Discretionary Capability Confinement

Philip W. L. Fong

Technical Report CS-2006-03
March 2006 (First Draft)
July 2006 (Revised)

Department of Computer Science
University of Regina
Regina, Saskatchewan, S4S 0A2
Canada

ISBN 0-7731-0567-0 (print)
ISBN 0-7731-0568-9 (online)

Discretionary Capability Confinement

Philip W. L. Fong

Department of Computer Science, University of Regina, Regina, SK, Canada
pwlfong@cs.uregina.ca

Abstract. Motivated by the need of application-level access control in dynamically extensible systems, this work proposes a static annotation system for modeling capabilities in a Java-like programming language. Unlike previous language-based capability systems, the proposed annotation system can provably enforce capability confinement. This confinement guarantee is leveraged to model a strong form of separation of duty known as hereditary mutual suspicion. The annotation system has been fully implemented in a standard Java Virtual Machine.

1 Introduction

Dynamic extensibility is a popular architectural feature of networked or distributed software systems [1]. In such systems, code units originating from potentially untrusted origins can be linked dynamically into the core system in order to deliver a short-lived service, or to augment the feature set of the core system in a more permanent manner. The protection infrastructure of a dynamically extensible system is often language based [2]. Previous work on language-based access control largely focuses on infrastructure protection via various forms of history-based access control [3–9]. The security posture of infrastructure protection tends to divide run-time principals into a trusted "kernel" vs untrusted "extensions", and focuses on controlling the access of kernel resources by extension code. This security posture does not adequately address the need of ***application-level security***, that is, the imposition of collaboration protocols among peer code units, and the enforcement of access control over resources that are defined and shared by these peer code units. This paper reports an effort to address this limitation through a language-based capability system.

The notion of ***capabilities*** [10, 11] is a classical access control mechanism for supporting secure cooperation of mutually suspicious code units [12, 13]. A capability is an unforgeable pair comprised of an object reference plus a set of access rights that can be exercised through the reference. In a capability system, possession of a capability is the necessary and sufficient condition for exercising the specified rights on the named object.

This inherent symmetry makes capability systems a natural protection mechanism for enforcing application-level security.

Previous approaches to implement language-based capability systems involve the employment of either the proxy design pattern [14] or load-time binary rewriting [15] to achieve the effect of interposition. Although these "dynamic" approaches are versatile enough to support ***capability revocation***, they are not without blemish. Leaving performance issues aside, a common critique [14, 16] is that an unmodified capability model fails to address the need of ***capability confinement***: once a capability is granted to a receiver, there is no way to prevent further propagation.

An alternative approach is to embed the notion of capabilities into a static type system [17]. In a ***capability type system*** [18, 19], every object reference is statically assigned a capability type, which imposes on the object reference a set of operational restrictions that constrains the way the underlying object may be accessed. When a code unit delegates a resource to an untrusted peer, it may do so by passing to the peer a resource reference that has been statically typed by a capability type, thereby exposing to the peer only a limited view of the resource.

The class hierarchy of a Java-like programming language [20, 21] provides non-intrusive building blocks for capability types. Specifically, one may exploit ***abstract types*** (i.e., abstract classes or interfaces in Java) as capability types. An abstract type exposes only a limited subset of the functionalities provided by the underlying object, and thus an object reference with an abstract type can be considered a capability of the underlying object. A code unit wishing to share an object with its peer may grant the latter a reference properly typed with an abstract type. The receiver of the reference may then access the underlying object through the constrained interface. This scheme, however, suffers from the same lack of capability confinement. The problem manifests itself in two ways.

1. ***Capability Theft.*** A code unit may "steal" a capability from code units belonging to a foreign protection domain, thereby amplifying its own access rights. Worst still, capabilities can be easily forged by unconstrained object instantiation and dynamic downcasting.
2. ***Capability Leakage.*** A code unit in possession of a capability may intentionally or accidentally "push" the capability to code units residing in a less privileged protection domain.

This paper proposes a lightweight, static annotation system called ***Discretionary Capability Confinement (DCC)***, which fully supports the adoption of abstract types as capability types and provably

prevents capability theft and leakage. Targeting Java-like programming languages, the annotation system offers the following features:

- While the binding of a code unit to its protection domain is performed statically, the granting of permissions to a protection domain occurs dynamically through the propagation of ***capabilities***.
- Inspired by Vitek *et al* [22–25], a protection domain is identified with a ***confinement domain***. Once a capability is acquired, it roams freely within the receiving confinement domain. A capability may only escape from a confinement domain via explicit capability granting.
- Following the design of Gong [26], although a method may freely exercise the capabilities it possesses, its ability to grant capabilities is subject to discretionary control by a ***capability granting policy***.
- Under mild conditions, capability confinement guarantees such as ***no theft*** and ***no leakage*** can be proven. Programmers can achieve these guarantees by adhering to simple annotation practices.
- An application-level collaboration protocol called ***hereditary mutual suspicion*** is enforced. This protocol entails a strong form of ***separation of duty*** [27, 28]: not only is the notion of mutually-exclusive roles supported, collusion between them is severely restricted because of the confinement guarantees above.

The contributions of this paper are the following:

- A widely held belief among security researchers is that language-based capability systems adopting the reference-as-capability metaphor cannot address the need of capability confinement [14, 16]. Employing type-based confinement, this work has successfully demonstrated that such a capability system is in fact feasible.
- The traditional approach to support separation of duty is through the imposition of mutually exclusive roles [29, 28]. This work proposes a novel mechanism, hereditary mutual suspicion, to support separation of duty in an object-oriented setting. When combined with confinement guarantees, this mechanism not only implements mutually exclusive roles, but also provably eliminate certain forms of collusion.

This paper is organized as follows. Sect. 2 motivates DCC by an example. Sect. 3 outlines the type constraints of DCC. Sect. 4 establishes the formal properties of DCC. Sect. 5 discusses extensions and variations. Sect. 6 reports implementation experiences. The paper concludes with related work and future work.

2 Motivation

The Hero-Sidekick Game. Suppose we are developing a role-playing game. Over time, a playable character, called a *hero* (e.g., Bat Man), may acquire an arbitrary number of *sidekicks* (e.g., Robin). A sidekick is a non-playable, AI-controlled character whose behavior is a function of the state and behavior of the hero to which it is associated. The intention is that a sidekick augments the power of its hero. For example, when the health of the hero is low, or when the hero is attacked by a villain of incomparably higher hit points, then a *defensive* sidekick may attempt to block the movement of the villain and take the hit points for the hero. Alternatively, when the hero is attempting a long-range offense, then a *scout* sidekick may automatically move towards the target to improve visibility. A group of scout sidekicks may also establish a defense perimeter when the hero is regenerating. Along the same vein, when the hero is attacking, an *offensive* sidekick may augment the fire power of the hero .... The maximum number of sidekicks that may be attached to a hero is a function of the hero's type and experience. A hero may adopt or orphan a sidekick at will. New sidekick and/or hero types may be introduced in future releases of the game.

A possible design of the game is to employ the Observer pattern [30] to capture the dynamic dependencies between heros and sidekicks, as is shown in Fig. 1, where sidekicks are observers of heros. The `GameEngine` class is responsible for creating instances of `Hero` and `Sidekick`, and managing the attachment and detachment of `Sidekick`s[1].

The set up in Fig. 1 would have worked had it not been the following complication: *a requirement of the game is such that users may dynamically download new hero or sidekick types from the internet during a game play.* The introduction of dynamic software extensions significantly complicates the security posture of the application. Specifically, the developer must now actively ensure fair game play by eliminating the possibility of cheating through the downloading of malicious characters. Two prototypical cheats are the following.

Cheat I: Capability Theft. A `Sidekick` reference can be seen as a capability, the possession of which makes a `Hero` instance more potent. A malicious `Hero` can augment its own power by creating new instances of concrete `Sidekick`s, or stealing existing instances from unprotected sources, and then attaching these instances to itself.

