

---

# Run-time Principals in Information-flow Type Systems

Stephen Tse      Steve Zdancewic

University of Pennsylvania

IEEE Symposium on Security and Privacy, May 2004

---

# Building secure software

---

**Goal:** help programmers build secure software that interacts with existing *security infrastructure* such as authentication, file permissions, cryptography...

# Building secure software

---

**Goal:** help programmers build secure software that interacts with existing *security infrastructure* such as authentication, file permissions, cryptography...

## Challenges:

1. Increasingly large and distributed *applications*

# Building secure software

---

**Goal:** help programmers build secure software that interacts with existing *security infrastructure* such as authentication, file permissions, cryptography...

## Challenges:

1. Increasingly large and distributed *applications*
2. Increasingly complex and refined *policies*:  
type safety, secrecy, integrity

# Building secure software

---

**Goal:** help programmers build secure software that interacts with existing *security infrastructure* such as authentication, file permissions, cryptography...

## Challenges:

1. Increasingly large and distributed *applications*
2. Increasingly complex and refined *policies*:  
type safety, secrecy, integrity
3. Existing mechanisms (e.g. PKI, OS) are crucial but higher levels of *abstraction* are needed

# Language-based security

---

**Our approach:** design type systems for specifying high-level policies and providing strong static guarantees

**Long History:** Lattice model [Denning, 1976], Jif [Myers et. al, 1998], Flow Caml [Pottier and Simonet, 2003], ...

# Language-based security

---

**Our approach:** design type systems for specifying high-level policies and providing strong static guarantees

**Long History:** Lattice model [Denning, 1976], Jif [Myers et. al, 1998], Flow Caml [Pottier and Simonet, 2003], ...

1. Programmers specify policies as *type annotations*
2. Compilers automatically catch *illegal info flows*

```
String{H} password = "secret";  
void print (String{L} x) {...}
```

*Connect infrastructures and languages?*

# Compile time vs. run time

---

- Most existing security languages specify policies known at *compile time*

```
String{H} password = "secret";  
void print (String{L} x) {...}
```

# Compile time vs. run time

---

- Most existing security languages specify policies known at *compile time*

```
String{H} password = "secret";  
void print (String{L} x) {...}
```

- But existing infrastructures express constraints in terms of *principals* known only at *run time*
  1. authentication: *public key* from PKI
  2. file permissions: *user id* from OS

# Compile time vs. run time

---

*How to interact with run-time systems?*

1. determine run-time principals from run-time systems
2. specify security policies in term of such principals
3. connect compile-time principals (e.g. `root`) with run-time principals (e.g. `user`)

```
principal user = Runtime.getUser();  
void print (String{user:} x) {...}  
void printroot (String{root:} y) {  
    if (user == root) print(y); }  
}
```

# Technical challenges

---

1. Formalize typing and evaluation of such a language
2. Prove the soundness and noninterference theorems
3. Allow downgrading information
4. Implement over a distributed system

# Language features

---

	Expression e		Type t
principal		root	: #root

The expression `root` is a principal constant, known at *compile time*, and has the *principal type* `#root`.

# Language features

---

	Expression e	Type t
principal	root	: #root
variable	id	: #user

The expression `id` is a principal variable (a first-class construct), known at *run time*, and has type `#user`.

# Language features

---

	Expression e	Type t
principal	root	#root
variable	id	#user

The expression `id` is a principal variable (a first-class construct), known at *run time*, and has type `#user`.

Universal and existential polymorphism:

1. `void print<user> (#user id, int{user:} x) { ... }`
2. `#user id = Runtime.getUser();`

# Language features

---

	Expression $e$	Type $t$
principal	<code>root</code>	<code>#root</code>
variable	<code>id</code>	<code>#user</code>
delegation	<code>if (id <math>\preceq</math> root) e<sub>1</sub> e<sub>2</sub></code>	<code>t</code>

Delegation (or acts-for) relation `id  $\preceq$  root` is an order on principals. [Abadi, Burrows, Lampson, Plotkin]

# Security theorems

---

**Soundness:** If  $\vdash e : \tau$ , then  $e \longrightarrow^* v$ .

*A well-typed program does not generate any run-time errors.*

# Security theorems

---

**Soundness:** If  $\vdash e : t$ , then  $e \longrightarrow^* v$ .

*A well-typed program does not generate any run-time errors.*

**Noninterference:** If  $x : \text{int}\{H\} \vdash e : \text{bool}\{L\}$  and  $\vdash v_1 : \text{int}\{H\}$  and  $\vdash v_2 : \text{int}\{H\}$ , then  $e\{v_1/x\} \longrightarrow^* v$  iff  $e\{v_2/x\} \longrightarrow^* v$ .

*If an observer  $e$  is a well-typed Boolean program of low-security  $\text{bool}\{L\}$ , then  $e$  cannot distinguish possibly different values such as  $v_1, v_2$  of higher security  $\text{int}\{H\}$ .*

# Declassification and DLM

---

- Noninterference provides a strong *static guarantee* but it is too restrictive for practical programming
  - ↪ Idea: declassification
  - ↪ Challenge: regulate its use

# Declassification and DLM

---

- Noninterference provides a strong *static guarantee* but it is too restrictive for practical programming
  - ↪ Idea: declassification
  - ↪ Challenge: regulate its use
- Decentralized label model [Myers, Liskov]
  - ↪ Labels: {Alice:Bob,Chris; Dave:Bob}
  - ↪ Alice and Dave are the *owners* of the data
  - ↪ Both Alice's and Dave's policies are enforced

