2004 프로그래밍언어연구회 여름학교

Current Techniques in Language-based Security

안 준 선

한국항공대학교

This is a revised version of "Current Techniques in Language-based Security" by Steve Zdancewic. (http://www.cs.uoregon.edu/activities/summerschool/summer04)

Map

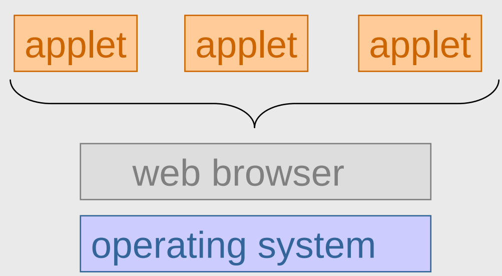
- Introducion
- Stack inspection
  - Java Security Model
  - Stack inspection
- Stack inspection from a PL perspective
  - Formalizing stack inspection : λ_{sec}
  - Translation to SPS : $\lambda_{sec} \rightarrow \lambda_{set}$
  - Type systems for stack inspection
- References

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 Collection
  - Eliminates memory management errors
- Library support
  - Cryptography, authentication, …

Mobile Code

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



- Principles of least privileges and complete mediation apply

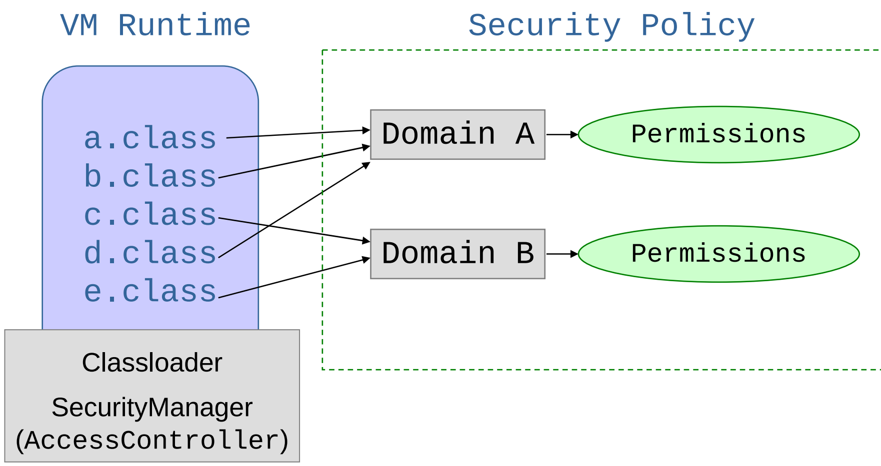
Access Control for Applets

- 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 ?

Map

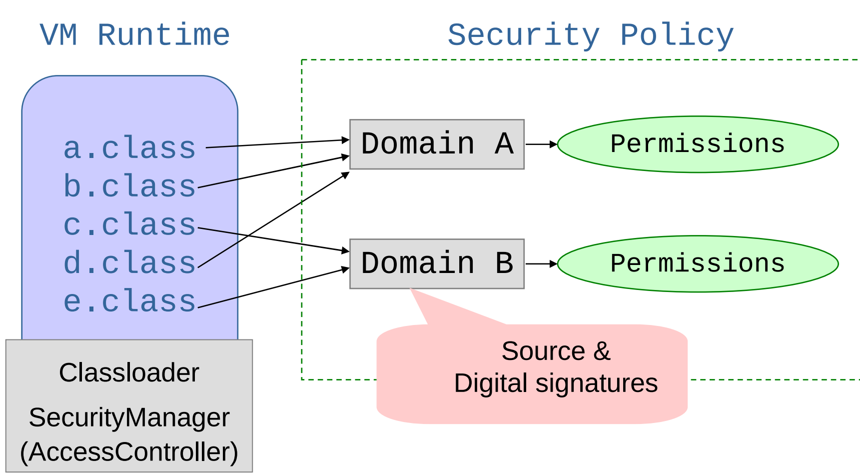
- Introducion
- Stack inspection
  - Java Security Model
  - Stack inspection
- Stack inspection from a PL perspective
  - Formalizing stack inspection : λ_{sec}
  - Translation to SPS : $\lambda_{sec} \rightarrow \lambda_{set}$
  - Type systems for stack inspection
- References