[1] Although Java syntax is adopted here, the example is applicable to other languages that share a similar object model with Java.

```
public interface Character { /* Common character behavior ... */ }
public interface Observable {
   State getState();
}
public abstract class Hero implements Character, Observable {
   protected Sidekick observers[];
   public final void attach(Sidekick sidekick) { /* Attach sidekick */ }
   public final void detach(Sidekick sidekick) { /* Detach sidekick */ }
   public final void broadcast() {
      for (Sidekick observer : observers)
         if (observer != null)
            observer.update(this);
   }
}
public interface Sidekick extends Character {
   void update(Observable hero);
}
public class GameEngine { /* Manage life cycle of characters ... */ }
```

Fig. 1. A set up of the hero-sidekick game

Cheat II: Capability Theft and Leakage. A `Hero` exposes two type interfaces: (i) a sidekick management interface (i.e., `Hero`), and (ii) a state query interface (i.e., `Observable`). While the former is intended to be used exclusively by the `GameEngine`, the latter is a restrictive interface through which `Hero`s may be accessed securely by `Sidekick`s. This means that a `Hero` reference is also a capability from the perspective of `Sidekick`. Upon receiving a `Hero` object through the `Observable` argument of the `update` method, a malicious `Sidekick` may downcast the `Observable` reference to a `Hero` reference, and thus exposes the sidekick management interface of the `Hero` object (i.e., capability theft). This in turn allows the malicious `Sidekick` to attach powerful sidekicks to the `Hero` object, thereby turning the `Hero` object into a more potent character (i.e., capability leakage).

Solution Approach. To control the capability propagation, DCC assigns the `Hero` and `Sidekick` interfaces to two distinct ***confinement domains*** [22], and restricts the exchange of capability references between the two domains. Specifically, capability references may only cross confinement boundaries via explicit argument passing. Capability granting is thus possible only under conscious ***discretion***. Notice that the above restrictions shall not apply to `GameEngine`, because it is by design responsible for managing the life cycle of `Hero`s and `Sidekick`s, and as such it requires the rights to acquire instances of `Hero`s and `Sidekick`s. This motivates the need to have a notion of ***trust*** to discriminate the two cases above.

To further control the granting of capabilities, a static ***capability granting policy*** [26] can be imposed on a method. For example, a capability granting policy can be imposed on the `broadcast` method so that the latter passes only `Observable` references to `update`, but never `Hero` references.

Our goal is not only to prevent capability theft and leaking between the abstract types `Hero` and `Sidekick`, but also between the subtypes of `Hero` and those of `Sidekick`. In other words, we want to treat `Hero` and `Sidekick` as ***roles***, prescribe capability confinement constraints between the two roles, and then require that their subtypes also conform to these constraints. DCC achieves this via a mechanism known as ***hereditary mutual suspicion***.

3 Discretionary Capability Confinement

In the Java platform, code units bind via dynamic linking, program verification that is performed against source code, or administrated only by the code producer, cannot be trusted. Therefore, if DCC is to be used for enabling secure cooperation, then it must be formulated at the bytecode level, so that it may be enforced by the code consumer. This section presents the DCC annotation system for the JVM bytecode language. The threat model is reviewed in Sect. 3.1, the main type constraints are specified in Sect. 3.2, and the utility of DCC in addressing the security challenges of the running example is discussed in Sect.3.3.

3.1 Threat Model

As the present goal is to restrict the forging and propagation of abstractly typed references, we begin the discussion with an exhaustive analysis of all means by which a reference type A may ***acquire*** a reference of type C. We use metavariables A, B and C to denote *raw* JVM reference types (i.e., after erasure). We consider class and interface types here, and defer the treatment of array types and genericity till Sect. 5.1.

1. A reference type B ***grants*** a reference of type C to reference type A when B invokes a method[2] declared in A, passing an argument via a formal parameter (including pseudo-parameter `this`) of type C.

[2] To simplify discussion, we adopt the following shorthand in this paper. By a ***method*** we mean either an instance or static method, or an instance or class initializer. By a ***field*** we mean either an instance or static field.

2. A reference type B ***shares*** a reference of type C with reference type A when one of the following occurs: **(a)** A invokes a method declared in B with return type C; **(b)** A reads a field declared in B with field type C; **(c)** B writes a reference into a field declared in A with field type C.
3. A reference type A ***generates*** a reference of type C when one of the following occurs: **(a)** A creates an instance of C; **(b)** A dynamically casts a reference to type C; **(c)** an exception handler in A with catch type C catches an exception.

3.2 Type Constraints

We postulate that the space of reference types is partitioned by the programmer into a finite number of ***confinement domains***, so that every reference type C is assigned to exactly one confinement domain via a domain label $l(C)$. We use metavariables $\mathcal{D}$ and $\mathcal{E}$ to denote confinement domains. The confinement domains are further organized into a ***dominance hierarchy*** by a programmer-defined partial order $\blacktriangleright$. We say that $\mathcal{D}$ ***dominates*** $\mathcal{E}$ whenever $\mathcal{E} \blacktriangleright \mathcal{D}$. The dominance hierarchy induces a pre-ordering of reference types. Specifically, if $l(B) = \mathcal{E}$, $l(A) = \mathcal{D}$, and $\mathcal{E} \blacktriangleright \mathcal{D}$ then we write $B \triangleright A$, and say that B ***trusts*** A. By definition $\triangleright$ is reflexive and transitive (but not antisymmetric). We thus write $A \bowtie B$ iff both $A \triangleright B$ and $B \triangleright A$. The binary relation $\bowtie$ is an equivalence relation, the equivalence classes of which are simply the confinement domains. If $C \triangleright A$ does not hold, then a reference of type C is said to be a ***capability*** for A. Intuitively, capabilities should provide the sole means for untrusted types to access methods declared in capability types. The following constraint is imposed to ensure that an untrusted access is always mediated by a capability:

($\mathcal{DCC}1$) **Mediated access.** Unless $B \triangleright A$, A shall not invoke a static method declared in B.

It is harmless for A to acquire a non-capability reference C (i.e., $C \triangleright A$). Capability acquisition, however, is restricted by a number of constraints, the first of which is the following:

($\mathcal{DCC}2$) **Discriminatory capability confinement.** Capability granting is the sole means by which a domain acquires capabilities:

1. A can generate a reference of type C only if $C \triangleright A$. [That is, no capability generation is permitted.]

2. B may share a reference of type C with A only if $C \triangleright A \vee A \bowtie B$. [That is, capability sharing is not permitted across domain boundaries.]

In other words, capability acquisition only occurs as a result of explicit granting. Once a capability is acquired, it roams freely within the receiving confinement domain. Escape from a confinement domain is only possible when the escaping reference does not escape as a capability, or when it escapes as a capability via argument passing.

We also postulate that there is a ***root domain*** $\top$ so that $\top \blacktriangleright \mathcal{D}$ for all $\mathcal{D}$. All Java platform classes are members of the root domain $\top$. This means they can be freely acquired by any reference type[3].

Capability granting is regulated by discretionary control. We postulate that every declared method has a unique designator, which is denoted by metavariables m and n. We occasionally write $A.m$ to stress the fact that m is declared in A. Associated with every method m is a programmer-supplied label $l(m)$, called the ***capability granting policy*** of m. The label $l(m)$ is a confinement domain. (If $l(n) = \mathcal{E}$, $l(m) = \mathcal{D}$, and $\mathcal{E} \blacktriangleright \mathcal{D}$, then we write $n \triangleright m$. Similarly, we write $m \triangleright A$ and $A \triangleright m$ for the obvious meaning.) Intuitively, the capability granting policy $l(m)$ dictates what capabilities may be granted by m, and to whom m may grant a capability.

($\mathcal{DCC}3$) **Discretionary capability granting.** If $A.m$ invokes[4] $B.n$, and C is the type of a formal parameter of n, then $C \triangleright B \vee A \bowtie B \vee (B \triangleright m \wedge C \triangleright m)$.

That is, capability granting ($\neg\, C \triangleright B$) across domain boundaries ($\neg\, A \bowtie B$) must adhere to the capability granting policy of the caller ($B \triangleright m \wedge C \triangleright m$). Specifically, a capability granting policy $l(m)$ ensures that m only grants capabilities to those reference types B satisfying $B \triangleright m$, and that m only grants capabilities of type C for which $C \triangleright m$.

A method may be tricked into (directly or indirectly) invoking another method that does not honor the caller's capability granting policy. This classical anomaly is known as the Confused Deputy [31]. The following

[3] Notice that the focus of this paper is not to protect Java platform resources. Instead, our goal is to enforce application-level security policies that prescribe interaction protocols among dynamically loaded software extensions. The organization of the domain hierarchy therefore reflects this concern: platform classes and application core classes belong respectively to the least and the most dominating domain.