# Declassification and DLM

---

- Noninterference provides a strong *static guarantee* but it is too restrictive for practical programming
  - ↪ Idea: declassification
  - ↪ Challenge: regulate its use
- Decentralized label model [Myers, Liskov]
  - ↪ Labels: {Alice:Bob,Chris; Dave:Bob}
  - ↪ Alice and Dave are the *owners* of the data
  - ↪ Both Alice's and Dave's policies are enforced
- Jif: DLM + robust declassification [Myers et al.]
  - ↪ Idea: owner has the *authority* to declassify
  - ↪ Challenge: authority for run-time principals

# Authority and certificates

---

- Acquisition: obtain a certificate that says Alice *grants* the authority of declassifying to Bob

cert c = **acquire** Alice ▷ Declassify Bob

# Authority and certificates

---

- Acquisition: obtain a certificate that says Alice *grants* the authority of declassifying to Bob

cert c = **acquire** Alice ▷ Declassify Bob

General form of authority: X ▷ i

X is a principal and i is a *privilege* such as

1. Delegate ATM: “allow ATM to act for X”
2. Declassify Net: “allow declassifying X’s data”
3. Withdraw 100: “allow debiting X’s account”

# Authority and certificates

---

- Acquisition: obtain a certificate that says Alice *grants* the authority of declassifying to Bob  
`cert c = acquire Alice ▷ Declassify Bob`
- Verification: determine whether a certificate *implies* that Alice *grants* the authority of declassifying to Bob  
`if (c  $\Rightarrow$  Alice ▷ Declassify Bob) e1 e2`

# Authority and certificates

---

- Acquisition: obtain a certificate that says Alice *grants* the authority of declassifying to Bob  
$$\text{cert } c = \text{acquire Alice } \triangleright \text{Declassify Bob}$$
- Verification: determine whether a certificate *implies* that Alice *grants* the authority of declassifying to Bob  
$$\text{if } (c \Rightarrow \text{Alice } \triangleright \text{Declassify Bob}) \ e_1 \ e_2$$
- Access control and Java stack inspection [Appel, Felton, Wallach; Fournet, Gordon; Pottier, Skalka, Smith]
- Robust declassification [Myers, Zdancewic]

# Application: distributed banking

---

```
void main (...) {  
    ...  
}
```

- Distributed system between ATM and Bank server
- Customer is a *run-time* principal
- Customer delegates to ATM for bank service
- Certificates are created and logged for audit purpose
- For brevity, assume a secure network is established

# Application: distributed banking

---

```
void main (cert cnet) {  
    #user{ATM:} id = login();  
    cert cdel = acquire id ▷ Delegate ATM;  
    ...  
}
```

# Application: distributed banking

---

```
void main (cert cnet) {  
    #user{ATM:} id = login();  
    cert cdel = acquire id ▷ Delegate ATM;  
    cert creq = acquire id ▷ Withdraw 100;  
    Msg{ATM:} m = new Msg(id, cdel, creq);  
    ...  
}
```

# Application: distributed banking

---

```
void main (cert cnet) {  
    #user{ATM:} id = login();  
    cert cdel = acquire id ▷ Delegate ATM;  
    cert creq = acquire id ▷ Withdraw 100;  
    Msg{ATM:} m = new Msg(id, cdel, creq);  
    ...  
    Msg{ATM:Net} x = declassify m as Msg{ATM:Net};  
    int{user:} balance = request(x); ...  
}
```

# Application: distributed banking

---

```
void main (cert cnet) {  
    #user{ATM:} id = login();  
    cert cdel = acquire id ▷ Delegate ATM;  
    cert creq = acquire id ▷ Withdraw 100;  
    Msg{ATM:} m = new Msg(id, cdel, creq);  
    if (cnet ⇒ ATM ▷ Declassify Net) {  
        Msg{ATM:Net} x = declassify m as Msg{ATM:Net};  
        int{user:} balance = request(x); ...  
    }  
}
```

# Abstract implementation

---

The run-time system of our language must maintain *principal hierarchy* and *authority certificates*. Specifically:

1. Principal:  $X$
2. Delegation test:  $\text{if } (X_1 \preceq X_2) \dots$
3. Capability and authority:  $X\{i\}$  and  $X \triangleright i$
4. Privileges:  $\text{Delegate } X$  and  $\text{Declassify } X$
5. Certificate acquisition:  $\text{acquire } X \triangleright i$
6. Certificate verification:  $\text{if } (c \Rightarrow X \triangleright i) \dots$

# PKI implementation

---

PKI naturally supports *distributed* access control:

principal	$X$	$\longrightarrow$	$K_X$	public key
delegate	$\text{Del } X$	$\longrightarrow$	$(\text{DEL}, K_X)$	certificate contents
declassify	$\text{Dcls } X$	$\longrightarrow$	$(\text{DCLS}, K_X)$	certificate contents
capability	$X \{i\}$	$\longrightarrow$	$K_X^{-1} \{i\}$	digital certificate
authority	$X \triangleright i$	$\longrightarrow$	$(K_X, i)$	public key, privilege

Delegation tests and certificate verification are mapped to comparing public keys in the *delegation chain* and *cryptographic verification*. [Chothia, Duggan, Vitek; ...]

# Conclusion

---

- **Scope:** language-based security in decentralized label model with declassification
- **Contributions:** connect *compile-time* and *run-time* principals in a sound type system
- **Features:** allow programmers to express policies that interact with existing security infrastructures
- Apollo project: interpreter with monadic labels  
↳ <http://www.cis.upenn.edu/~stse/apollo>
- Security-Oriented Languages (SOL)  
↳ <http://www.cis.upenn.edu/~stevez/sol>