Using Access Control for Secure Information Flow in a Java-like Language

Anindya Banerjee and David A. Naumann

ab@cis.ksu.edu, naumann@cs.stevens-tech.edu

Kansas State University and Stevens Institute of Technology

www.cis.ksu.edu/~ab,

www.cs.stevens-tech.edu/~naumann

Problem

- Goal: Modular, static checking of security policies, e.g., confidentiality, for extensible software: no information flow from High input channels to Low output channels (including some covert channels)

Problem

- Goal: Modular, static checking of security policies, e.g., confidentiality, for extensible software: no information flow from High input channels to Low output channels (including some covert channels)
- Focus: implementations (not protocol designs) involving mobile code, subclassing, pointers —constrained by types, scope, and *runtime access control*
- Observation: extensible software implemented in Java associate *permissions*(“rights”) with code to prevent run-time security errors.

Problem

- Goal: Modular, static checking of security policies, e.g., confidentiality, for extensible software: no information flow from High input channels to Low output channels (including some covert channels)
- Focus: implementations (not protocol designs) involving mobile code, subclassing, pointers —constrained by types, scope, and *runtime access control*
- Observation: extensible software implemented in Java associate *permissions*("rights") with code to prevent run-time security errors.
- 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.

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.

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.

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 enabled and granted 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 }}
```

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: untrusted
   unit tryToSteal(){ output:= s.readKey() } // aborts
   unit steal(){ output:= s.trojanHorse() }} // leak
```

Security types specify/check policy

class Sys { // static permissions *rdkey*

int readKey(){ // policy annotation: $\bullet \rightarrow \mathrm{H}$

test *rdkey* **then** result:= nativeReadKey() **else abort** }

int trojanHorse(){ // policy annotation: $\bullet \rightarrow \mathrm{H}$

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; // policy annotation: L

unit tryToSteal(){ output:= s.readKey() } // aborts

unit steal(){ output:= s.trojanHorse() }} // illegal flow H to L

Checking information flow by typing

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

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

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

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

Checking information flow by typing

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

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

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

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

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

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

Checking information flow by typing

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

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

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

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

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

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

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

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

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

Noninterference theorem (“Rules enforce policy”): typability implies that Low outputs do not depend on High inputs

Examples of security typing

class PlugIn { // no static permissions

  int output; // policy annotation: $\mathtt{L}$

  unit steal(){ output:= s.trojanHorse() }

Assignment rule requires trojanHorse: $\bullet \rightarrow \mathtt{L}$.

Examples of security typing

class PlugIn { // no static permissions

int output; // policy annotation: L

unit steal(){ output:= s.trojanHorse() }

Assignment rule requires trojanHorse: $\bullet \longrightarrow \mathrm{L}$.

class Sys { // static permissions *rdkey*

int trojanHorse(){

enable *rdkey* **in int** x:= readKey();

if (x **mod** 2) $>$ 0 **then** result := 0 **else** result := 1 }}

Conditional rule requires result: H, hence trojanHorse: $\bullet \longrightarrow \mathrm{H}$.

Selective release for trusted clients

```
class Kern { // static permissions stat,sys
  private string infoH; // policy H
  private string infoL; // policy L
  string getHinfo(){ // type • → H
    test sys then result:= self.infoH else abort }
  string getStatus(){ // type • → ???
     /* 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: $\bullet \to \mathrm{H}$.

Want: *no stat* then getStatus: $\bullet \to \mathrm{L}$, o.w., getStatus: $\bullet \to \mathrm{H}$.

class Vend1 { // untrusted: static permission *other*

Kern k:=. . . ;

private string v; // policy L

string status(){ // policy • → L

result:= self.v ++ k.getStatus() } // gets infoL

```
class Vend1 { // untrusted: static permission other
  Kern k:=. . . ;
  private string v; // policy L
  string status(){ // policy • → L
    result:= self.v ++ k.getStatus() } // gets infoL

  string status2(){ // • → L
    enable stat in result:= self.v ++ k.getStatus() } // gets infoL
```