[4] In the case of instance methods, if $A.m$ invokes $B.n$, the actual method that gets dispatched may be a method $B'.n'$ declared in a proper subtype B' of B. Constraints ($\mathcal{DCC}3$) and ($\mathcal{DCC}4$) only regulate method invocation. Dynamic method dispatching is regulated by controlling method overriding through ($\mathcal{DCC}6$).

constraint ensures that capability granting policies are always preserved along a call chain.

($\mathcal{DCC}4$) **No Amplification of Rights.** A method m may invoke another method n only if $n \triangleright m$.

We now turn to constraints that capture how capability confinement interacts with subtyping. We write $A <: B$ whenever A is either B itself or one of B's subtypes. A subtype exposes the interface of its supertypes. Specifically, if a reference type A has acquired a reference of type B, then A has effectively acquired a reference of every type B' that is a supertype of B. This is because implicit widening conversion is not considered a reference acquisition event in our threat model (for efficiency concern). The following constraint is imposed to ensure that widening conversion is safe: i.e., widening does not turn a non-capability into a capability.

($\mathcal{DCC}5$) **Safe widening.** If $A <: B$ then $B \triangleright A$.

Dynamic method dispatching presents a second point of interaction between capability confinement and subtyping. When an instance method $B.n$ is invoked, the method that actually gets dispatched may be a method $B'.n'$ declared in a subtype B' of B. This allows $B'.n'$ to "*impersonate*" $B.n$, potentially allowing $B'.n'$ to **(i)** grant capabilities in a way that violates the capability granting policy of $B.n$, **(ii)** return a capability to a caller with whom B' is not supposed to share capabilities, or **(iii)** accept a capability argument that is originally intended for B rather than B'. To avoid these anomalies, the following constraints are imposed.

($\mathcal{DCC}6$) **Impersonation Avoidance.** Suppose $B.n$ is overridden by $B'.n'$. The following must hold:

1. $n' \triangleright n$. [That is, overriding never relaxes capability granting rights.]
2. If the method return type is C, then $C \triangleright B \vee B \bowtie B'$. [That is, a method that returns a capability may not be overridden by a method declared in a different domain.]
3. If C is the type of a formal parameter, then $C \triangleright B' \vee B \bowtie B'$. [That is, a method may be granted a capability only if it does not override a method declared in a different domain.]

Notice the intentional asymmetry between constraints 2 and 3 above. This asymmetry reflects the fact that the two constraints are used differently in the proof of our confinement results.

If reference types A and B do not trust each other (i.e., neither $A \triangleright B$ nor $B \triangleright A$ hold), they are said to be ***mutually suspicious***. The following constraint requires that mutual suspicion is preserved by subtyping.

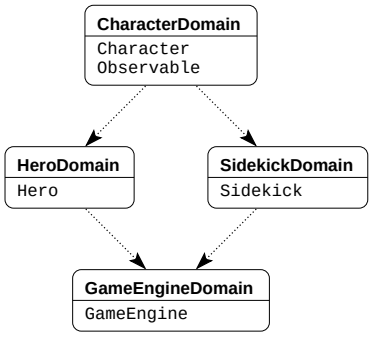

Fig. 2. Dominance hierarchy for the hero-sidekick application. Arrows represent "dominated-by" relationships (▶)

($\mathcal{DCC}7$) **Hereditary mutual suspicion (1st form).** Suppose A and B are mutually suspicious. If $A' <: A$ and $B' <: B$, then A' and B' are also mutually suspicious.

This constraint results in a strong form of static separation of duty [28]. Firstly, as the trust relation $\triangleright$ is reflexive, no reference type can be simultaneously a subtype of both A and B. This renders A and B mutually exclusive roles. Secondly, as we shall see in Sect. 4.4, a class of collusion between A and B can be completely eliminated under mild conditions. This constraint will be reformulated in Sect. 5.2 to facilitate modular enforcement.

3.3 Addressing the Security Challenges

The challenge of capability theft and leakage described in our running example (Sect. 2) can be fully addressed by DCC. A simple solution is described here, and a more sophisticated solution, involving controlled capability exchange, is described in Sect. 5.3.

The confinement domains and dominance hierarchy as shown in Fig. 2 can be defined for the hero-sidekick game application. Because `HeroDomain` and `SidekickDomain` are incomparable in the dominance hierarchy, `Hero` and `Sidekick` are capabilities for each other. Consequently, not only are `Sidekick`s not allowed to downcast an `Observable` reference to a `Hero` capability (i.e., Cheat II), `Hero`s are also forbidden to create new `Sidekick` capabilities or to steal such capabilities through aliasing (Cheat I). Furthermore, the dominance hierarchy also renders `GameEngineDomain` the most dominating confinement domain, thereby allowing `GameEngine` to have full access to the reference types declared

$$
\begin{array}{rcll}
A, B, C & \in & \mathcal{C} & \text{raw reference types} \\
m, n & \in & \mathcal{M} & \text{method designators} \\
p, q, r & \in & \mathcal{O} & \text{object references} \\
S, T & ::= & \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle & \text{VM states} \\
\Pi & ::= & \emptyset \mid \Pi \cup \{r : C\} & \text{object pools} \\
\Gamma & ::= & \emptyset \mid \Gamma \cup \{B \rightsquigarrow q : C\} \mid \Gamma \cup \{p : B \rightsquigarrow q : C\} & \text{link graphs} \\
\Phi & ::= & \emptyset \mid \Phi \cup \{r : C\} & \text{stack frames} \\
\sigma & ::= & \diamond \mid \mathit{push}(\Phi, A.m, C, \sigma) & \text{proper stacks}
\end{array}
$$

Fig. 3. FJVM states

in the rest of the confinement domains. We also annotate every method $A.m$ displayed in Fig. 1 with a capability granting policy of $l(m) = l(A)$: e.g., $l(\texttt{update}) = l(\texttt{SidekickDomain})$. Consequently, even if a `Sidekick` obtains a `Hero` reference, it is still not allowed to attach any sidekick to that `Hero` instance (Cheat II). Lastly, hereditary mutual suspicion allows us to turn `Hero` and `Sidekick` into mutually suspicious roles, so that their subtypes cannot conspire to communicate capabilities.

4 Confinement Properties

The problem of capability confinement is a major challenge in capability-based systems [14, 16]. As in other discretionary access control mechanisms, safety analysis [32–34] must be conducted to characterize the conditions under which capabilities are not propagated to unintended parties. This section establishes such confinement guarantees for DCC. The results are formalized in a lightweight model of the JVM called Featherweight JVM (FJVM) [35], which is introduced in Sect. 4.1. Sect. 4.2 embeds the DCC typing rules into FJVM. The main confinement theorem is then established in Sect. 4.3. The corollaries of the confinement theorem are discussed in Sect. 4.4.

4.1 Featherweight JVM

The FJVM model (Figs. 3 and 4) is a nondeterministic production system that describes how the JVM state evolves over time in reaction to access events. Nondeterminism is employed because we are not modeling the execution of a specific bytecode sequence, but rather all possible access events that may be generated by the JVM when well-typed bytecode sequences are executed.

