

Secure Information Flow and Stack-based Access Control for a Java-like Language

Anindya Banerjee

ab@cis.ksu.edu

Kansas State University

Joint work with David A. Naumann, Stevens Instt. of Technology

Secure Information Flow

Goal: Modular, static checking of security policies, e.g., confidentiality, for extensible software: no information flow from secret/classified input channels to public output channels (including covert channels).

Formalize for systems programmed in Java-like languages.

- ◆ System with **H**igh and **L**ow inputs, $L \leq H$.
H \equiv secret/private/classified
- ◆ **L** users permitted to see **L** outputs.

(Security Policy: Confidentiality \equiv “PROTECT SECRETS”)

Noninterference

- ◆ Noninterference (NI) [Goguen-Meseguer '82]

“No matter how **H** inputs change, **L** outputs remain same”.

≡ No information flow from **H** to **L**.

Example: Aliasing

```
class LPatient extends Object { //basic patient record
  String name;
  String getName() {return self.name;}
  unit setName(String n) {self.name := n;} }
```

```
class XPatient extends LPatient {
  String hiv; //SECRET
  String getHIV() {return self.hiv;}
  unit setHIV(String s) {self.hiv := s;} }
```

Example: Aliasing

```
LPatient lp := readFile();  
String LBuf := lp.getName(); String HBuf := lp.getName();
```

LBuf ~ lp.name ~ HBuf

```
XPatient xp := new XPatient(); xp.setName(LBuf);
```

LBuf ~ lp.name ~ HBuf ~ xp.name

```
String HBuf := readFromTrustedChannel(); xp.setHIV(HBuf);
```

HBuf ~ xp.hiv; LBuf ~ lp.name ~ xp.name

LBuf := HBuf; lp.setName(xp.getHIV())

xp.name ~ xp.hiv

Annotated Types Prevent Direct Data Flows

```
class LPatient extends Object {  
  (String, L) name;  
  (String, L) getName() {return self.name;}  
  (unit, L) setName((String, L) n) {self.name := n;} }
```

```
class XPatient extends LPatient {  
  (String, H) hiv; //SECRET  
  (String, H) getHIV() {return self.hiv;}  
  (unit, L) setHIV((String, H) s) {self.hiv := s;} }
```

Example: Aliasing revisited

```
(LPatient, L) lp := readFile();
```

```
(String, L) LBuf := lp.getName();
```

```
(String, H) HBuf := lp.getName();
```

```
(XPatient, L) xp := new XPatient(); xp.setName(LBuf:L);
```

```
(String, H) HBuf := readTrusted...; xp.setHIV(HBuf:H);
```

Example: Aliasing revisited

```
(LPatient, L) lp := readFile();
```

```
(String, L) LBuf := lp.getName();
```

```
(String, H) HBuf := lp.getName();
```

```
(XPatient, L) xp := new XPatient(); xp.setName(LBuf:L);
```

```
(String, H) HBuf := readTrusted...; xp.setHIV(HBuf:H);
```

```
LBuf:(String,L) := HBuf:(String,H)
```

```
lp.setName(xp.getHIV():(String,H))
```

Example: Implicit Control Flow (Conditional)

```
class XPatient extends LPatient { //hiv, getHIV, setHIV }

String leakStatus() {
    var String s;
    if (self.hiv) {s := 'YES';} else {s := 'NO'};
    return s;
}
```

Example: Implicit Control Flow (Conditional)

```
class XPatient extends LPatient { //hiv, getHIV, setHIV }
```

```
String leakStatus() {  
    var String s; //level of s???  
    if (self.hiv) {s := 'YES';} else {s := 'NO'};  
    return s;  
}
```

```
xp := new XPatient();    xp.setName(xp.leakStatus())
```

Example: Implicit Control Flow (Conditional)

```
class XPatient extends LPatient { //hiv, ...
  (String, H) leakStatus() {
    var (String, H) s;
    if (self.hiv) {s := 'YES';} else {s := 'NO'};
    return s;}
xp := new XPatient();    xp.setName(xp.leakStatus():H)
```

Example: Implicit Control Flow (Conditional)

```
class XPatient extends LPatient { //hiv, ...
  (String, H) leakStatus() {
    var (String, H) s;
    if (self.hiv) {s := 'YES';} else {s := 'NO';}
    return s;}
xp := new XPatient();    xp.setName(xp.leakStatus():H)
```

If guard is **H**, only **H**-variables and **H**-fields may be modified.

