



Discretionary Capability Confinement

Philip W. L. Fong

`pwl.fong@cs.uregina.ca`

Department of Computer Science

University of Regina

Regina, Saskatchewan, Canada



Overview



1. Language-Based Capability Systems
2. Discretionary Capability Confinement (DCC)
 - Programming model
 - Confinement guarantees
 - Discussions





Language-Based Capability Systems



Dynamically Extensible Systems



- Impossible to anticipate user requirements:

- \Rightarrow late binding of functionalities

- **Dynamically Extensible Systems**

Executable extensions are dynamically linked into the address space of a software system, either to deliver a short-lived service, or to augment the capability of the host system in a permanent manner.

- Examples:

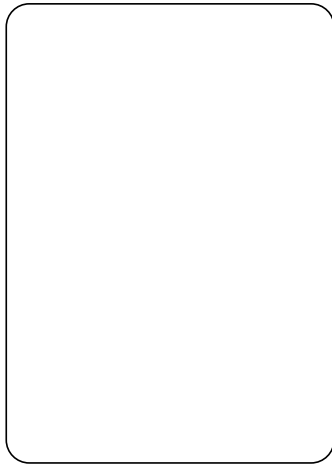
- Mobile code systems
- Scriptable applications
- Software systems with plug-in architectures



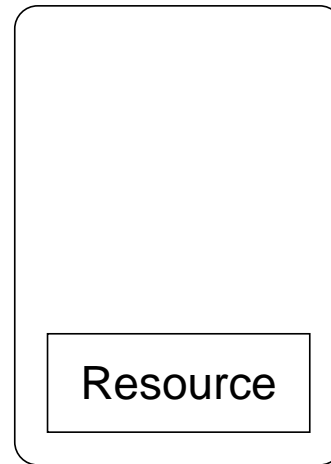


Dynamic Extension (1)

**Code Producer
Process**

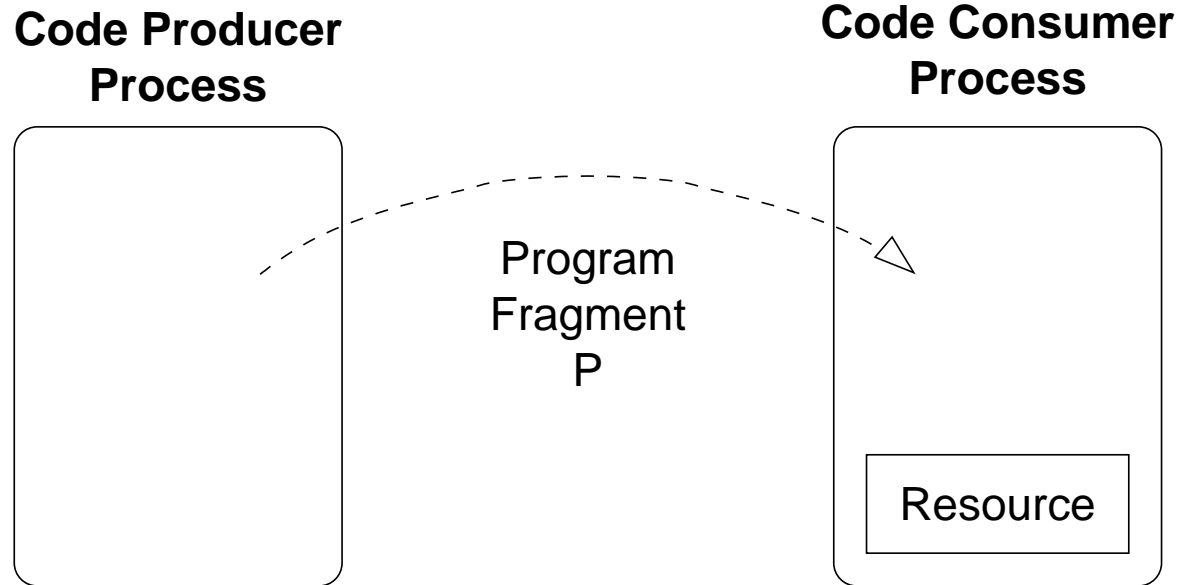


**Code Consumer
Process**





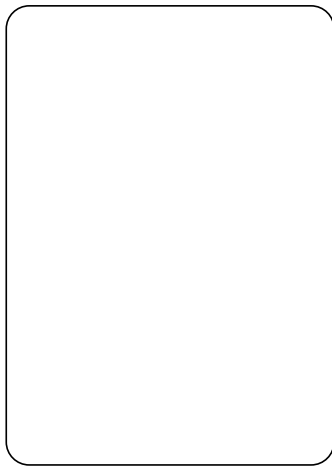
Dynamic Extension (2)



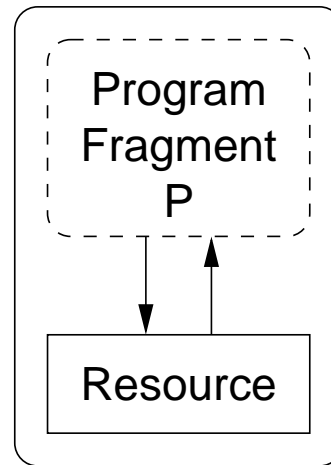


Dynamic Extension (3)

**Code Producer
Process**



**Code Consumer
Process**



Mutual Suspicion



Mutual Suspicion: *Collaborating code units within the same process are suspicious of one another.*

- Traditional OS protection mechanisms use a process to define protection boundaries.
- Code units within the same process are supposed to trust one another.

The left hand does not trust the right hand!

- An extensible system must now assume the security posture of a multiprogramming operating system.



Language-Based Security



- **Language-Based Security [Schneider et al. 2001]**
 - Employing programming language technologies to address the security challenges of complex software systems.