$$\frac{\Phi \vdash r : C \qquad C <: B}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : B\}, A.m, \sigma \rangle} \quad \text{(T-Widen)}$$

$$\frac{\begin{array}{c} r \text{ is a fresh object reference from } \mathcal{O} \\ \mathbf{new}\langle B \rangle \in \Sigma[A.m] \end{array}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi \cup \{r : B\}, \Gamma; \Phi \cup \{r : B\}, A.m, \sigma \rangle} \quad \text{(T-New)}$$

$$\frac{\begin{array}{c} \Phi \vdash r : C \qquad \Pi \vdash r : C' \qquad C' <: B \\ \mathbf{checkcast}\langle B \rangle \in \Sigma[A.m] \end{array}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : B\}, A.m, \sigma \rangle} \quad \text{(T-CheckCast)}$$

$$\frac{\begin{array}{c} \Gamma \vdash B \rightsquigarrow q : C \\ \mathbf{getstatic}\langle B : C \rangle \in \Sigma[A.m] \end{array}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{q : C\}, A.m, \sigma \rangle} \quad \text{(T-GetStatic)}$$

$$\frac{\begin{array}{c} \Phi \vdash q : C \\ \mathbf{putstatic}\langle B : C \rangle \in \Sigma[A.m] \end{array}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma \cup \{B \rightsquigarrow q : C\}; \Phi, A.m, \sigma \rangle} \quad \text{(T-PutStatic)}$$

$$\frac{\begin{array}{c} \Phi \vdash p : B_0 \qquad B_0 <: B \qquad \Gamma \vdash p : B \rightsquigarrow q : C \\ \mathbf{getfield}\langle B : C \rangle \in \Sigma[A.m] \end{array}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{q : C\}, A.m, \sigma \rangle} \quad \text{(T-GetField)}$$

$$\frac{\begin{array}{c} \Phi \vdash p : B_0 \qquad B_0 <: B \qquad \Phi \vdash q : C \\ \mathbf{putfield}\langle B : C \rangle \in \Sigma[A.m] \end{array}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma \cup \{p : B \rightsquigarrow q : C\}; \Phi, A.m, \sigma \rangle} \quad \text{(T-PutField)}$$

$$\frac{\begin{array}{c} \Phi \vdash \overline{r} : \overline{C} \\ \mathbf{invokestatic}\langle B.n : \overline{C} \to C \rangle \in \Sigma[A.m] \end{array}}{\begin{array}{c} \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi', B.n, \sigma' \rangle \\ \text{where } \Phi' = \{\overline{r} : \overline{C}\} \text{ and } \sigma' = \mathit{push}(\Phi, A.m, C, \sigma) \end{array}} \quad \text{(T-InvokeStatic)}$$

$$\frac{\begin{array}{c} \Phi \vdash r_0 : C_0 \qquad C_0 <: B \qquad \Phi \vdash \overline{r} : \overline{C} \\ \Pi \vdash r_0 : B'' \qquad B'' <: B' \qquad B' <: B \\ \mathbf{invokemethod}\langle B.n : \overline{C} \to C \rangle [B'.n'] \in \Sigma[A.m] \end{array}}{\begin{array}{c} \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi', B'.n', \sigma' \rangle \\ \text{where } \Phi' = \{r_0 : B', \overline{r} : \overline{C}\} \text{ and } \sigma' = \mathit{push}(\Phi, A.m, C, \sigma) \end{array}} \quad \text{(T-InvokeMethod)}$$

$$\frac{\Phi' \vdash r : C}{\langle \Pi, \Gamma; \Phi', B.n, \mathit{push}(\Phi, A.m, C, \sigma) \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : C\}, A.m, \sigma \rangle} \quad \text{(T-Return)}$$

Fig. 4. FJVM transitions

Object References. The JVM model manipulates *object references*. Every object reference r is an instance of exactly one class. An instance of C may contain an arbitrary number of typed fields, each of which is declared either in C one of its supertypes. Each field in turn stores an object reference. A field may only be instantiated once but never updated. The null reference is modeled by the absence of link.

VM State. A configuration of the form $\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle$ represents the global state of the JVM (Fig. 3). The components to the left of the semicolon model the heap, while those to the right model the execution stack.

Heap. The *object pool* Π is a finite set of *allocations* $r : C$. Intuitively, Π records all the objects that have been created by the VM, together with their class membership. The *link graph* Γ is a finite set of *links*. A *global link* $B \rightsquigarrow q : C$ records that some static field declared in B, with field type C, stores the object reference q. An *object link* $p : B \rightsquigarrow q : C$ records that the object p contains a field declared in B, with field type C, storing the object reference q. Together Π and Γ models the global state of the heap.

Stack. The VM state components Φ, $A.m$, σ model the stack of the current thread of execution. At the top of the stack is a stack frame Φ and an execution context $A.m$. A *stack frame* is a finite set of *labeled references* $r : C$. The set Φ models the references accessible in a JVM stack frame. Each reference r is associated with a type label C. The type label is an instrumentation that allows us to track the part of a reference's type interface that is visible to an execution context. The *execution context* $A.m$ is the currently executing method. The last component σ models the call chain that leads to the current VM state. Specifically, a *proper stack* σ is either an *empty stack*, $\diamond$, or a *non-empty stack*, $push(\Phi, A.m, C, \sigma)$, where Φ is the caller stack frame, $A.m$ is the execution context of the caller, C is the declared return type of the callee method, and σ is another proper stack.

Notations. In the following, we write $\overline{x}$ for a list $x_1, \ldots, x_k$. We also write $X \vdash x$ if $x \in X$. Obvious variations shall be clear from the context. For example, we write $\Phi \vdash \overline{r} : \overline{C}$ for the conjunction of the list $\Phi \vdash r_1 : C_1$, $\ldots$, $\Phi \vdash r_k : C_k$.

Transitions. The transition rules in Fig. 4 define the state transition relation $\rightarrow_\Sigma$. The production T-Widen "promotes" the type label of an

object reference in the stack frame. The transition rule T-New creates a fresh object reference in the object pool, and makes that reference accessible in the top stack frame. The rule T-CheckCast models dynamic type casting, and tags an object reference in the stack frame with an alternative type label consistent with the class of the object reference. The production T-GetStatic models static field getting, and makes the content of a static field accessible in the current stack frame. The production T-PutStatic models static field setting, and introduces a global link into the link graph. The production T-GetField models instance field getting, and makes the target of an existing object link accessible in the top stack frame. The production T-PutField models instance field setting, and creates a link between two object references. The transition rule T-InvokeStatic models the invocation of static methods. The caller invokes a static method $B.n$, saves the caller stack frame (Φ), creates a new stack frame (Φ') for the callee, passes the arguments ($\overline{r} : \overline{C}$) from the caller stack frame (Φ) to the callee stack frame (Φ'), and constrains the return type (C). The transition rule T-InvokeMethod models instance method invocation. The rule works in similar way as T-InvokeStatic, except for two points: (i) although the method signature $B.n$ is invoked, the actual method that gets dispatched is the method $B'.n'$; (ii) the receiver object ($r_0 : C_0$) is passed as an implicit argument, and assumes a type of B' when it is in the callee stack frame. The transition rule T-Return pops the top stack frame (Φ'), resurrects the stack frame (Φ) of the caller (A), and makes the return value ($r : C$) available in the caller stack frame.

Safety Policy. The transition relation $\rightarrow_\Sigma$ is parameterized by a *safety policy* Σ. Intuitively, a safety policy Σ specifies for each execution context $A.m$ the set $\Sigma[A.m]$ of permitted events. An event e has the following structure.

$$
\begin{array}{rcl}
e & ::= & \mathbf{new}\langle B\rangle \\
& \mid & \mathbf{checkcast}\langle B\rangle \\
& \mid & \mathbf{getstatic}\langle B : C\rangle \\
& \mid & \mathbf{putstatic}\langle B : C\rangle \\
& \mid & \mathbf{getfield}\langle B : C\rangle \\
& \mid & \mathbf{putfield}\langle B : C\rangle \\
& \mid & \mathbf{invokestatic}\langle B.n : \overline{C} \rightarrow C\rangle \\
& \mid & \mathbf{invokemethod}\langle B.n : \overline{C} \rightarrow C\rangle[B'.n']
\end{array}
$$

The transition rules in Fig. 4 ensure that the production relation $\rightarrow_\Sigma$ observes the parameter policy Σ.

$$\frac{B \triangleright A}{\mathbf{new}\langle B\rangle \in \Sigma[A.m]} \quad \text{(P-New)}$$

$$\frac{B \triangleright A}{\mathbf{checkcast}\langle B\rangle \in \Sigma[A.m]} \quad \text{(P-CheckCast)}$$

$$\frac{C \triangleright A \vee A \bowtie B}{\mathbf{getstatic}\langle B : C\rangle \in \Sigma[A.m]} \quad \text{(P-GetStatic)}$$

$$\frac{C \triangleright B \vee A \bowtie B}{\mathbf{putstatic}\langle B : C\rangle \in \Sigma[A.m]} \quad \text{(P-PutStatic)}$$

$$\frac{C \triangleright A \vee A \bowtie B}{\mathbf{getfield}\langle B : C\rangle \in \Sigma[A.m]} \quad \text{(P-GetField)}$$

$$\frac{C \triangleright B \vee A \bowtie B}{\mathbf{putfield}\langle B : C\rangle \in \Sigma[A.m]} \quad \text{(P-PutField)}$$

$$\frac{\begin{array}{c} n \triangleright m \qquad B \triangleright A \\ C \triangleright A \vee A \bowtie B \\ (\forall i \,.\, C_i \triangleright B) \vee A \bowtie B \vee (B \triangleright m \ \wedge \ \forall i \,.\, C_i \triangleright m) \end{array}}{\mathbf{invokestatic}\langle B.n : \overline{C} \to C\rangle \in \Sigma[A.m]} \quad \text{(P-InvokeStatic)}$$

$$\frac{\begin{array}{c} n \triangleright m \qquad n' \triangleright n \\ C \triangleright A \vee A \bowtie B \\ C \triangleright B \vee B \bowtie B' \qquad (\forall i \,.\, C_i \triangleright B') \vee B \bowtie B' \\ (\forall i \,.\, C_i \triangleright B) \vee A \bowtie B \vee (B \triangleright m \ \wedge \ \forall i \,.\, C_i \triangleright m) \end{array}}{\mathbf{invokemethod}\langle B.n : \overline{C} \to C\rangle[B'.n'] \in \Sigma[A.m]} \quad \text{(P-InvokeMethod)}$$