“No write down”

Example: Implicit Control Flow (Dynamic Dispatch)

```
class XPatient extends LPatient { //hiv, ...
```

```
class YN extends Object {(bool, L)val() {return true;}}  
class Y extends YN {(bool, L)val() {return true;}}  
class N extends YN {(bool, L)val() {return false;}}
```

Example: Implicit Control Flow (Dynamic Dispatch)

```
class XPatient extends LPatient { //hiv, ...
  (YN, H) leak() {
    var (YN, H) o;
    if (self.hiv) {o := new Y();} else {o := new N();}
    return o;}}

class YN extends Object {(bool, L)val() {return true;}}
class Y extends YN {(bool, L)val() {return true;}}
class N extends YN {(bool, L)val() {return false;}}
```

Example: Implicit Control Flow (Dynamic Dispatch)

```
class XPatient extends LPatient { //hiv, ...
  (YN, H) leak(){
    var (YN, H) o;
    if (self.hiv) {o := new Y();} else {o := new N();}
    return o;}}
xp.leak() : (YN, H);

class YN extends Object {(bool, L)val() {return true;}}
class Y extends YN {(bool, L)val() {return true;}}
class N extends YN {(bool, L)val() {return false;}}
```

Example: Implicit Control Flow (Dynamic Dispatch)

```
class XPatient extends LPatient { //hiv, ...
  (YN, H) leak(){
    var (YN, H) o;
    if (self.hiv) {o := new Y();} else {o := new N();}
    return o;}}
xp.leak() : (YN, H);    xp.leak().val() : (bool, ???)

class YN extends Object {(bool, L)val() {return true;}}
class Y extends YN {(bool, L)val() {return true;}}
class N extends YN {(bool, L)val() {return false;}}
```

Example: Implicit Control Flow (Dynamic Dispatch)

```
class XPatient extends LPatient { //hiv, ...  
  (YN, H) leak(){...}  
}
```

- ◆ `xp.leak()` : (YN, H)
- ◆ `xp.leak().val()` : (bool, H)

If level of receiver **H**, level of returned result from method call **H**.

Leaks via aliasing in field update

Assume XPatient has L field bloodGroup.

```
(string, L) test((bool, H) g) {
  (XPatient, L) xp1 := new XPatient;
  (XPatient, L) xp2 := new XPatient;
  (XPatient, H) hp;
  (String, L) bg := xp1.bloodGroup;
  //initial value of xp1's bloodGroup
  if g then hp := xp1 else hp := xp2; // aliasing
  hp.bloodGroup := "Z";
  if bg = xp1.bloodGroup then
    result := "no" else result := "yes"; // g is leaked}
```

Towards a formalization: observations

- ◆ An object may be aliased by L-var, H-var. But L cannot read H variable/field.
- ◆ In conditionals/dyn. dispatch/field update, assignment may be confined to H-vars, H-fields.

Thus input states, output states *indistinguishable* by L.

Checking information flow by typing

Data types: $T ::= \mathbf{unit} \mid \mathbf{bool} \mid C$ Levels: $\kappa ::= L \mid H$

Expression types: (T, κ) means that value is $\leq \kappa$

Commands: $(\mathbf{com} \kappa_1, \kappa_2)$ assigns to vars $\geq \kappa_1$, to fields $\geq \kappa_2$

Typings (in context Δ): $\Delta \vdash e : (T, \kappa)$ $\Delta \vdash S : (\mathbf{com} \kappa_1, \kappa_2)$

