Capability Type Systems for Secure Cooperation

Philip W. L. Fong
pwlfong@cs.uregina.ca

Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada
Overview

- Motivation
- Capability Types for Data Flow Control
- Discretionary Capability Confinement
- Enabling Technologies for Link-Time Type Checking
Motivation
Secure Cooperation

Protection of mutually suspicious code units from one another as they collaborate in the same execution environment.

Applications
- e.g., mobile code systems, software plug-in’s, OS kernel extensions

Prototypical Platforms
- Java, CLR
Problem Statement

- Language-based access control mechanisms:
  - Employing programming language technologies to address access control challenges of secure cooperation:
    - Stack inspection [Wallach et al 2000]
    - Inlined reference monitor [Erlingsson & Schneider 2000]
    - Access control based on shallow execution history [Abadi & Fournet 2003] [Fong 2004 / IEEE S&P]
Problem Statement

- Language-based access control mechanisms:
  - Employing programming language technologies to address access control challenges of secure cooperation:
    - Stack inspection [Wallach et al 2000]
    - Inlined reference monitor [Erlingsson & Schneider 2000]
    - Access control based on shallow execution history [Abadi & Fournet 2003] [Fong 2004 / IEEE S&P]
  - Enforce only control flow policies [Jensen et al 1999]
  - Fail to account for data flow policies:
    - Controlling propagation of delegated resources
    - Object-level communication protocols
Prison Mail System (PMS)

- A toy problem originally proposed by Ambler and Hoch (1977) for studying protection in programming languages.
- We present here a simplified, object-oriented variant without diluting its original challenges.
In the PMS are 3 types of objects:

1. **Prisoner**
   - forbidden from direct communication
   - all message exchanges must be mediated by the PMS

2. **Guard**
   - responsible for message delivery

3. **Mail**
   - message carriers
public interface Prisoner {
    void send(Prisoner receiver, Guard messenger);
    void receive(Mail message);
}

public interface Guard {
    void deliver(Mail message, Prisoner receiver);
}

public final class Mail {
    public Mail(String) { msg = m; }
    public String read() { return msg; }
    private String msg;
}
PMS: Interaction Protocol

sender : Prisoner

messenger : Guard

receiver : Prisoner

send(receiver, messenger)
deliver(message, receiver)
receive(message)
public interface Prisoner {
    (1) void send(Prisoner receiver, Guard messenger);
    (3) void receive(Mail message);
}

public interface Guard {
    (2) void deliver(Mail message, Prisoner receiver);
}

public final class Mail {
    public Mail(String) { msg = m; }
    public String read() { return msg; }
    private String msg;
}
PMS: Protection Problems

1. Mutual Suspicion.
   The messenger is not allowed to read the message.

2. Rights Amplification.
   The receiver is allowed to read the message.

3. Controlled Propagation of Capability.
   Message in transit must not be leaked to other Guards.

   The sender shall not send the message directly to the receiver.

   The sender and the messenger must not store away the receiver for future use.
Capability Types for Data Flow Control
Intuition: Capability Type Systems

- A type qualifier restricts how one may access the underlying datum:

```c
int *p;
```
A type qualifier restricts how one may access the underlying datum:

\[
\text{const int } *p;
\]
Intuition: Capability Type Systems

A type qualifier restricts how one may access the underlying datum:

\[
\text{const int *p;}
\]

A **capability** [Dennis & Van Horn 1966] is an unforgeable pair:

\[
\langle \text{object-reference, access-rights} \rangle.
\]
A type qualifier restricts how one may access the underlying datum:

\[
\text{const int } \ast \text{p;}
\]

A capability [Dennis & Van Horn 1966] is an unforgeable pair:

\[
\langle \text{object-reference, access-rights} \rangle.
\]

In a programming language context, a type qualifier plays the role of \textit{access-rights}:

\[
\text{access-rights Object ref;}
\]

Such is the motivation for capability type systems [Boyland \textit{et al} 2001].
Intuition: Data Flow Control

Access control by restricting argument passing:

\[ f(a, b) \]

If \( a \) is prevented from being passed into \( f \), then the invocation of \( f \) will fail.

Application to object-oriented context:

\[ a.f(b) \]

The message receiver \( a \) is also an argument!

- Controlling to which method \( a \) may be passed as an argument allows us to present a restricted view of \( a \), in which some methods of \( a \) are inaccessible . . .
- . . . thus a capability type system.