Fig. 5. A safety policy for DCC

4.2 Modeling DCC in FJVM

We model the type rules of DCC by the safety policy depicted in Fig. 5. ($\mathcal{DCC}1$) is captured in P-InvokeStatic as the premise $B \triangleright A$. ($\mathcal{DCC}2$) is encoded in P-New, P-CheckCast, P-GetStatic, P-PutStatic, P-GetField, P-InvokeStatic and P-InvokeMethod as the premise $C \triangleright A \vee A \bowtie B$, and in P-PutField as the premise $C \triangleright B \vee A \bowtie B$. ($\mathcal{DCC}3$) is enforced by P-InvokeStatic and P-InvokeMethod via the premise $(\forall i \,.\, C_i \triangleright B) \vee A \bowtie B \vee (B \triangleright m \wedge \forall i \,.\, C_i \triangleright m)$. ($\mathcal{DCC}4$) is enforced by P-InvokeStatic and P-InvokeMethod via the premise $n \triangleright m$. ($\mathcal{DCC}6$) is captured in P-InvokeMethod by the three premises $n' \triangleright n$, $C \triangleright B \vee B \bowtie B'$, and $(\forall i \,.\, C_i \triangleright B') \vee B \bowtie B'$. ($\mathcal{DCC}5$) and ($\mathcal{DCC}7$) are not explicitly modeled: ($\mathcal{DCC}5$) is implicitly assumed in our proofs, and ($\mathcal{DCC}7$) is orthogonal to the confinement results to be proven below.

$$\frac{}{\mathit{SafeStack}(A.m, \diamond)} \qquad \frac{n \triangleright m \qquad C \triangleright A \vee A \bowtie B \qquad \mathit{SafeStack}(A.m, \sigma)}{\mathit{SafeStack}(B.n, \mathit{push}(\Phi, A.m, C, \sigma))}$$

Fig. 6. A stack safety judgment

$$\frac{l(B) = \mathcal{D} \qquad \Gamma \vdash B \rightsquigarrow q : C}{\mathit{Accessible}_{[\mathcal{D}]}(q : C \mid \Gamma)} \qquad \frac{l(B) = \mathcal{D} \qquad \Gamma \vdash p : B \rightsquigarrow q : C}{\mathit{Accessible}_{[\mathcal{D}]}(q : C \mid \Gamma)}$$

$$\frac{\Phi \vdash r : C' \qquad C' <: C}{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \Phi)}$$

$$\frac{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \Phi)}{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \mathit{push}(\Phi, A.m, C', \sigma))} \qquad \frac{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \sigma)}{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \mathit{push}(\Phi, A.m, C', \sigma))}$$

$$\frac{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \Gamma)}{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle)}$$

$$\frac{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \Phi)}{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle)} \qquad \frac{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \sigma)}{\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle)}$$

Fig. 7. Accessibility judgments

4.3 Confinement Theorem

A number of judgments are defined to help articulate the main confinement theorem. The *SafeStack* judgment defined in Fig. 6 asserts that the call chain represented by a VM stack has increasingly restrictive capability granting policies, and that capabilities are only returned to callers in the same domain as the callees. The family of *Accessible* judgments defined in Fig. 7 asserts that a labeled reference ($r : C$ or $q : C$) is accessible from a domain ($\mathcal{D}$) in a given VM state.

Our first lemma asserts that stack safety is preserved by the DCC safety policy in Fig. 5.

Lemma 1. *Suppose* $\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi', \Gamma'; \Phi', A'.m', \sigma' \rangle$. *Then* $\mathit{SafeStack}(A.m, \sigma)$ *implies* $\mathit{SafeStack}(A'.m', \sigma')$.

Proof. Only the following cases are relevant.

Case T-InvokeStatic: $\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi', B.n, \sigma' \rangle$ where $\Phi' = \{\overline{r} : \overline{C}\}$ and $\sigma' = \mathit{push}(\Phi, A.m, C, \sigma)$. Suppose $\mathit{SafeStack}(A.m, \sigma)$. Then, by P-InvokeStatic, we have $n \triangleright m$ and $C \triangleright A \vee A \bowtie B$, and thus we deduce $\mathit{SafeStack}(B.n, \sigma')$ as required.

Case T-InvokeMethod: $\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi', B'.n', \sigma' \rangle$ where $\Phi' = \{r_0 : B', \overline{r} : \overline{C}\}$ and $\sigma' = \mathit{push}(\Phi, A.m, C, \sigma)$. Suppose

SafeStack$(A.m, \sigma)$. By P-InvokeMethod, we have $n \triangleright m$ and $n' \triangleright n$, and therefore we obtain $n' \triangleright m$. By P-InvokeMethod again, we have $C \triangleright A \vee A \bowtie B$. If $C \triangleright A$ does not hold, then $A \bowtie B$ must hold, and consequently $C \triangleright B$ does not hold. But then P-InvokeMethod also guarantees $C \triangleright B \vee B \bowtie B'$. We thus obtain $C \triangleright A \vee A \triangleright B'$. Therefore, we deduce *SafeStack*$(B'.n', \sigma')$ as required.

Case T-Return: $\langle \Pi, \Gamma; \Phi', B.n, \mathit{push}(\Phi, A.m, C, \sigma)\rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : C\}, A.m, \sigma\rangle$. By definition *SafeStack*$(B.n, \mathit{push}(\Phi, A.m, C, \sigma))$ implies *SafeStack*$(A.m, \sigma)$. □

Associate with every transition rule is a domain and a set of labeled references that are said to be *active*.

Transition Rule	**Active Domain**	**Active References**
T-Widen	$l(A)$	$r : B$
T-New	$l(A)$	$r : B$
T-CheckCast	$l(A)$	$r : B$
T-GetStatic	$l(A)$	$q : C$
T-PutStatic	$l(B)$	$q : C$
T-GetField	$l(A)$	$q : C$
T-PutField	$l(B)$	$q : C$
T-InvokeStatic	$l(B)$	$\overline{r} : \overline{C}$
T-InvokeMethod	$l(B')$	$r_0 : B', \overline{r} : \overline{C}$
T-Return	$l(A)$	$r : C$

Intuitively, a transition rule has the potential side effect of causing a labeled reference that is active to become accessible in the active domain. A labeled reference or a domain that is not active is said to be *inactive*. Unless for an active domain and a labeled reference that is active, no new labeled reference is made accessible in a domain. This notion is made formal by the following lemma.

Lemma 2. *Suppose* $\langle \Pi, \Gamma; \Phi, A.m, \sigma\rangle \rightarrow_\Sigma \langle \Pi', \Gamma'; \Phi, A'.m', \sigma'\rangle$. *Consider a domain* $\mathcal{D}$ *and a labeled reference* $r : C$. *Unless both* $\mathcal{D}$ *and* $r : C$ *are active for the transition rule that brings about the transition,* $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle)$ *implies* $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \sigma\rangle)$.

Proof. Straightforward case analysis. □

The next lemma is the centerpiece of our confinement result. It asserts that, under the DCC safety policy, if a domain acquires new capabilities as a result of a transition step, then the capability acquisition must conform to the capability granting policy of the execution context.