```
class Vend1 { // untrusted: static permission other
  Kern k:=...;
  private string v; // policy L
  string status(){ // policy • → L
    result:= self.v ++ k.getStatus() } // gets infoL

  string status2(){ // • → L
    enable stat in result:= self.v ++ k.getStatus() } // gets infoL

class Vend2 { // partially trusted: static permissions stat,other
  Kern k:=...;
  string statusH(){ // • → H
    enable stat in result:= k.getStatus() }} // gets infoH
```

Our approach

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

Our approach

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

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

  test *stat*

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

  else result:= self.infoL }

class Vend1 { // static permission *other*

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

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

```
string getStatus(){ // both •--{stat}-->L and •--∅-->H
  test stat
  then enable sys in result:= self.getHinfo()
  else result:= self.infoL }

class Vend2 { // static permissions stat,other
  string statusH(){ // •--∅-->H
    enable stat in result:= k.getStatus() }}
```

Ok using getStatus: $\bullet \xrightarrow{\varnothing} \mathrm{H}$, but not using getStatus: $\bullet \xrightarrow{\{stat\}} \mathrm{L}$.

Technical details

Typing Judgements:

$$\Delta; P \vdash e : (T, \kappa) \quad \text{//}\Delta \text{ security type context}$$

$$\Delta; P \vdash S : \mathrm{com}$$

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.

Notation $\Delta^{\dagger}$ for Δ with security annotations erased.

Checking method declarations

Recap: Security type $\kappa \xrightarrow{P} \kappa_2$ means that if args $\leq \kappa$ and caller permissions disjoint from P then result $\leq \kappa_2$. Method may have multiple security types (c.f., getStatus).

To check

$$C\ \kappa_0 \vdash T\ m(U\ x)\{\ S\ \}\ //\mathit{mtype}(m, C) = U \to T$$

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

$$\Delta;\ (P \cap \mathit{staticPerms}(C)) \vdash S : \mathit{com}$$

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

Checking getStatus

string getStatus()$\{$ // both $\bullet \xrightarrow{\{\textit{stat}\}} \mathrm{L}$ and $\bullet \xrightarrow{\varnothing} \mathrm{H}$

test *stat*

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

else result:= self.infoL $\}$

For $\bullet \xrightarrow{\varnothing} \mathrm{H}$: $\mathsf{result} : \mathrm{H}\ ;\ \varnothing \vdash$ **test** *stat* **then** $\ldots$

(*N.B.* $\varnothing \cap \mathsf{staticPerms}(\mathsf{Kern}) = \varnothing$)

Checking getStatus

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

test *stat*

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

else result:= self.infoL }

For $\bullet \xrightarrow{\varnothing} \mathrm{H}$: $\mathrm{result} : \mathrm{H}\ ;\ \varnothing \vdash$ **test** *stat* **then** ...

(*N.B.* $\varnothing \cap \mathrm{staticPerms}(\mathrm{Kern}) = \varnothing$)

For $\bullet \xrightarrow{\{stat\}} \mathrm{L}$: $\mathrm{result} : \mathrm{L};\ \{stat\} \vdash$ **test** *stat* **then** ...

(*N.B.* $\{stat\} \cap \mathrm{staticPerms}(\mathrm{Kern}) = \{stat\}$)

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(getStatus, SubKern) = *smtypes*(getStatus, Kern)

For $\bullet \xrightarrow{\{stat\}} \mathrm{L}$: $\mathsf{result} : \mathrm{H}\ ;\ \varnothing \vdash$ **test** *stat* **then** ...

(*N.B.* $\{stat\} \cap \mathsf{staticPerms}(\mathsf{SubKern}) = \varnothing$)

Checking access control operations

If $\Delta; (P - (P' \cap \mathsf{staticPerms}(\Delta(self)))) \vdash S : \mathsf{com}$
then $\Delta; P \vdash$ **enable** P' **in** $S : \mathsf{com}$

Checking access control operations