Language-Based Security

- **Language-Based Security [Schneider et al. 2001]**
 - Employing programming language technologies to address the security challenges of complex software systems.
- **Examples**
 - Program analysis
 - type systems
 - program verification
 - Program rewriting

Plan



- **Goal:**
 - Supporting mutual suspicion in extensible systems
- **Means:**
 - Language-based capability systems



Two Related Phenomena

1. Why do we have Trojan Horses?

[Rees 1996]

Two Related Phenomena



1. Why do we have Trojan Horses?

- Programs inherits all the privileges of the users who invoke them.

[Rees 1996]



Two Related Phenomena



1. Why do we have Trojan Horses?
 - Programs inherits all the privileges of the users who invoke them.
2. What can go wrong when mutually suspicious code units collaborate with one another?

[Rees 1996]



Two Related Phenomena



1. Why do we have Trojan Horses?
 - Programs inherits all the privileges of the users who invoke them.
2. What can go wrong when mutually suspicious code units collaborate with one another?
 - Callee inherits privileges of the caller.
 - Input to subprograms can be tweaked to cause privilege escalation.

[Rees 1996]



The Confused Deputy Anomaly

```
public class Compiler {  
    public void compile(String src, String obj) { ... }  
}
```

[Hardy 1988]

• **Setting:**

- Compiler has privilege to write to a global log file.
- Compiler must also inherit the privilege of caller so that it can write to the specified object file.

The Confused Deputy Anomaly

```
public class Compiler {  
    public void compile(String src, String obj) { ... }  
}
```

[Hardy 1988]

- **Setting:**

- Compiler has privilege to write to a global log file.
- Compiler must also inherit the privilege of caller so that it can write to the specified object file.

- **What if:**

- Caller pass in the name of the log file as the object file name ...
- Log file will be overwritten!

Principle of Least Privilege



- *Every program and every user of the system should operate using the least set of privileges necessary to complete the job.*

[Saltzer & Schroeder 1975]



Principle of Least Privilege

- *Every program and every user of the system should operate using the least set of privileges necessary to complete the job.*

[Saltzer & Schroeder 1975]

- **Want:**
 - a fine-grained mechanism for dynamically granting the least possible privilege to a subprogram

Capability



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

⟨object-reference, access-rights⟩

Possession of a capability is **both** the necessary and sufficient condition for access.



Capability

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

$\langle \textit{object-reference}, \textit{access-rights} \rangle$

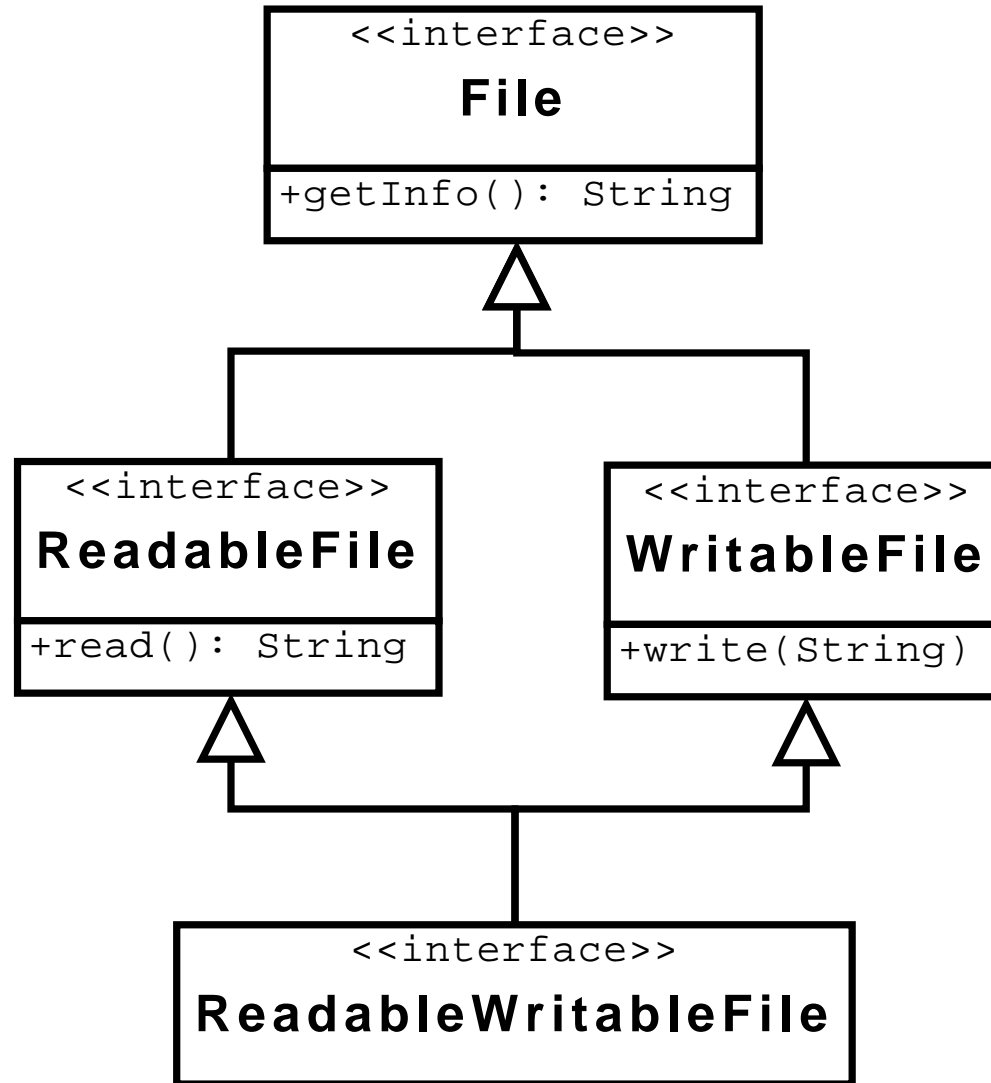
Possession of a capability is **both** the necessary and sufficient condition for access.