Lemma 3. *Suppose* $\langle \Pi, \Gamma; \Phi, A.m, \sigma\rangle \rightarrow_\Sigma \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle$ *so that* $\mathit{SafeStack}(A.m, \sigma)$. *Let* $\mathcal{D}$ *be an arbitrary domain. If* $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle)$, *then at least one of the following conditions holds:*

1. $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \sigma\rangle)$ *(previously accessible)*
2. $l(C) \blacktriangleright \mathcal{D}$ *(not a capability)*
3. $C \triangleright m \ \wedge\ \mathcal{D} \blacktriangleright l(m)$ *(controlled capability propagation)*

Proof. We proceed by case analysis on the transition rules in Fig. 4. By Lemma 2, we only need to consider the active domain and the labeled references that are active.

Case T-Widen: Consider the active labeled reference $r : B$ and active domain $l(A)$. By T-Widen, there exists C so that $\Phi \vdash r : C$ and $C <: B$. Thus, condition 1 holds.

Case T-New: Consider the active labeled reference $r : B$ and active domain $l(A)$. By P-New, $B \triangleright A$, and thus condition 2 holds.

Case T-CheckCast: Consider the active labeled reference $r : B$ and active domain $l(A)$. By P-CheckCast, $B \triangleright A$, and thus condition 2 holds.

Case T-GetStatic: Consider the active labeled reference $q : C$ and active domain $l(A)$. By P-GetStatic, either $C \triangleright A$ or $A \bowtie B$. In the former case, condition 2 holds. In the latter, T-GetStatic guarantees $\Gamma \vdash B \rightsquigarrow q : C$, which in turn entails condition 1.

Case T-PutStatic: Consider the active labeled reference $q : C$ and active domain $l(B)$. By P-PutStatic, either $C \triangleright B$ or $A \bowtie B$. In the former case, condition 2 holds. In the latter, T-GetStatic guarantees $\Phi \vdash q : C$, which in turn entails condition 1.

Case T-GetField: Similar to T-GetStatic.

Case T-PutField: Similar to T-PutStatic.

Case T-InvokeStatic: Consider the active domain $l(B)$ and the active labeled references $\overline{r} : \overline{C}$. By P-InvokeStatic, one of $(\forall i \,.\, C_i \triangleright B)$ or $A \bowtie B$ or $(B \triangleright m \ \wedge\ \forall i \,.\, C_i \triangleright m)$ must hold. In the first case, condition 2 is assured. In the second case, condition 1 is assured. In the third case, condition 3 is assured.

Case T-InvokeMethod: Consider the active domain $l(B')$ and the active labeled references $r_0 : B'$ and $\overline{r} : \overline{C}$. For the labeled reference $r_0 : B'$, obviously condition 2 holds. We focus on the labeled references $\overline{r} : \overline{C}$. By P-InvokeMethod, either $(\forall i \,.\, C_i \triangleright B')$ or $B \bowtie B'$. If the former is true, condition 2 is guaranteed. Otherwise, $(\forall i \,.\, C_i \triangleright B')$

does not hold, but $B \bowtie B'$ holds. But then P-InvokeMethod guarantees that one of $(\forall\, .\, C_i \triangleright B)$ or $A \bowtie B$ or $(B \triangleright m \ \wedge\ \forall i\, .\, C_i \triangleright m)$ must hold. The first alternative is impossible because $(\forall i\, .\, C_i \triangleright B')$ is false but $B \bowtie B'$ is true. By $B \bowtie B'$, the second alternative implies $A \bowtie B'$, and thus condition 1 is assured. Again, by $B \bowtie B'$, the third alternative implies $(B' \triangleright m \ \wedge\ \forall i\, .\, C_i \triangleright m)$, and thus condition 3 is assured.

Case T-Return: Consider the active domain $l(A)$ and the active labeled references $r : C$. Because $\mathit{SafeStack}(B.n, \mathit{push}(\Phi, A.m, C, \sigma))$, either $C \triangleright A$ or $A \bowtie B$. In the former case, condition 2 holds. In the latter, since T-Return guarantees $\Phi' \vdash r : C$, where Φ' is the stack frame of B, we therefore conclude that condition 1 holds. □

We are now ready to state and prove the main confinement theorem.

Theorem 4 (Discretionary Capability Confinement). *Suppose* $\langle \Pi, \Gamma; \Phi, A.m, \diamond\rangle \xrightarrow{*}_{\Sigma} \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle$. *Let* $\mathcal{D}$ *be an arbitrary domain. If* $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle)$, *then at least one of the following conditions holds:*

1. $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \diamond\rangle)$ *(previously accessible)*
2. $l(C) \blacktriangleright \mathcal{D}$ *(not a capability)*
3. $C \triangleright m \ \wedge\ \mathcal{D} \blacktriangleright l(m)$ *(controlled capability propagation)*

Proof. We proceed by straightforward induction on the number of state transitions.

Base Case: $\langle \Pi, \Gamma; \Phi, A.m, \diamond\rangle = \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle$. Then condition 1 is given by the assumption of the theorem.

Induction Step: There is a state $\langle \Pi'', \Gamma''; \Phi'', A''.m'', \sigma''\rangle$, such that $\langle \Pi, \Gamma; \Phi, A.m, \diamond\rangle \xrightarrow{*}_{\Sigma} \langle \Pi'', \Gamma''; \Phi'', A''.m'', \sigma''\rangle$ and $\langle \Pi'', \Gamma''; \Phi'', A''.m'', \sigma''\rangle \rightarrow_{\Sigma} \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle$. Suppose that condition 2 and 3 do not hold. Since $\mathit{SafeStack}(A.m, \diamond)$, we conclude from Lemma 1 that $\mathit{SafeStack}(A''.m'', \sigma'')$. Therefore, $m'' \triangleright m$. Now, it cannot be the case that $C \triangleright m'' \ \wedge\ \mathcal{D} \blacktriangleright l(m'')$, or else, by $m'' \triangleright m$, it contradicts with the assumption that condition 3 does not hold. By Lemma 3, we deduce $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi'', \Gamma''; \Phi'', A''.m'', \sigma''\rangle)$. By the induction hypothesis, we further obtain condition 1 as required. □

4.4 Theft and Leakage

In the following, we identify annotation practices that preserve useful confinement properties. Specifically, a method $A.m$ is said to be ***safe*** iff

$m \triangleright A$. *Executing a safe method $A.m$ will only cause those domains dominated by $l(A)$ to acquire capabilities that A can generate.* Programmers concerned with capability confinement may then arrange their code to invoke untrusted software extensions only via safe method interfaces.

Theft. Capability theft occurs when executing code in a domain causes the domain to acquire capabilities it does not already possess. The absence of theft makes capabilities unforgeable. Theorem 4 entails that executing safe methods always guarantees the absence of capability theft.

Corollary 5 (No Theft). *Suppose $m \triangleright A$ and $\langle \Pi, \Gamma; \Phi, A.m, \diamond \rangle \xrightarrow{*}_{\Sigma} \langle \Pi', \Gamma'; \Phi', A'.m', \sigma' \rangle$. Let $\mathcal{D}$ be $l(A)$. If $Accessible_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma' \rangle)$, then at least one of the following conditions holds:*

1. $Accessible_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \diamond \rangle)$ *(previously accessible)*
2. $l(C) \blacktriangleright \mathcal{D}$ *(not a capability)*

Proof. By way of contradiction, suppose conditions 1 and 2 do not hold. Theorem 4 implies $C \triangleright m$, which, by $m \triangleright A$, in turn implies $C \triangleright A$, contradicting the assumption that condition 2 does not hold. □

Intuitively, the above statement says, if the execution of a safe method in a domain causes the domain to acquire new references, those references are never capabilities.

Leakage. Capability leakage occurs when executing code in a domain causes a foreign domain to acquire a capability that the foreign domain does not already possess. When a capability is granted to a domain, it is in the interest of the granter that the grantee will not leak the granted capability. Theorem 4 entails that a safe method never leaks capabilities to a domain that is not dominated by the home domain of the method.