Java Security Model



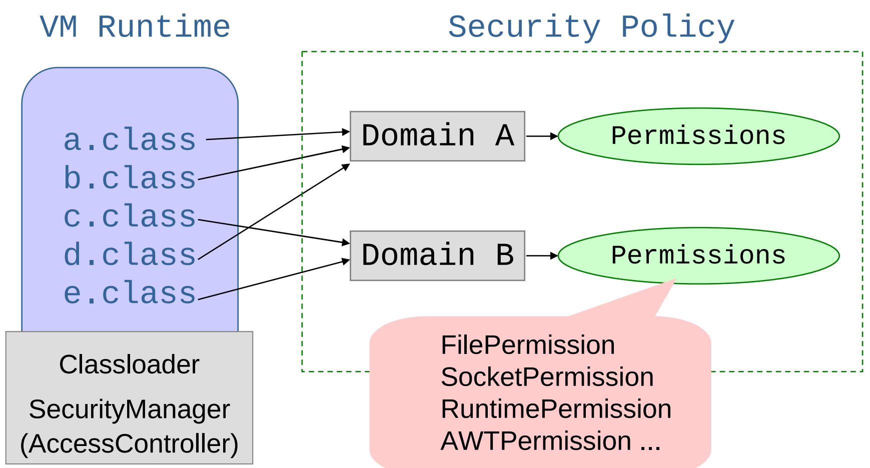
http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html

Java Security Model



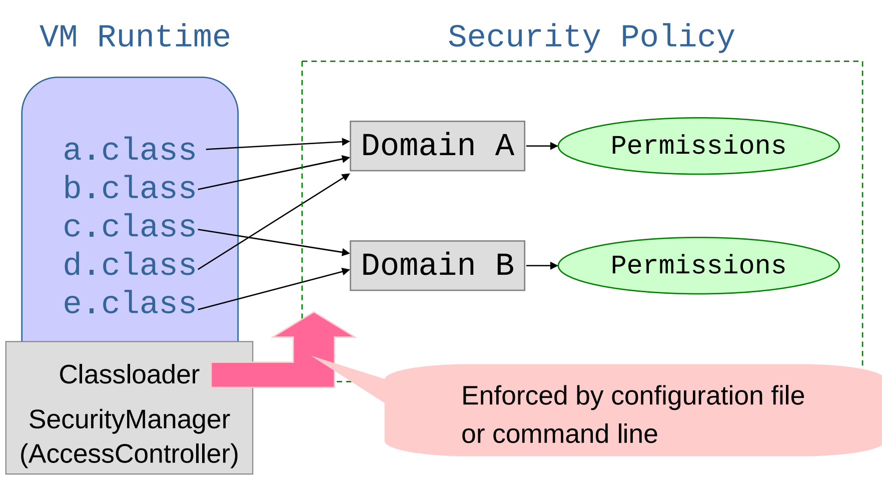
http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html

Java Security Model



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

Java Security Model



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

Stack Inspection Model

- 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 a protection domain not authorized for the privilege is encountered
  - succeed if privilege is enabled in a encountered frame
- Primitives
  - checkPermission(Permission)
  - enablePrivilege(Permission)

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 {
  static void run() {
    ...
    s.fileWrite(“/tmp/foo.txt”, “Hello!”);
    ...
    s.fileWrite(“/home/sigpl/important.tex”, “kwijibo”);
    ...
  }
}
```


Stack Inspection Example

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

Stack Inspection Example

Policy Database

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

fp

Stack Inspection Example

Policy Database

```
static 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 … */
```

```
static 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

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

```
static void run() {
 …
 s.fileWrite(“/tmp/foo.txt”, “Hello!”);
 …
}
```

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

Succeed!

Policy Database

Stack Inspection Example

```
static 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);
```

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

Fail

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

fp

Policy Database

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
}

  // 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)

Discussion : Stack Inspection

- Stack inspection seems appealing:
  - Fine grained, flexible, configurable policies
  - Distinguishes between code of varying degrees of trust
- But…
  - Policy is distributed throughout the software, and is not apparent from the program interfaces.
  - Semantics tied to the operational behavior of the program (defined in terms of stacks!)
    - How do we understand what the policy is?
    - How do we compare implementations
    - How do we validate transformations
  - Is it any good?