Assignment rule: if $x : (C, \kappa_1) \vdash e : (T, \kappa_2)$

and $\kappa_2 \leq \kappa_1$ then $x : (C, \kappa_1) \vdash x := e : (\mathbf{com} \kappa_1, H)$

Conditional rule: if $\Delta \vdash e : (\mathbf{bool}, \kappa_1)$ and

$\Delta \vdash S_i : (\mathbf{com} \kappa_2, \kappa_2)$ and $\kappa_1 \leq \kappa_2$ then

$\Delta \vdash \mathbf{if} \ e \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 : (\mathbf{com} \kappa_2, \kappa_2)$

Noninterference theorem (“Rules enforce policy”): typability implies that **L**ow outputs do not depend on **H**igh inputs

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system
 - ◆ Data flow (via mutable fields)

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system
 - ◆ Data flow (via mutable fields)
 - ◆ Control flow (via dynamic dispatch)

Our Contribution

Type-based analysis for secure information flow.

- ◆ Sequential, Java-like language
- ◆ Security type system
 - ◆ Data flow (via mutable fields)
 - ◆ Control flow (via dynamic dispatch)
- ◆ Proof of Noninterference (denotational semantics, compositional proofs, machine checked proofs)

Runtime Access Control

- ◆ Observation: extensible software implemented in Java associate *permissions* (“rights”) with code to prevent run-time security errors.
- ◆ Focus: implementations (not protocol designs) involving mobile code, subclassing, pointers —constrained by types, scope, and *runtime access control*
- ◆ Question: How to connect access control mechanism (used widely) and information flow analysis (often restrictive)?

Access control by “stack inspection”

Local policy assigns *static permissions* to classes (based on code origin: local disk, signed download, etc).

When untrusted code calls trusted code, latter must execute with “right” permissions – dependent on permissions of untrusted code.

Run-time permissions computed/checked using run-time stack.

test p **then** S_1 **else** S_2 executes S_1 only if the class for “each frame on stack” has p in its static permissions.

enable p **in** S limits test to stack frames up to one that explicitly enabled p .

Eager semantics: security context parameter, the set of current permissions (updated by **enable** and call).

Example: Permissions

```
class Sys { // static permissions chpass, wpass
  unit writepass(string x){
    test wpass // access guard to protect integrity
    then nativeWrite(x,"passfile") else abort }
  unit passwd(string x){
    test chpass then enable wpass in writepass(x)
    else abort }}

class User { // static permission chpass (but not wpass)
  Sys s:= ... ;
  unit use(){ enable chpass in s.passwd("mypass") } // ok
  unit try(){ enable wpass in s.writepass("mypass") }} // aborts
```

Info release vs. Info flow

```
class Sys { // static permissions rdkey
  int readKey(){ // policy: confidential key
    test rdkey then result:= nativeReadKey() else abort }
  int trojanHorse(){
    enable rdkey in int x:= readKey();
    if (x mod 2) > 0 then result := 0 else result := 1 }}
class PlugIn { // no static permissions
  Sys s:= ...;
  int output; // policy: public
  unit tryToSteal(){ output:= s.readKey() } // aborts
  unit steal(){ output:= s.trojanHorse() }} // leak
```

Security types specify/check policy

```
class Sys { // static permissions rdkey
  int readKey(){ // security type:  $\bullet \rightarrow H$ 
    test rdkey then result:= nativeReadKey() else abort }
  int trojanHorse(){ // security type:  $\bullet \rightarrow H$ 
    enable rdkey in int x:= readKey();
    if (x mod 2) > 0 then result := 0 else result := 1 }}
class PlugIn { // no static permissions
  Sys s:= ...;
  int output; // security type: L
  unit tryToSteal(){ output:= s.readKey() } // aborts
  unit steal(){ output:= s.trojanHorse() }} // illegal flow H to L
```

Selective release for trusted clients

```
class Kern { // static permissions stat,sys
  string infoH; // security type H
  string infoL; // security type L
  string getHinfo(){ // security type  $\bullet \rightarrow H$ 
    test sys then result:= self.infoH else abort }
  string getStatus(){ // security type  $\bullet \rightarrow ???$ 
    /* trusted, untrusted callers may both use getStatus */
    test stat // selective release of info
    then enable sys in result:= self.getHinfo()
    else result:= self.infoL }... }
```

Usual info. flow analysis restrictive – getStatus: $\bullet \rightarrow H$.