Corollary 6 (No Leakage). *Suppose $m \triangleright A$ and $\langle \Pi, \Gamma; \Phi, A.m, \diamond \rangle \xrightarrow{*}_{\Sigma} \langle \Pi', \Gamma'; \Phi', A'.m', \sigma' \rangle$. Let $\mathcal{D}$ be a domain such that $\mathcal{D} \blacktriangleright l(A)$ is not true. If $Accessible_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma' \rangle)$, then at least one of the following conditions holds:*

1. $Accessible_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \diamond \rangle)$ *(previously accessible)*
2. $l(C) \blacktriangleright \mathcal{D}$ *(not a capability)*

Proof. By way of contradiction, suppose conditions 1 and 2 do not hold. Theorem 4 implies $\mathcal{D} \blacktriangleright l(m)$, which, by $m \triangleright A$, in turn implies $\mathcal{D} \blacktriangleright l(A)$, contradicting the assumption of the corollary. □

Mutual Suspicion. Suppose A and B are mutually suspicious, and a safe method $A.m$ is invoked. By Corollary 5, no reference of type B will be acquired by A as a result of the invocation. Similarly, by Corollary 6, no reference of type A will be acquired by B. Consequently, mutually suspicious types never exchange capabilities as a result of invoking safe methods. If the two types have never been explicitly granted capabilities of one another, then they cannot invoke methods declared in each others type interface. Collusion of this kind is therefore completely eliminated.

5 Extensions and Variations

5.1 Accommodating Other Language Constructs

Arrays. The array types $C[\,]$, $C[\,][\,]$, ... are said to be ***carrier types*** for declared type C. An object reference with a carrier type is a ***carrier***. If $\mathcal{D}$ acquires a carrier (e.g., of type $C[\,]$) for a capability type C, while $\mathcal{E}$ obtains a carrier-type reference (e.g., of type `Object`$[\,]$) to the same object, then $\mathcal{E}$ can store references into the carrier, while $\mathcal{D}$ can retrieve the said references as type-C capabilities. Special type constraints must be introduced into DCC to avoid the misuse of carriers as covert channels for capability communication.

In general, $\mathcal{E}$ can communicate capabilities to $\mathcal{D}$ via an array only if (i) the array has been acquired by $\mathcal{D}$ as a *capability carrier*, while (ii) $\mathcal{E}$ acquires a *carrier alias* of that array. Corresponding to these two necessary conditions are two approaches to handle arrays in DCC:

Approach I: *Permit (i) but not (ii).* A domain is allowed to instantiate and share capability carriers, but no carrier alias may be created across domain boundaries.
- Instantiation of carriers (including capability carriers) is freely allowed, but dynamic casting of a reference into a carrier is not allowed (even if the carrier is not a capability carrier).
- Sharing and granting of carriers across domain boundaries are not allowed.
- A method with carrier-type formal parameters and/or return value can only be overriden by methods declared in the same domain.

Approach II: *Permit (ii) but not (i):* Carrier aliases are allowed to be created across domain boundaries, but a domain must not acquire any capability carrier.
- Generation of capability carriers is not allowed.
- Sharing and granting of capability carriers across domain boundaries are not allowed.

- If the return type of a method is a carrier type for a capability type, then the method can only be overriden by a method declared in the same domain.
- If the type of a method formal parameter is a carrier type for a capability type, then the method can only override a method declared in the same domain.

Although either approach is secure, Approach II is preferred because it is backward compatible to pure Java: a Java program in which all declared types belong to the root domain $\top$ trivially satisfies all type constraints.

To implement Approach II, some minor changes are introduced to the type constraints in Sect. 3.2. Specifically, the array type $C[\,]$ is considered to be as capable as C: i.e., $C \bowtie C[\,]$. Under this assumption, almost all of the type constraints in Sect. 3.2 can be reused as is, except for one, namely, ($\mathcal{DCC}3$). The type constraint must be revised to explicitly forbid the granting of capability carriers:

($\mathcal{DCC}3'$) **Discretionary capability granting.** If $A.m$ invokes $B.n$, and C is the type of a formal parameter of n, then $C \triangleright B \vee A \bowtie B \vee (B \triangleright m \wedge C \triangleright m \wedge$ *"C is not an array"*).

Genericity. Genericity does not present any security challenge to the present design of DCC[5]. Genericity is a purely source-level construct that is translated into bytecode via type erasure. The source-level generic type `Set<`C`>` is translated into the raw reference type `Set`. Set members are retrieved as `Object` references. The compiler introduces a dynamic cast to convert the retrieved `Object` reference into a type-C reference. There are two impliciations to this set up. Firstly, because generic containers such as `Set` belong to the root domain, DCC permits the acquisition and transmission of capability containers. Secondly, if C is a capability type, then P-CheckCast will effectively forbid the retrieval of any type-C capabilities from generic containers. This is consistent with the overall design philosophy of DCC: capability acquisition must only occur as a result of explicit granting (i.e., argument passing). In summary, there is no security motivation for imposing any additional type constraint to account for genericity.

[5] In an earlier version of this manuscript, genericity was mistaken to be a challenge to DCC. Specifically, the acquisition of a capability container `Set<`C`>` was incorrectly considered to be equivalent to the acquisition of all the type-C capabilities that the container carries. In this view, the dynamic casts introduced by the compilation process impose unnecessary constraints (via P-CheckCast) over the retrieval of capabilities that are *already* acquired. This understanding was later found to be inconsistent with the notion of accessibility as specified in Fig. 7.

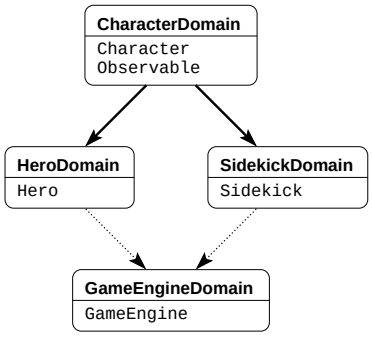

Fig. 8. Revised dominance hierarchy for the hero-sidekick application. Dotted arrows represent "dominated-by" relationships (►). Solid arrows represent "strongly-dominated-by" relationships (:►)

Nested Types. Nested types do not pose any difficulty to the present formulation of DCC, as they are mainly source-level constructs, and are encoded at the bytecode level as regular JVM classes. It is, however, natural for a reference type and its nested types to belong to the same confinement domain in order for them to collaborate properly. A compile-time type checker may volunteer to issue a warning if this is not the case, but failing to do so does not post any security threat at run time.

5.2 Modular Enforcement of Hereditary Mutual Suspicion

Hereditary mutual suspicion ($\mathcal{DCC}7$) interacts with dynamic linking in a non-trivial manner. Specifically, ($\mathcal{DCC}7$) is universally quantified over all subtypes of two mutually exclusive roles. The enforcement of ($\mathcal{DCC}7$) thus involves a time complexity quadratic to the number of subtypes of the mutually exclusive roles, making it very inefficient. Worst still, because of the dynamic linking semantics of the JVM, some of these subtypes may not have been completely loaded, making it impossible to enforce ($\mathcal{DCC}7$) at link time. This section addresses the above two issues by examining a reformulation of ($\mathcal{DCC}7$) that facilitates ***modular enforcement***. We begin by rephrasing ($\mathcal{DCC}7$) in a more succinct and yet equivalent form, involving only three participants:

($\mathcal{DCC}7'$) **Hereditary mutual suspicion (2nd form).** Suppose A and B are mutually suspicious. If A' is a subtype of A, then A' and B are also mutually suspicious.

We show that ($\mathcal{DCC}7$) follows from ($\mathcal{DCC}7'$) (the other direction is obvious). Suppose A' and B' are subtypes of A and B respectively. It

suffices to demonstrate that, if A' and B' are not mutually suspicious, then $(\mathcal{DCC}7')$ implies A and B are not mutually suspicious. If $B' \triangleright A'$, then, by $(\mathcal{DCC}5)$, $B \triangleright A'$, and thus it follows from $(\mathcal{DCC}7')$ that A and B are not mutually suspicious. The case for $A' \triangleright B'$ is symmetrical.