Further development:

- Controlling propagation and sharing of capabilities.
Capability Types for Data Flow Control

\[ T ::= \top \quad \text{Top: most restricted - no aliasing} \]
\[ \mid \bot \quad \text{Bottom: least restricted - any aliasing} \]
\[ \mid P \rightarrow T \quad \text{Propagation: pass into } P \text{ as a } T \text{ argument} \]
\[ \mid [T] \quad \text{Sharing: store into a field as a } T \text{ value} \]
\[ \mid T_1 \sqcap T_2 \quad \text{Choice: exercise either } T_1 \text{ or } T_2 \]
\[ \mid \chi \quad \text{Abstraction: recursive capability types} \]

\[ P ::= \text{a set of method signatures} \]

\[ X ::= \text{a capability type variable} \]
Example  Consider the recursive definition:

\[ R = (\text{READ} \rightarrow \bot) \sqcap [R] \]

where \( \text{READ} = \{ \text{String Mail.read()} \} \).

- A *Mail* reference with a capability type satisfying the above recursive definition can be read right away, or be saved into a field for both future reading and future sharing.
- The capability type \( R \) corresponds to the following *labelled transition system (LTS)*:

```
[ . ]
```

```
R  \rightarrow \text{String Mail.read()}  \rightarrow \bot
```

public interface Guard {
    void deliver(Mail message, Prisoner receiver);
}
Solution Annotation

```java
public interface Guard {
    void deliver(@flow( RECV → [READ → ⊥] ) Mail message,
                  Prisoner receiver);
}

where

    RECV    def = {void Prisoner.receive(Mail)}
    READ    def = {String Mail.read()}
```
public interface Guard {
    void deliver(@flow(RECV → [READ → ⊥] ) Mail message,
                 @flow(RECV → ⊥ ) Prisoner receiver);
}

where

RECV def = {void Prisoner.receive(Mail)}
READ def = {String Mail.read()}
public interface Prisoner {
    void send(@flow( DLVR → RECV → ⊥) Prisoner receiver,
              @flow( DLVR → ⊥ ) Guard messenger);
    void receive(@flow( [READ → ⊥] ) Mail message);
}

public interface Guard {
    void deliver(@flow( RECV → [READ → ⊥] ) Mail message,
                 @flow( RECV → ⊥ ) Prisoner receiver);
}

where

    DLVR  def=  {void Guard.deliver(Mail, Prisoner)}
    RECV  def=  {void Prisoner.receive(Mail)}
    READ  def=  {String Mail.read()}
What Have Been Achieved?

- All 5 protection problems are fully addressed.
- Type constraints are formulated at the level of JVM bytecode.
  - Enforceable at link time by the code consumer.
  - Type checking involves data flow analysis.
- Implementation is underway:
  - Compile-time tool for (i) type checking annotated Java source code and (ii) embedding type annotations into Java classfiles.
  - Link-time type checker embedded in a JVM.
- Future work: type inference & certifying compiler.
Discretionary Capability
Confinement
Challenge: Capability Spoofing Attacks
Challenge: Capability Spoofing Attacks

What if an untrusted class implements both the *Prisoner* and *Guard* interfaces?
Challenge: Capability Spoofing Attacks

- What if an untrusted class implements both the *Prisoner* and *Guard* interfaces?
- What if a *Guard* creates, and subsequently colludes with, a *Prisoner* instance?
Challenge: Capability Spoofing Attacks

- What if an untrusted class implements both the *Prisoner* and *Guard* interfaces?
- What if a *Guard* creates, and subsequently colludes with, a *Prisoner* instance?

Crux

Ability to forge capabilities
Discretionary Capability Confinement

- Discretionary Capability Confinement (DCC)
  [with Boting Yang, submitted for review]:
  - A type system for avoiding capability forging:
    - statically enforceable
    - lightweight - no data flow analysis
  - Orthogonal but complementary to the previous type system.
DCC Basic Concepts

Confinement Domains

A

B

capability-of

C

subsumed-by

trusts
(DCC1) **Capability confinement.** A reference may escape a confinement domain only in 2 cases:
1. it does not escape as a capability
2. it escapes via argument passing
DCC Typing Constraints

(\textbf{DCC1}) \textbf{Capability confinement.} A reference may escape a confinement domain only in 2 cases:
1. it does not escape as a capability
2. it escapes via argument passing

(\textbf{DCC2}) \textbf{Mediated access.} If $C$ does not trust $A$, the $A$ shall not invoke the static methods declared in $C$. 
DCC Typing Constraints

(DCC1) **Capability confinement.** A reference may escape a confinement domain only in 2 cases:
1. it does not escape as a capability
2. it escapes via argument passing

(DCC2) **Mediated access.** If $C$ does not trust $A$, the $A$ shall not invoke the static methods declared in $C$.

(DCC3) **Consistent subtyping.** A reference type is always trusted by its supertypes.
DCC Typing Constraints

(DCC1) **Capability confinement.** A reference may escape a confinement domain only in 2 cases:
1. it does not escape as a capability
2. it escapes via argument passing

(DCC2) **Mediated access.** If \( C \) does not trust \( A \), the \( A \) shall not invoke the static methods declared in \( C \).