- In a type-safe programming language, a type plays the role of *access-rights*.

$T \ x ;$

A statically-typed reference is a capability!

A Java Example



Want This ...

```
class Alice {  
    ReadableWritableFile file;  
    void delegate() {  
        Bob.process(file);  
    }  
}  
class Bob {  
    static void process(ReadableFile file) {  
        ...  
    }  
}
```

Merits of Capability



1. Fine-grained
2. Name and privilege are no longer separated



Wait! What about Capability Confinement?

1. Capability Theft

- “steal” capabilities from foreign protection domains

2. Capability Leakage

- “push” capabilities to less privileged protection domains
- The lack of **capability confinement** is considered to be the main weakness of capability systems [*Wallach et al. 1997, Chander et al. 2001*].

Contributions



Discretionary Capability Confinement (DCC)

A static annotation system for Java:

- Statically typed references as capabilities
- Provably avoids capability theft and leakage
- Support a strong form of static separation of duty known as **hereditary mutual suspicion**

ESORICS 2006, IJIS 2008

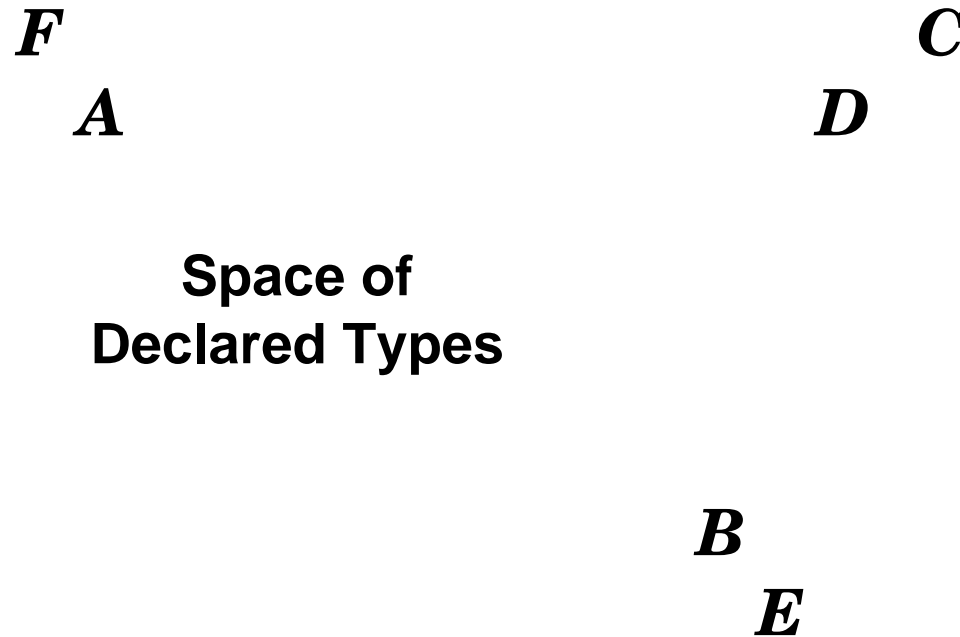




Programming Model

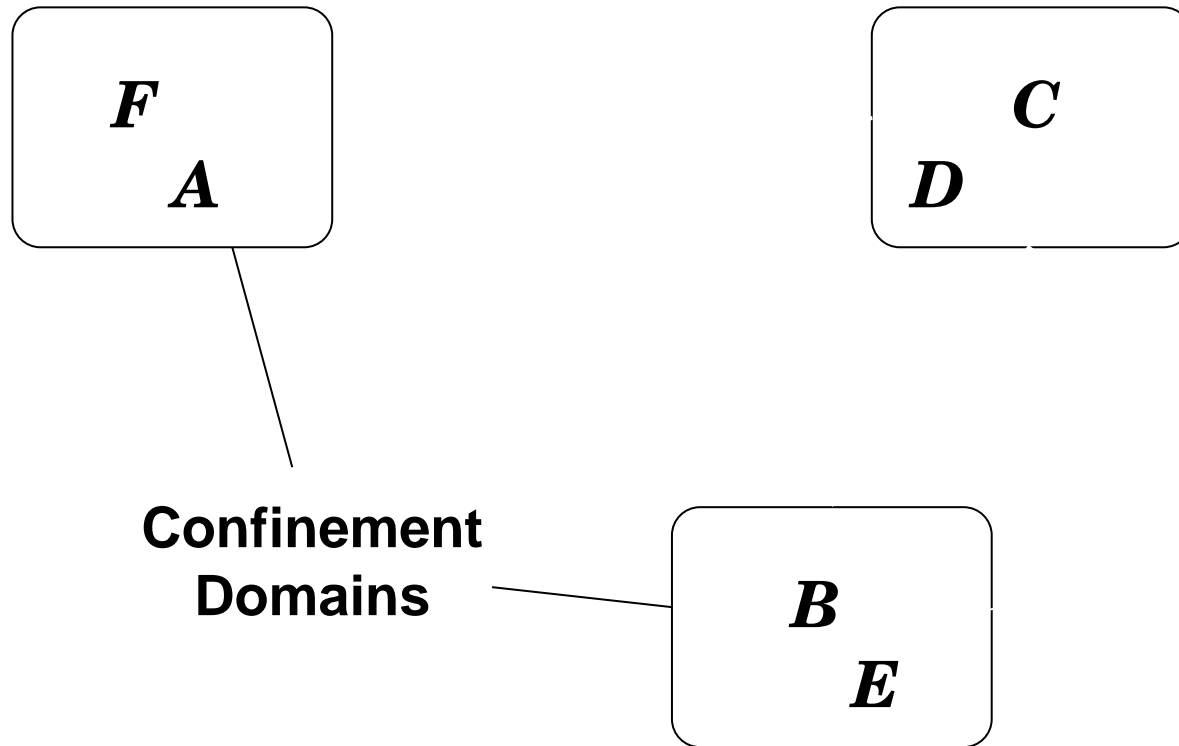


Confinement Domains & Capabilities (1)



- Declared Types: classes or interfaces

Confinement Domains & Capabilities (2)



[Vitek et al.]