Want: *no stat* then getStatus: $\bullet \rightarrow L$, *o.w.*, getStatus: $\bullet \rightarrow H$.

```

class Comp1 { // untrusted: static permission other
  Kern k:=...;
  string v; // security type L
  string status(){ // security type  $\bullet \rightarrow L$ 
    result:= self.v ++ k.getStatus() } // gets infoL

  string status2(){ //  $\bullet \rightarrow L$ 
    enable stat in result:= self.v ++ k.getStatus() } // gets infoL

class Comp2 { // partially trusted: static permissions stat,other
  Kern k:=...;
  string statusH(){ //  $\bullet \rightarrow H$ 
    enable stat in result:= k.getStatus() }} // gets infoH

```

Our approach

Notation $\kappa \xrightarrow{P} \kappa_2$ for method type means: when called with argument with level $\leq \kappa$, type of result $\leq \kappa_2$ provided caller does *not* have the permissions in set P .

```
string getStatus() { // both  $\bullet \xrightarrow{\{stat\}} L$  and  $\bullet \xrightarrow{\emptyset} H$ 
```

```
  test stat
```

```
  then enable sys in result := self.getHinfo()
```

```
  else result := self.infoL }
```

```
class Comp1 { // static permission other
```

```
  ... result := k.getStatus() // ok, using Kern.getStatus:  $\bullet \xrightarrow{\{stat\}} L$ 
```

```
  ... enable stat in result := k.getStatus() // ok, using  $\bullet \xrightarrow{\{stat\}} L$ 
```

Technical details

Typing Judgements:

$\Delta; P \vdash e : (T, \kappa)$ // Δ security type context

$\Delta; P \vdash S : (\text{com } \kappa_1, \kappa_2)$

In security context Δ , expression e has type (T, κ) when permissions *disjoint from P are enabled*, i.e., P is upper bound of excluded permissions.

Checking test

Consider $\Delta; P \vdash \mathbf{test} P' \mathbf{then} S_1 \mathbf{else} S_2 : com$

If $P' \cap P \neq \emptyset$ then test *must fail* at run time. Check S_2 :

$\Delta; P \vdash S_2 : com$

If $P' \cap P = \emptyset$ then test *may succeed*. Check both S_1 and S_2 :

$\Delta; P \vdash S_1 : com$ and $\Delta; P \vdash S_2 : com$

Checking test in getStatus

```
string getStatus() { // both  $\bullet \xrightarrow{\{stat\}} L$  and  $\bullet \xrightarrow{\emptyset} H$   
  test stat  
  then enable sys in result:= self.getHinfo()  
  else result:= self.infoL }
```

$\Delta; \emptyset \vdash$ **test** *stat* **then** ...

$\{stat\} \cap \emptyset = \emptyset$, so analyze both branches of **test**.

$\Delta; \{stat\} \vdash$ **test** *stat* **then** ... **else** result:= self.infoL

$\{stat\} \cap \{stat\} = \{stat\}$, so analyze **else** branch.

Note that at run-time only result:= self.infoL is relevant.

Trusted calling Untrusted

```
class NaiveProgram extends Object { // all permissions, R  
  unit Main() {  
    string s := BadPlugIn.TempFile();  
    File.Delete(s); }  
}
```

- (1) body of TempFile executed with $R \cap \text{Perms}(\text{BadPlugIn})$
- (2) File.Delete(s) executed with **R**

```
class BadPlugIn extends Object { // no FileIO  
  string TempFile() { result := "...//tmp/password ..."; }  
}
```

```
class File extends Object { // R  
  unit Delete(string s) { test FileIO then Win32.Delete(s) else abort; }  
}
```

Integrity

```
class NaiveProgram extends Object { // R
  unit Main() {
    string s := BadPlugIn.TempFile(); //s : H
    File.Delete(s); }
}
```

```
class BadPlugIn extends Object { // no FileIO
  string TempFile() { result := "...//tmp/password ..." } } //• $\xrightarrow{\emptyset}$ H
```

File.Delete has both type $H^{\{FileIO\}} \rightarrow \bullet$ and type $L \xrightarrow{\emptyset} \bullet$.