Map

- Introducion
- Stack inspection
  - Java Security Model
  - Stack inspection
- Stack inspection from a PL perspective
  - Formalizing stack inspection : λ_{sec}
  - Translation to SPS : $\lambda_{sec} \rightarrow \lambda_{set}$
  - Type systems for stack inspection
- References

λsec

- Call by value lambda calculus with security primitives
- Provides a contextual equivalence based on operational semantics
- Axioms for program transformation

λsec 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
```

Abstract Stack Inspection

- Abstract permissions

p,q ∈ P	Set of all permissions
R,S ⊆ P	Principals (sets of permissions)

- Models protection domains and permissions
- Examples:

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

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”} ⇓ fail
System{readFile “f2”} ⇓ <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[(λx.e) v]	→	E[e{v/x}]
E[`enable p in v`]	→	E[v]
E[R{v}]	→	E[v]
E[fail]	→	fail
E[test p then e else f]	→	E[e]
if Stack(E) ⊢ p		
E[test p then e else f]	→	E[f]
if ¬(Stack(E) ⊢ p)		

Stack Inspection

$e \Downarrow o$ iff $e \rightarrow^* o$

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")?

Stack(E') ⊦ fileWrite("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

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

irrelevant enable

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

check enable

Abstract Stack Inspection

$$. \vdash p$$

empty stack enables all

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

enable succeeds*

$$\frac{x \vdash p}{x.\text{enable}(q) \vdash p}$$

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' = (λx.e') e
ok = λx.x
loop = (λx.x x)(λx.x x) (note: loop $\not\Downarrow$)
f = λx. let z = x ok in λ_.z
g = λx. let z = x ok in λ_.(x ok)

Claim: f $\not\approx$ g

Proof:
Let C[ ] = $\varnothing${[ ] λ_.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 → …

Axiomatic Equivalence

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

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

Example: Tail Calls

Ordinary evaluation:
R{(λx.S{e}) v} → R{S{e{v/x}}}

Tail-call eliminated evaluation:
R{(λx.S{e}) v} → S{e{v/x}}

Not sound in general!

But OK if S ⊇ R

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})
→* System{Applet{fileHandler “f2” leak} okS}
→* System{Applet{System{System{
   λ_.System{leak (readFile “f2”)}}}} okS}
→* System{λ_.System{leak (readFile “f2”)} okS}
→* System{System{leak <f2 contents>}}
→* System{System{Applet{output <f2 contents>}}}
→* System{System{Applet{ok}}}
→* ok
```


Discussion : λ_{sec}

- 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)

Map

- Introducion
- Stack inspection
  - Java Security Model
  - Stack inspection
- Stack inspection from a PL perspective
  - Formalizing stack inspection : λ_{sec}
  - Translation to SPS : $\lambda_{sec} \rightarrow \lambda_{set}$
  - Type systems for stack inspection
- References