- Every declared type is a member of a **Confinement Domain**.
- A confinement domain is a statically defined protection domain.

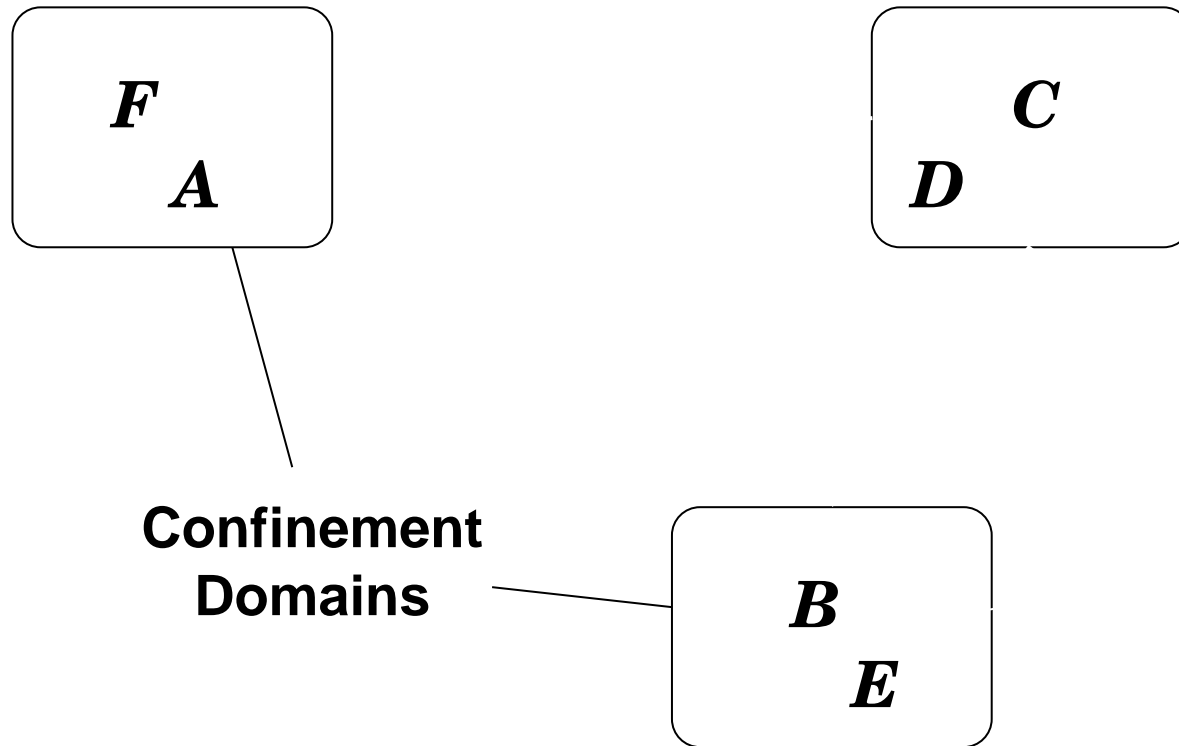
Example: Domain Membership

```
@Confined( Dom1.class )  
public class A { ... }
```

```
@Confined( Dom2.class )  
public class B { ... }
```

```
@Confined( Dom3.class )  
public class C { ... }
```

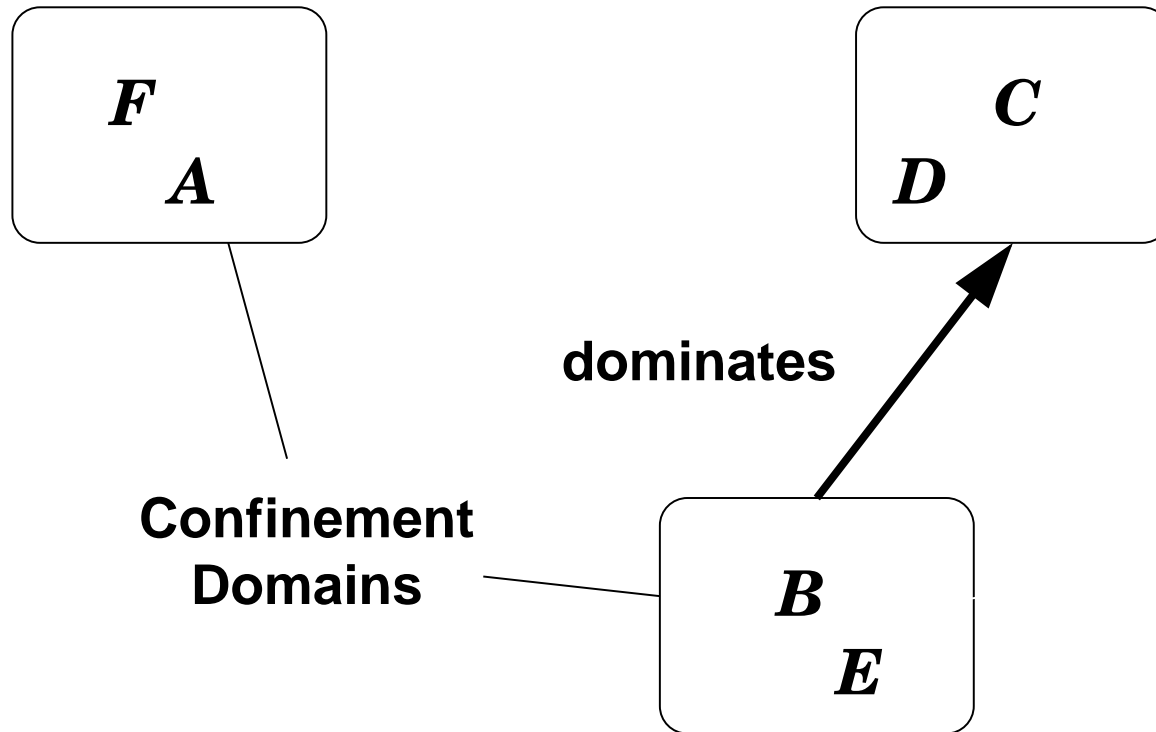
Confinement Domains & Capabilities (2)



