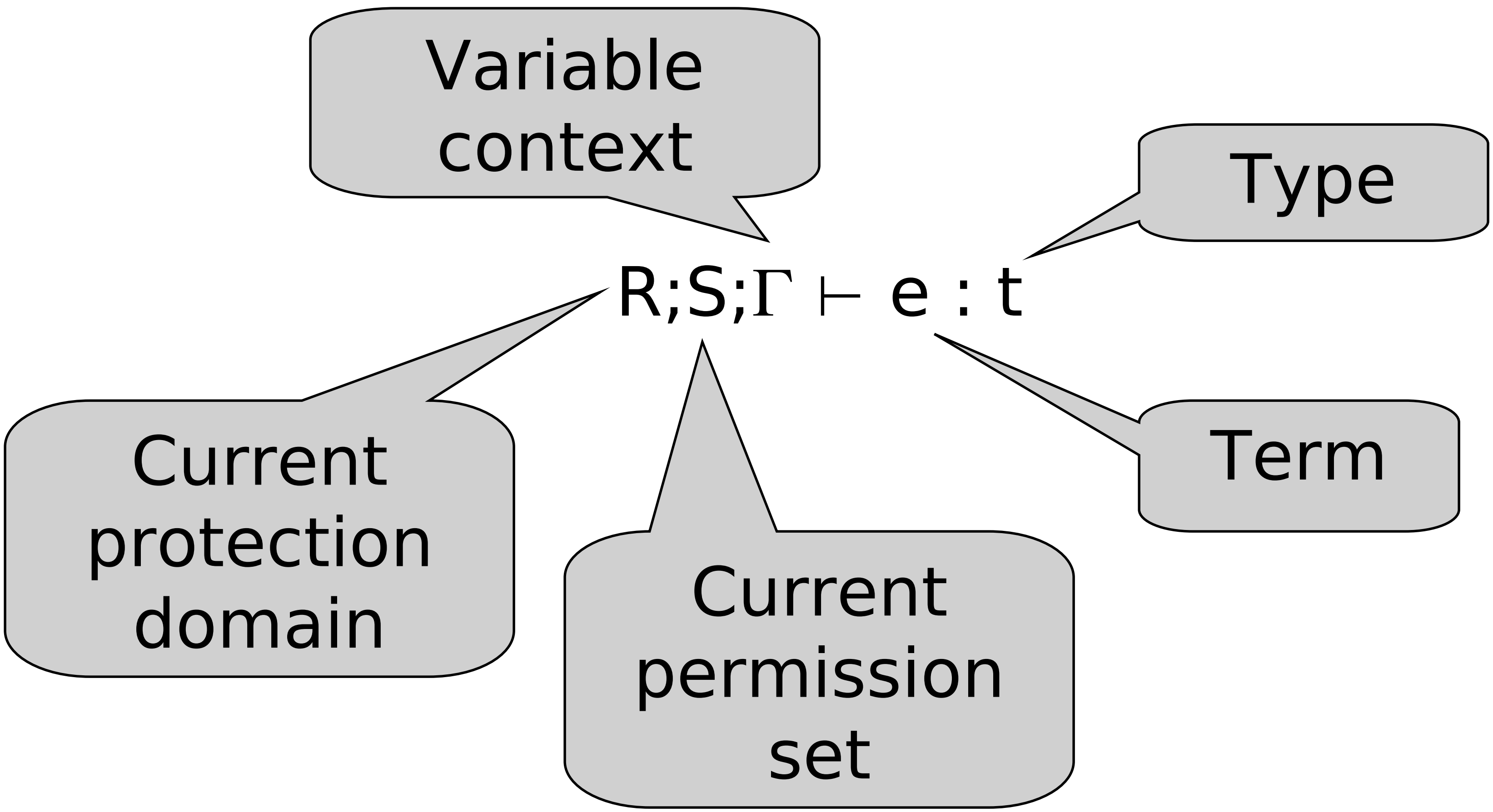
Deriving a Type System

- Eager stack inspection judgment:

$$R; S; Stack(E) \vdash T$$

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

Typing Judgments



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 2nd argument, is always a set, so we “specialize” the type to: $t -\{S\}\rightarrow u$

Types

- Types:

t ::=	types
int, string, …	base types
$t \text{ –}\{S\}\rightarrow t$	functions

Simple Typing Rules

Variables: $R;S;\Gamma \vdash x : \Gamma(x)$

Abstraction:

$$\frac{R;S';\Gamma,x{:}t1 \vdash e \ : \ t2}{R;S;\Gamma \vdash \lambda x.e \ : \ t1 \ \text{–}\{S'\}\!\rightarrow t2}$$

More Simple Typing Rules

Application:

$$\frac{R;S;\Gamma \vdash e : t \xrightarrow{\{S\}} t' \qquad R;S;\Gamma \vdash f : t}{R;S;\Gamma \vdash e\ f : t'}$$

Let:

$$\frac{R;S;\Gamma \vdash e : u \qquad R;S;\Gamma,x{:}u \vdash f : t}{R;S;\Gamma \vdash \text{let } x = e \text{ in } f : t}$$

Typing Rules for Enable

Enable fail:

$$\frac{R;S;\Gamma \vdash e : t \qquad p \notin R}{R;S;\Gamma \vdash \text{enable } p \text{ in } e : t}$$

Enable succeed:

$$\frac{R;S\cup\{p\};\Gamma \vdash e : t \qquad p \in R}{R;S;\Gamma \vdash \text{enable } p \text{ in } e : t}$$

Rule for Check

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

$$\frac{R;\ S\cup\{p\};\Gamma \vdash e\ :\ t}{R;\ S\cup\{p\};\Gamma \vdash \text{check } p \text{ then } e : t}$$

Rule for Test

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

$$\frac{R; S\cup\{p\};\Gamma \vdash e\ : t \qquad R;S-\{p\};\Gamma \vdash f\ : t}{R;S;\Gamma \vdash \text{test } p \text{ then } e \text{ else } f: t}$$

Rule for Protection Domains

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

$$\frac{S';S \cap S';\Gamma \vdash e \ : t}{R;S;\Gamma \vdash S'\{e\}: t}$$

Weakening (Subsumption)

It is always safe to "forget" permissions.

$$\frac{R;S';\Gamma \vdash e : t \qquad S' \subseteq S}{R;S;\Gamma \vdash e : t}$$

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

h = System{enable "f1" in
Applet{(λx.
System{check "f1" then write x})
"kwijibo"}}

Then $P;S;\varnothing \vdash h : unit$ for any S

Example: Bad Code

g = System{enable "f1" in
Applet{(λx.
System{check "f2" then write x})
"kwijibo"}}

Then $R;S;\varnothing \vdash g : t$ is not derivable for any R,S, and t.

Static vs. Dynamic Checks

Calling this function requires the *static* permission p:

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

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)
  - Tse & Zdancewic – Run-time Principals in Information-flow Type Systems (IEEE S&P'04)