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
  check p then e           check perm.
  let x = e in f           local decl.
```

- check p then e ≡ test p then e else fail

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.

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"}

Translation Correctness

Theorem:

- If e →* v then [[e]] →* [[v]]
- If e →* fail then [[e]] →* fail
- Furthermore, if e diverges, so does [[e]].

Discussion : Translation

- 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!

Map

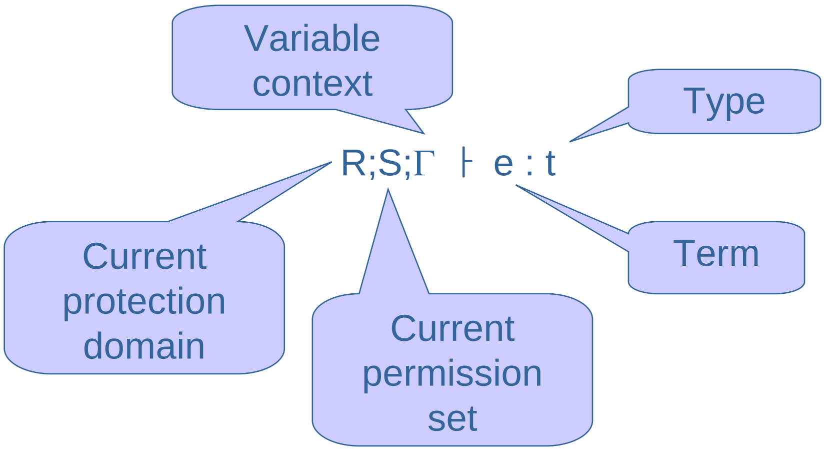
- Introducion
- Stack inspection
  - Java Security Model
  - Stack inspection
- Stack inspection from a PL perspective
  - Formalizing stack inspection : λ_{sec}
  - Translation to SPS : $\lambda_{sec} \rightarrow \lambda_{set}$
  - Type systems for stack inspection
- References

Types for Stack Inspection

- Type system for λ_{sec}
  - Statically detect security failures.
  - Eliminate redundant checks.
  - Example of nonstandard type system for enforcing security properties.

Typing Judgments

- Eager stack inspection judgment:



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:
  $t \xrightarrow{\{S\}} u$

Types

- Types:

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

Simple Typing Rules

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

Abstraction:

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

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;∅ ⊦ e : t then either e →* v or e diverges.
- In particular: e never fails. (i.e. check always succeeds)

Example: Good Code

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

Where System = {“f1”,”f2”, …}
  Applet = {“f1”}

Then P;S;∅ ⊦ h : unit for any S

Example: Good Code

System;((S∩System)U{"f1"}) ∩ Applet; {x→"kwijibo"} ├ write x: unit

System;((S∩System)U{"f1"})∩Applet∩System; {x→"kwijibo"} ├ check "f1" then write x: unit

Applet;((S∩System)U{"f1"})∩Applet; {x→"kwijibo"} ├ System{check "f1" then write x}: unit

Applet; ((S∩System)U{"f1"}∩Applet; ∅ ├ (λx.System{check "f1" then write x}) "kwijibo": unit

System;(S∩System)U{"f1"}; ∅ ├ Applet{(λx.System{check "f1" then write x}) "kwijibo"} : unit

System;S∩System; ∅ ├ enable "f1" in Applet{(λx.System{check "f1" then write x}) "kwijibo"} : unit

P;S; ∅ ├ System{enable "f1" in Applet{(λx.System{check "f1" then write x}) "kwijibo"}} : unit

Example: Bad Code

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

Then R;S;∅ ├ g : t is not derivable for any R,S, and t.

Example: Bad Code

System;((S∩System)U{“f1”})∩Applet; {x→”kwijibo”} ├ write x: unit

---

System;((S∩System)U{“f1”})∩Applet∩System; {x→”kwijibo”} ├ check “f2” then write x: unit

---

Applet;((S∩System)U{“f1”})∩Applet; {x→”kwijibo”} ├ System{check “f2” then write x}: unit

---

Applet; ((S∩System)U{“f1”}∩Applet; ∅ ├ (λx.System{check “f2” then write x}) “kwijibo”: unit

---

System;(S∩System)U{“f1”}; ∅ ├ Applet{(λx.System{check “f2” then write x}) “kwijibo”} : unit

---

System;S∩System; ∅ ├ enable “f2” in Applet{(λx.System{check “f2” then write x}) “kwijibo”} : unit

---

P;S; ∅ ├ System{enable “f2” in Applet{(λx.System{check “f1” then write x}) “kwijibo”}} : unit

Expressiveness

- This type system is very simple
  - No subtyping
  - No polymorphism
  - Hard to do inference
- Can add all of these features…
- See Francois’ paper for a nice example.
  - based on HM(X)

Map

- Introducion
- Stack inspection
  - Java Security Model
  - Stack inspection
- Stack inspection from a PL perspective
  - Formalizing stack inspection : λ_{sec}
  - Translation to SPS : $\lambda_{sec} \rightarrow \lambda_{set}$
  - Type systems for stack inspection
- References

References

- 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
- A Systematic Approach to Static Access Control
  François Pottier, Christian Skalka, Scott Smith
- Securing Java
  Gary McGraw and Edward W. Felten
- Inside Java 2 Platform Security
  L. Gong

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)