[Vitek et al.]

- Every declared type is a member of a **Confinement Domain**.
- A confinement domain is a statically defined protection domain.

Confinement Domains & Capabilities (3)



- A **dominance relation** (pre-order) is defined among the domains.
 - The more dominating domains are freer to acquire typed references.

Example: Dominance Hierarchy



@Domain

```
public interface Dom1 extends Root { }
```

@Domain

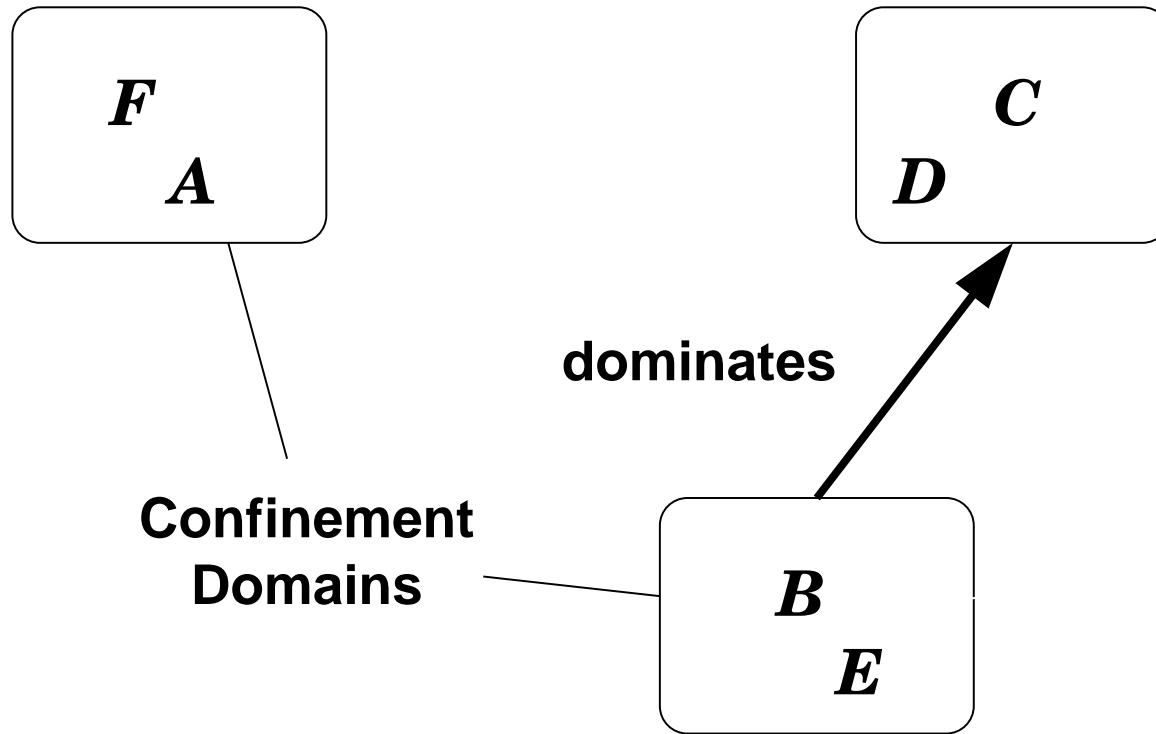
```
public interface Dom2 extends Dom3 { }
```