Can show that Main's body is not well-typed.

History-based Access Control

```
class NaiveProgram extends Object { // all permissions,  $\mathcal{R}$ 
  unit Main() {
    string s := BadPlugIn.TempFile();
    File.Delete(s); } }
```

- (1) BadPlugIn.Tempfile() called with \mathcal{R}
- (2) body of TempFile executed with $\mathcal{R} \cap \text{Perms}(\text{BadPlugIn})$
- (3) File.Delete(s) executed with $\mathcal{R} \cap \text{Perms}(\text{BadPlugIn})$

$\text{FileIO} \notin \mathcal{R} \cap \text{Perms}(\text{BadPlugIn})$. Hence **test** *FileIO* **then ...**
fails the test.

Security Typing

Notation $\kappa \xrightarrow{P;Q} \kappa_2$ for method type means:

- (1) caller's permissions are disjoint from P
- (2) on return, permissions for rest of computation disjoint from Q .

Similarly use $\Delta; P \vdash S : \text{com}; Q$:

- (1) excluded permissions before S is P
- (2) excluded permissions after S is Q .

To check $\Delta; P \vdash S_1; S_2 : \text{com}; Q$ there must exist Q_1 such that $\Delta; P \vdash S_1 : \text{com}; Q_1$ and $\Delta; Q_1 \vdash S_2 : \text{com}; Q$.

Noninterference theorem

Theorem: If a command (or complete class table) satisfies the security typing rules then it is noninterfering.

Noninterfering command: Suppose $\Delta; P \vdash S : \text{com}$.

Let heaps h, h' and stores η, η' be indistinguishable by L (written $h \sim h'$ and $\eta \sim \eta'$) and suppose $Q \cap P = \emptyset$.

Let $(h_0, \eta_0) = \llbracket \Delta^\dagger \vdash S \rrbracket (h, \eta) Q$ and $(h'_0, \eta'_0) = \llbracket \Delta^\dagger \vdash S \rrbracket (h', \eta') Q$.

Then $\eta_0 \sim \eta'_0$ and $h_0 \sim h'_0$.

Sequential language with pointers, mutable state, private fields, class-based visibility, dynamic binding & inheritance, recursive classes, casts & type tests, access control.

Related work: Stack Inspection

- ◆ Li Gong (1999): documents stack inspection for Java and how method call follows the principle of least privilege.
- ◆ Wallach, Appel, Felten (2000): describe stack inspection in terms of ABLP logic for access control.
- ◆ Pottier, Skalka, Smith (2000 –): Static analysis for access checks that never fail. Basis for program optimizations.
- ◆ Fournet, Gordon (2002): Comprehensive study of stack inspection and program optimizations permitted by stack inspection.
- ◆ Abadi, Fournet (2003): Protection of trusted callers calling untrusted code.

Related work: Information Flow

- ◆ Noninterference: Goguen-Meseguer, Denning-Denning
- ◆ Type-based analyses for information flow:
 - 1996– Smith, Volpano (Simple Imperative Language)
 - 1999– Abadi et al. (DCC – Info. flow as dependence analysis)
 - 1999– Sabelfeld, Sands (Threads, Poss. NI, Prob. NI)
 - 1999 Myers (Java – but NI open)
 - 2000– Pottier, Simonet, Conchon (Core ML)
- ◆ Sabelfeld and Myers: survey on language-based information flow security (2003).

Related work: Access Control and Information Flow

- ◆ Rushby: Access control \equiv assigning levels to variables. Proof and mechanical checking of noninterference.
- ◆ Heintze and Riecke (SLam); Pottier and Conchon (Core ML): Static access control – access labels have no run-time significance.
- ◆ Stoughton (1981): Dynamic access control and information flow together in a simple imperative language with semaphores. However, no formal results are proven.