If $\Delta; (P - (P' \cap \mathrm{staticPerms}(\Delta(self)))) \vdash S : \mathrm{com}$
then $\Delta; P \vdash$ **enable** P' **in** $S : \mathrm{com}$

Simple test rule:

If $P' \cap P = \varnothing$ (so test *may* succeed)
and $\Delta; P \vdash S_1 : \mathrm{com}$ and $\Delta; P \vdash S_2 : \mathrm{com}$ then
$\Delta; P \vdash$ **test** P' **then** S_1 **else** $S_2 : \mathrm{com}$

Checking access control operations

If $\Delta; (\mathsf{P} - (\mathsf{P}' \cap \mathsf{staticPerms}(\Delta(self)))) \vdash S : \mathsf{com}$
then $\Delta; \mathsf{P} \vdash$ **enable** P' **in** $S : \mathsf{com}$

Simple test rule:

If $\mathsf{P}' \cap \mathsf{P} = \varnothing$ (so test *may* succeed)
and $\Delta; \mathsf{P} \vdash S_1 : \mathsf{com}$ and $\Delta; \mathsf{P} \vdash S_2 : \mathsf{com}$ then
$\Delta; \mathsf{P} \vdash$ **test** P' **then** S_1 **else** $S_2 : \mathsf{com}$

Key rule - tests that *must* fail:

If $\mathsf{P}' \cap \mathsf{P} \neq \varnothing$ and $\Delta; \mathsf{P} \vdash S_2 : \mathsf{com}$
then $\Delta; \mathsf{P} \vdash$ **test** P' **then** S_1 **else** $S_2 : \mathsf{com}$

Checking getStatus in Kern

string getStatus(){ // both $\bullet \xrightarrow{\{\mathit{stat}\}} \mathrm{L}$ and $\bullet \xrightarrow{\varnothing} \mathrm{H}$

test *stat*

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

else result:= self.infoL }

For $\bullet \xrightarrow{\varnothing} \mathrm{H}$: $\mathtt{result} : \mathrm{H}\ ;\ \varnothing \vdash$ **test** *stat* **then** ...

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

Checking getStatus in Kern

string getStatus(){ // both $\bullet \xrightarrow{\{\mathit{stat}\}} \mathrm{L}$ and $\bullet \xrightarrow{\varnothing} \mathrm{H}$

test *stat*

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

else result:= self.infoL }

For $\bullet \xrightarrow{\varnothing} \mathrm{H}$: $\mathfrak{result} : \mathrm{H}\ ;\ \varnothing \vdash$ **test** *stat* **then** ...

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

For $\bullet \xrightarrow{\{\mathit{stat}\}} \mathrm{L}$: $\mathfrak{result} : \mathrm{L};\ \{\mathit{stat}\} \vdash$ **test** *stat* **then** ...

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

Thus only result:= self.infoL is relevant.

Noninterference theorem

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

Noninterference theorem

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

Safe command: Suppose $\Delta; P \vdash S : \mathrm{com}$.

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

Let $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S]\!](h, \eta)Q$ and $(h_0', \eta_0') = [\![\Delta^\dagger \vdash S]\!](h', \eta')Q$.

Then $\eta_0 \sim \eta_0'$ and $h_0 \sim h_0'$.

Noninterference theorem

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

Safe command: Suppose $\Delta; P \vdash S : \mathsf{com}$.

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

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

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)
2002–	Banerjee and Naumann (fragment of Java)
2003–	Sabelfeld and Myers (survey)

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.

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 API specs
- ✘ not all covert channels
- ✘ no declassification (Myers&Zdancewic)
- ✘ protection of caller (Abadi, Fournet)
- ♦ need more examples of security-aware programs

Future work

- case studies in extensible applications for ad hoc nets (dance performance, physical therapy, sports training): attacks on Bluetooth authentication via applications with minimal security requirements
- inference (ongoing); polymorphism & threads
- optimizing transformations (cf. Fournet&Gordon)
- connections with parametricity; with dependent types; with declassification
- machine checking our proofs