@Domain

```
public interface Dom3 extends Root { }
```

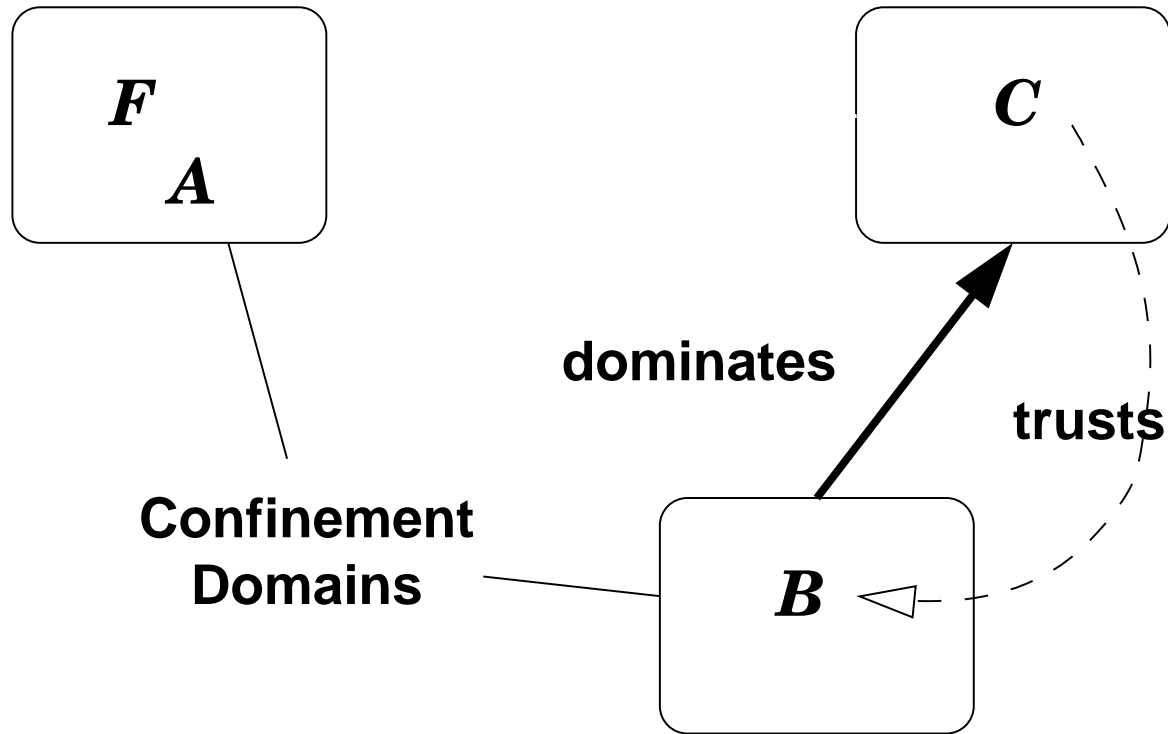


Confinement Domains & Capabilities (3)



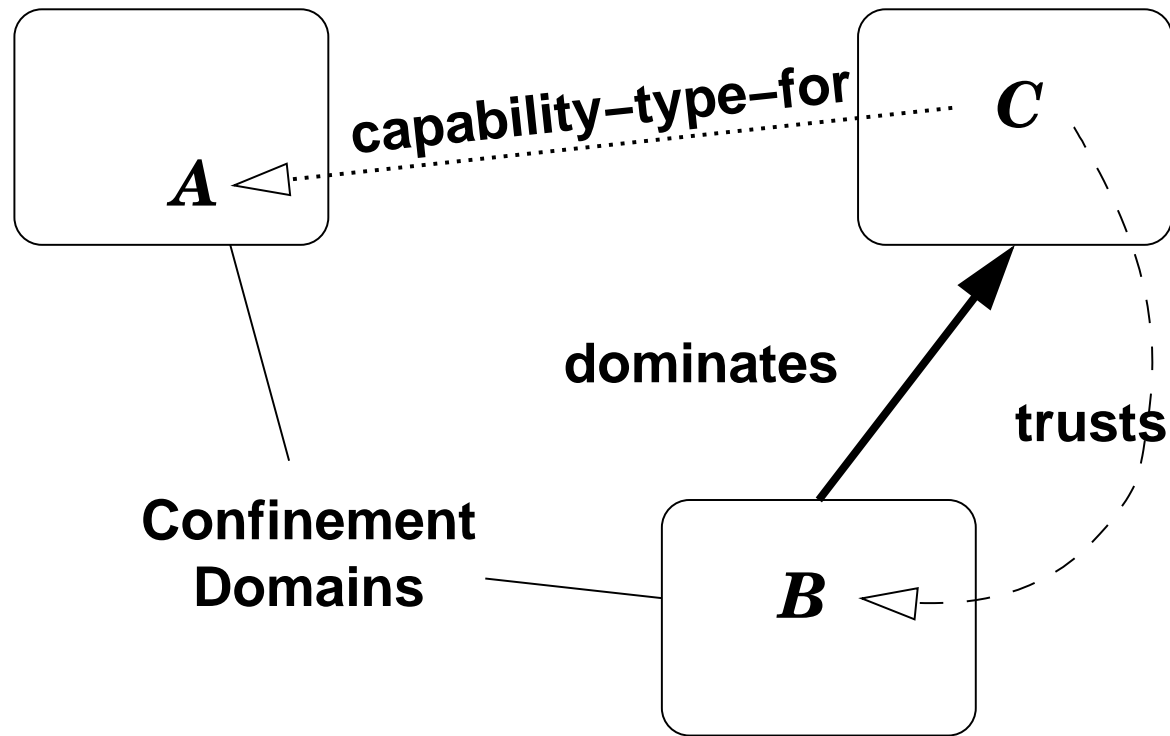
- A **dominance relation** (pre-order) is defined among the domains.
 - The more dominating domains are freer to acquire typed references.

Confinement Domains & Capabilities (4)



- The dominance relation (over domains) induces a partial order over declared types.
- The more dominating domains contain more trusted declared types.

Confinement Domains & Capabilities (5)



- If C does not trust A , then C is a capability type from the perspective of A .
- Acquisition of C -type references by A should be carefully controlled.

Capability Propagation



Consider a statically typed reference $r : C$

- Once $r : C$ enters a domain, it roams freely within that domain.
- $r : C$ may escape from the domain only if one of the following holds:
 1. $r : C$ is not a capability of the receiving domain
 2. r is passed as a type- C argument



Capability Propagation



Consider a statically typed reference $r : C$

- Once $r : C$ enters a domain, it roams freely within that domain.
- $r : C$ may escape from the domain only if one of the following holds:
 1. $r : C$ is not a capability of the receiving domain
 2. r is passed as a type- C argument

We are **NOT** confining **references** (r)

We are confining **statically typed references** ($r : C$)



Capability Propagation



Consider a statically typed reference $r : C$

- Once $r : C$ enters a domain, it roams freely within that domain.
- $r : C$ may escape from the domain only if one of the following holds:
 1. $r : C$ is not a capability of the receiving domain
 2. r is passed as a type- C argument
 - ... subject to the control of the caller's **capability granting policy** ...