(DCC3) **Consistent subtyping.** A reference type is always trusted by its supertypes.

(DCC4) **Hereditary mutual suspicion.** If \( A \) and \( B \) does not trust one another, then so do \( A \) and a subtype of \( B \).
DCC Annotations for PMS

Root
{ Object, String, etc }

PrisonerDomain
{ Prisoner, Mail }

GuardDomain
{ Guard }

AppCoreDomain
{ Main }

subsumed-by

subsumed-by

subsumed-by

subsumed-by
What Have Been Achieved?

- DCC is interesting in its own right:
  - implementing a useful capability type system for Java with minimum perturbation to the semantics of the language.
  - a basic building block for building more sophisticated capability type systems.
What Have Been Achieved?

- DCC is interesting in its own right:
  - implementing a useful capability type system for Java with minimum perturbation to the semantics of the language.
  - a basic building block for building more sophisticated capability type systems.

- Type constraints are formulated at the level of JVM bytecode.
  - Enforceable at link time by the code consumer.
  - Type checking very efficient: no data flow analysis involved
What Have Been Achieved?

DCC is interesting in its own right:
- implementing a useful capability type system for Java with minimum perturbation to the semantics of the language.
- a basic building block for building more sophisticated capability type systems.

Type constraints are formulated at the level of JVM bytecode.
- Enforceable at link time by the code consumer.
- Type checking very efficient: no data flow analysis involved

Fully implemented:
- Compile-time type checker based on JDK 5.0 annotation facility and Apache BCEL
- Link-time type checker based on PVM
What Have Been Achieved?

DCC is interesting in its own right:
- implementing a useful capability type system for Java with minimum perturbation to the semantics of the language.
- a basic building block for building more sophisticated capability type systems.

Type constraints are formulated at the level of JVM bytecode.
- Enforceable at link time by the code consumer.
- Type checking very efficient: no data flow analysis involved

Fully implemented:
- Compile-time type checker based on JDK 5.0 annotation facility and Apache BCEL
- Link-time type checker based on PVM

A pure Java implementation based on IsoMod is underway
What Have Been Achieved?

- Inferring DCC annotations from legacy systems:
  - Application: certifying compiler
  - DCC type inference is NP-Complete
    - Complexity core: \( DCC_1 \)
  - A branch-and-bound type inference algorithm
    - based on graph transformation
  - Infer DCC annotations for medium-size open source software such as Jython (336 classes), JRuby (468 classes), Kawa (746 classes) in a matter of minutes
Enabling Technologies for Link-Time Type Checking
Link-Time Type Checking

Need to embed link-time type checkers into the dynamic linking process of a JVM
Link-Time Type Checking

Need to embed link-time type checkers into the dynamic linking process of a JVM

2 enabling technologies:
- PVM - Pluggable Verification Modules
- IsoMod - A Module System for Software Isolation
Pluggable Verification Modules

Pluggable Verification Modules (PVM) [OOPSLA’04]

- Bytecode verification of the JVM becomes a pluggable service that can be readily replaced, reconfigured and augmented.
- Application-specific verification services can be safely incorporated into the dynamic linking process of the JVM
- Based on a verification architecture known as **Proof Linking**

**Challenge:** Static verification interacts with dynamic linking

- How does one separate the two?
Modular Verification

Verifier

Run-time Environment

Loader

Untrusted Code Unit

Verified Code Unit

Proof Linker

Commitments + Obligations

Obligations

Linking Primitives | Obligations
---|---
resolve S in X | O₁ O₂ O₃

Commitment Database

Obligation Discharging Schedule
Proof Linking

1. Request that symbol $S$ is to be resolved in $X$

2. Look up obligations of requested action

3. Prove obligations by consulting commitments

4. Resolve

<table>
<thead>
<tr>
<th>Linking Primitives</th>
<th>Obligations</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolve $S$ in $X$</td>
<td>$O_1 O_2 O_3$</td>
</tr>
</tbody>
</table>

Commitment Database

Obligation Discharging Schedule
Proof Linking Architecture

- Java instantiation satisfies 3 correctness conditions — Safety, Monotonicity, Completion
  [ACM FSE’98]
- Correctness proof formally verified by PVS
  [ACM TOSEM’00]
- Support distributed verification protocols
  [ICSE’98, JVM’01]
IsoMod

A Module System for Software Isolation
[with Simon Orr, submitted for review]

Access control by controlling name visibility
A Module System for Software Isolation
[with Simon Orr, submitted for review]
- Access control by controlling name visibility

- An expressive, declarative policy language
- DCC fully encoded in IsoMod
IsoMod

- A Module System for Software Isolation
  [with Simon Orr, submitted for review]
  - Access control by controlling name visibility

- An expressive, declarative policy language
  - DCC fully encoded in IsoMod
  - Implementation is underway
    - Pure Java implementation as a custom class loader