Deployment

- ◆ Interface specs. need to express security policies of interest.
- ◆ Method typings are one way
 - ◆ Specify access and information flow policies

Given: information flow policies for fields and methods.

Checker based on our analysis can find bugs — policy not satisfied.

Conclusion

- ✓ static enforcement of noninterference (Smith& Volpano)
- ✓ account for runtime access control (Hennessy&Riely for async pi calculus)
- ✓ handles pointers, subclassing & dynamic bind (Myers)
- ✓ suggests permission-aware interface specs
- ✗ not all covert channels
- ✗ no declassification (Myers&Zdancewic)
- ◆ need more examples of security-aware programs

Ongoing and future work

- ◆ machine checking proofs: proof of the main noninterference theorem formalized in PVS (Dave).
- ◆ type inference (ongoing with Qi Sun and Dave Naumann); polymorphism & threads
- ◆ explaining insecure information flow (ongoing with Torben Amtoft)
- ◆ more general access control policies and global security properties they guarantee.
- ◆ security for web services.

Checking method declarations

Recap: Security type $\kappa \xrightarrow{P} \kappa_2$ means that if $\text{args} \leq \kappa$ and caller permissions disjoint from P then $\text{result} \leq \kappa_2$.

To check

$$C \vdash T \text{ m}(\text{U } x)\{ S \} // \text{mtype}(\text{m}, C) = \text{U} \rightarrow T$$

we must check, for all $(\kappa \xrightarrow{P} \kappa_2) \in \text{smtypes}(\text{m}, C)$, that

$$\Delta; (P \cap \text{Perms}(C)) \vdash S : \text{com}$$

where $\Delta = x : (\text{U}, \kappa), \text{self} : (C, \kappa_0), \text{result} : (T, \kappa_2)$

Checking getStatus

```
string getStatus() { // both  $\bullet \xrightarrow{\{stat\}} L$  and  $\bullet \xrightarrow{\emptyset} H$   
  test stat  
  then enable sys in result := self.getHinfo()  
  else result := self.infoL }
```

For $\bullet \xrightarrow{\emptyset} H$: result : H ; $\emptyset \vdash$ **test *stat*** then ...
(N.B. $\emptyset \cap Perms(Kern) = \emptyset$)

For $\bullet \xrightarrow{\{stat\}} L$: result : L ; $\{stat\} \vdash$ **test *stat*** then ...
(N.B. $\{stat\} \cap Perms(Kern) = \{stat\}$)

Subclass of Kern: overriding getStatus

```
class Kern { // static permissions stat,sys
  string getHinfo(){
    test sys then result:= self.infoH else abort }...}
class SubKern extends Kern { // no static permissions
  string getStatus(){ // override
    enable sys // no effect
    in result:= self.getHinfo() }
```

$smtypes(\text{getStatus}, \text{SubKern}) = smtypes(\text{getStatus}, \text{Kern})$

For $\bullet \xrightarrow{\{stat\}} L: \text{result} : L; \emptyset \vdash \text{enable } sys \text{ in } \dots$

(N.B. $\{stat\} \cap Perms(\text{SubKern}) = \emptyset$)

Checking method calls

For method call

$$\Delta; P \vdash x := e.m(f) : \text{com}$$

suppose static type of e is D .

Call is allowed if for some $\kappa \xrightarrow{P'} \kappa_3 \in \text{smtypes}(m, D)$:

$$P' \cap \text{Perms}(\Delta^\dagger \text{self}) \subseteq P$$

The complete rules include subsumption on levels.

Method call in history-based access control

If S is $x := e.m(f)$, call is allowed if for some

$\kappa \xrightarrow{P';Q'} \kappa_3 \in \text{smtypes}(m, D): P' \cap \text{Perms}(\Delta^\dagger \text{self}) \subseteq P$ and
 $Q \subseteq Q' \cap \text{Perms}(\Delta^\dagger \text{self})$.