Capability Granting Policies

- Every method m is annotated with a **capability granting policy** [Gong 1989], which specifies
 1. the types of capabilities that can be granted by m
 2. the domains to which m may grant a capability

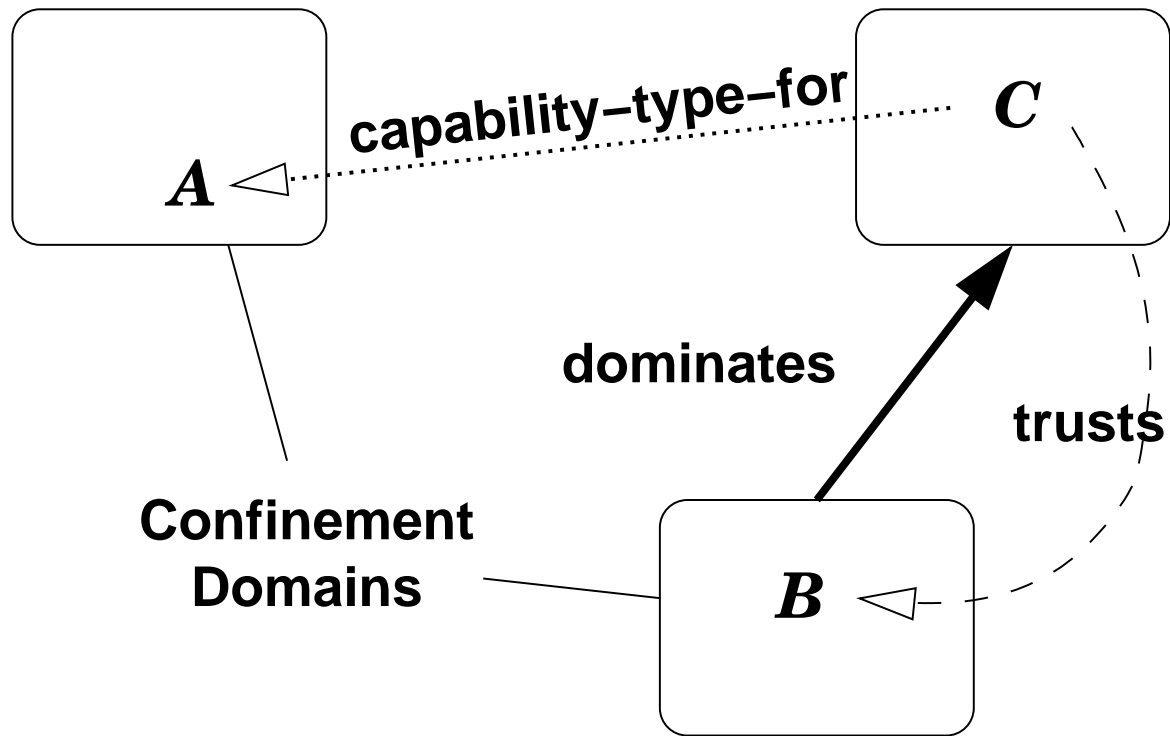
Example: Capability Granting Policy

```
@Confined( Dom2.class )
public class B {
    @Grants( Dom2.class )
    void m() {
        C.n(new B());
    }
}
```

```
@Confined( Dom3.class )
public class C {
    public static void n(B x) { ... }
}
```



Example: Capability Granting Policy



Hereditary Mutual Suspicion



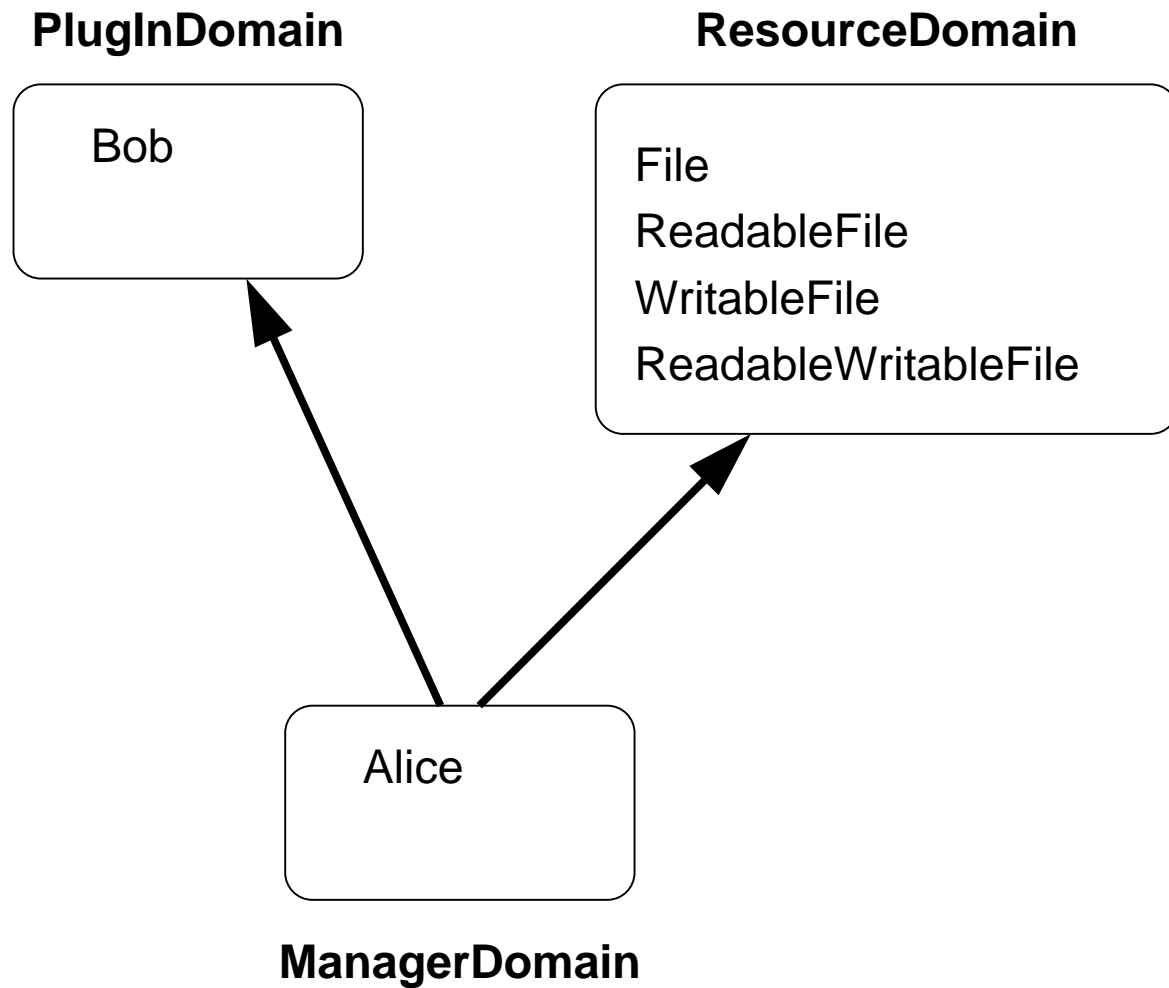
Definition Declared types A and B are said to be ***mutually suspicious*** if they do not trust one another.

Hereditary Mutual Suspicion If A and B are mutually suspicious, then so are a subtype of A and a subtype of B .





Example: Alic, Bob, and Files



Subtlety #1: Static Method



• Mediated Access:

- A is not allowed to invoke static method $C.m$ if C is a capability type for A
- This ensures accesses are mediated via capabilities.



Subtlety #2: Confused Deputy



- **No Amplification of Privilege:**

- Method $A.m$ is not allowed to invoke method $B.n$ if the capability granting policy of $A.m$ is more restrictive than that of $B.n$.
- Capability granting privilege diminishes along a call chain.



Subtlety #3: Dynamic Method Dispatching

● Impersonation Avoidance:

- Each method interface promises the caller the following:
 1. destination domain of capability arguments
 2. origin domain of returned capability
 3. capability granting policy
- Method $B'.n'$ is not allowed to override $B.n$ unless $B'.n'$ “preserves” the interface promised by $B.n$.
- Avoid “*impersonation*” caused by method overriding.

Subtlety #4: Principle of Subsumption



• Save Widening:

- B is not allowed to be a subtype of A if B is a capability type for A .
- Preserve substitutivity by ensuring that upcast never turn a capability to a non-capability.





Confinement Guarantees



Semantic Model



Featherweight JVM

- a non-deterministic production system for modeling all possible execution traces of a JVM
- reasoning about **reachability** and **stack invariants**
- appeared in **SCP 2007**



Confinement Theorem



Theorem (Discretionary Capability Confinement)

Suppose $S \xrightarrow{*}_{\Sigma} T$, where S has the form $\langle \Pi, \Gamma; \Phi, A.m, \diamond \rangle$. Let \mathcal{D} be an arbitrary domain. If $\text{Accessible}_{[\mathcal{D}]}(r : C \mid T)$, then at least one of the following conditions holds:

1. $\text{Accessible}_{[\mathcal{D}]}(r : C \mid S)$ (previously accessible)
2. $I(C) \blacktriangleright \mathcal{D}$ (not a capability)
3. $C \triangleright m \wedge \mathcal{D} \blacktriangleright I(m)$ (controlled capability propagation)



Safe Methods



Definition

A method $A.m$ is said to be **safe** iff $m \triangleright A$.

Inuition

Executing a safe method $A.m$ will only cause those domains dominated by $I(A)$ to acquire capabilities that A can instantiate.



Corollaries



Corollary (No Theft)

Invoking a safe method never leads to capability theft.

Corollary (No Leakage)

A safe method never leaks capabilities to a domain that is not dominated by the home domain of the method.





Discussions



Extensions

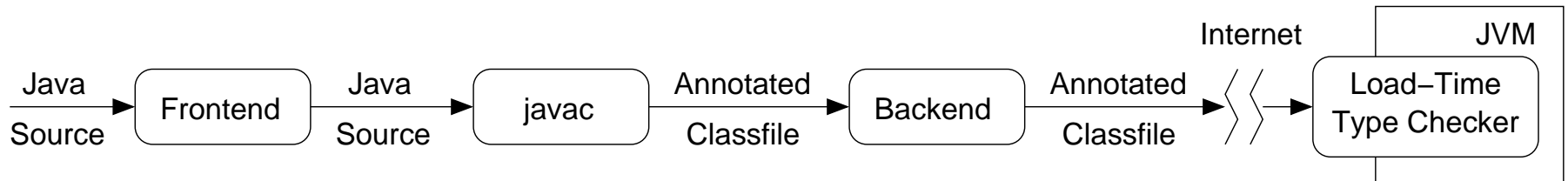


- Handling arrays, genericity, nested classes
- Modular enforcement of Hereditary Mutual Suspicion
- Finer-grained representation of capability granting policies

[See IJIS 2008 paper for more details.]



Implementation



DCC Constraints are Scoping Rules

- DCC typing constraints are nothing more than **scoping rules**:
 - Scoping rules govern whether a symbol is visible to a given code unit.
 - DCC typing constraints have been completely encoded in the ISO MOD policy language.
 - ISO MOD is a system that allows programmers to customize the scoping rules for Java (appeared in **ACSAC 2006**).
- Very efficient to check.
 - No dataflow analysis involved.

Future Work



- Inferring DCC annotations from legacy code
- Allowing a limited form of right amplification while maintaining privilege attenuation
- Delegation control
- Obligations





Questions?