We adopt a conservative design that facilitates the modular enforcement of hereditary mutual suspicion $(\mathcal{DCC}7')$. Specifically, we want to be able to check a reference type A only once at link time, and then conclude that it will not participate in the violation of $(\mathcal{DCC}7')$ in the future. To this end, we (1) lift the reasoning of mutual suspicion from the level of reference types to the level of confinement domains, and (2) capture in a binary relation the sufficient condition by which mutual suspicion is preserved in subtyping. A programmer-supplied partial order $:\blacktriangleright$ is postulated, so that:

$$
\begin{array}{ll}
(\mathcal{HMS}1) & \top :\blacktriangleright \mathcal{D} \\
(\mathcal{HMS}2) & \mathcal{D} :\blacktriangleright \mathcal{E} \Rightarrow \mathcal{D} \blacktriangleright \mathcal{E} \\
(\mathcal{HMS}3) & (\mathcal{D} :\blacktriangleright \mathcal{E} \wedge \mathcal{D}' \blacktriangleright \mathcal{E}) \Rightarrow (\mathcal{D} \blacktriangleright \mathcal{D}' \vee \mathcal{D}' \blacktriangleright \mathcal{D})
\end{array}
$$

We say that $\mathcal{D}$ ***strongly dominates*** $\mathcal{E}$ whenever $\mathcal{E} :\blacktriangleright \mathcal{D}$. The $:\blacktriangleright$ relation induces a pre-ordering of Java reference types: we write $B :\triangleright A$ iff $l(B) = \mathcal{E}$, $l(A) = \mathcal{D}$ and $\mathcal{E} :\blacktriangleright \mathcal{D}$. It follows readily from definition that $:\triangleright$ is reflexive and transitive, and $B :\triangleright A \Rightarrow B \triangleright A$. We restate hereditary mutual suspicion in a form that facilitates modular enforcement.

$(\mathcal{DCC}7'')$ **Hereditary mutual suspicion (3rd form).** If $A <: B$, then $B :\triangleright A$.

We show that $(\mathcal{DCC}7')$ follows from $(\mathcal{DCC}7'')$, $(\mathcal{HMS}2)$ and $(\mathcal{HMS}3)$. Suppose A and B are mutually suspicious, so that $l(A) = \mathcal{D}$ and $l(B) = \mathcal{D}'$. Suppose further that $A' <: A$ and $l(A') = \mathcal{E}$. By $(\mathcal{DCC}7'')$, we have $\mathcal{D} :\blacktriangleright \mathcal{E}$. Applying the contrapositive of $(\mathcal{HMS}3)$, $\mathcal{D}' \blacktriangleright \mathcal{E}$ does not hold. Similarly, $\mathcal{E} \blacktriangleright \mathcal{D}'$ does not hold, or else $(\mathcal{HMS}2)$ and the transitivity of $\blacktriangleright$ would imply $\mathcal{D} \blacktriangleright \mathcal{D}'$, a contradiction. Consequently, A' and B are mutually suspicious as required by $(\mathcal{DCC}7')$.

Notice that $(\mathcal{DCC}7'')$ is sound but incomplete, meaning that programs satisfying $(\mathcal{DCC}7'')$ are guaranteed to satisfy $(\mathcal{DCC}7')$, but some programs satisfying $(\mathcal{DCC}7')$ may not satisfy $(\mathcal{DCC}7'')$. We trade completeness for tractability: $(\mathcal{DCC}7'')$ can be enforced even in the presence of dynamic loading, and, as the number of confinement domains is much fewer than the number of reference types, $(\mathcal{DCC}7'')$ can be enforced efficiently.

A revised dominance hierarchy of the hero-sidekick game application is given in Fig. 8.

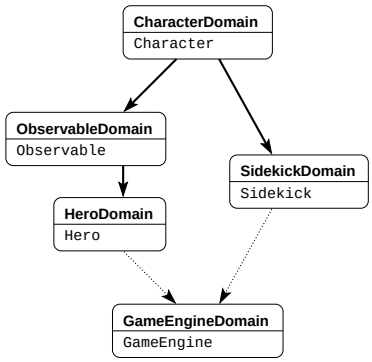

Fig. 9. An alternative dominance hierarchy for the hero-sidekick application

5.3 Finer-Grained Control of Capability Granting

The capability granting policy $l(m)$ plays two roles: it controls what capabilities can be granted when m is executing, and to whom shall these capabilities be granted. In previous discussion, we unified these two roles into one label to simplify exposition. In the following, we explore an alternative design that leads to a finer-grained control of capability granting. As we shall see, this flexibility supports a controlled form of capability exchange between mutually suspicious reference types. To motivate this, consider the alternative dominance hierarchy for the hero-sidekick as shown in Fig. 9. In this set up, `Observable` becomes a capability for `Sidekick`. In order for `Hero.broadcast` to communicate `Observable` references to `Sidekick.update`, we will have to adopt the annotations $l($`broadcast`$)$ = `GameEnginDomain` and $l($`update`$)$ = `SidekickDomain`. This configuration is acceptable in our example, in which the `broadcast` method is composed of only three lines of code that invokes only `update`. The configuration, however, violates the principle of least privilege [36], especially for a complex `broadcast` method that may invoke methods other than `update`. It would be desirable if we can impose a capability granting policy that allows, and only allows, `broadcast` to grant `Observable` capabilities to the `SidekickDomain`.

We begin with separating the two roles into separate labels. Specifically, the capability granting policy of a method m is specified by two domain labels: $t(m)$, the ***target*** of m, which specifies what domains may acquire capabilities as a result of executing m, and $p(m)$, the ***potency*** of m, which specifies what capabilities may be granted when m is executed. More specifically, ($\mathcal{DCC}3'$) is modified to reflect this separation of roles:

$$\frac{\begin{array}{c} t(n) \blacktriangleright t(m) \qquad p(n) \blacktriangleright p(m) \qquad B \triangleright A \\ C \triangleright A \vee A \bowtie B \\ (\forall i \,.\, C_i \triangleright B) \vee A \bowtie B \vee (l(B) \blacktriangleright t(m) \ \wedge \ \forall i \,.\, l(C_i) \blacktriangleright p(m)) \end{array}}{\mathbf{invokestatic}\langle B.n : \overline{C} \to C\rangle \in \Sigma[A.m]} \text{ (P-InvokeStatic}')$$

$$\frac{\begin{array}{c} t(n) \blacktriangleright t(m) \qquad p(n) \blacktriangleright p(m) \\ t(n') \blacktriangleright t(n) \qquad p(n') \blacktriangleright p(n) \\ C \triangleright A \vee A \bowtie B \\ C \triangleright B \vee B \bowtie B' \qquad (\forall i \,.\, C_i \triangleright B') \vee B \bowtie B' \\ (\forall i \,.\, C_i \triangleright B) \vee A \bowtie B \vee (l(B) \blacktriangleright t(B) \ \wedge \ \forall i \,.\, l(C_i) \blacktriangleright p(m)) \end{array}}{\mathbf{invokemethod}\langle B.n : \overline{C} \to C\rangle[B'.n'] \in \Sigma[A.m]} \text{ (P-InvokeMethod}')$$

Fig. 10. Revised safety policy for DCC

($\mathcal{DCC}3''$) **Discretionary capability granting.** If $A.m$ invokes $B.n$, and C is the type of a formal parameter of n, then $C \triangleright B \vee A \bowtie B \vee (\underline{l(B) \blacktriangleright t(m)} \wedge \underline{l(C) \blacktriangleright p(m)} \wedge$ *"C is not an array"*).

The rules ($\mathcal{DCC}4$) and ($\mathcal{DCC}6$) are adapted accordingly.

($\mathcal{DCC}4'$) **No Amplification of Rights.** A method m may invoke another method n only if $\underline{t(n) \blacktriangleright t(m)}$ and $\underline{p(n) \blacktriangleright p(m)}$.

($\mathcal{DCC}6'$) **Impersonation Avoidance.** Suppose $B.n$ is overridden by $B'.n'$.

1. $\underline{t(n') \blacktriangleright t(n)}$ and $\underline{p(n') \blacktriangleright p(n)}$.
2. *[No change.]*
3. *[No change.]*

The safety policy rules P-InvokeStatic and P-InvokeMethod are revised to reflect this change. The resulting formulation is depicted in Fig. 10. The reformulated safety policy rules allow us to establish a variant of our main confinement theorem.

Theorem 7 (Discretionary Capability Confinement (Revised)). *Suppose $\langle \Pi, \Gamma; \Phi, A.m, \diamond\rangle \xrightarrow{*}_{\Sigma} \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle$. Let $\mathcal{D}$ be an arbitrary domain. If $Accessible_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle)$, then at least one of the following conditions holds:*

1. $Accessible_{[\mathcal{D}]}(r : C \mid \langle \Pi, \Gamma; \Phi, A.m, \diamond\rangle)$ *(previously accessible)*
2. $l(C) \blacktriangleright \mathcal{D}$ *(not a capability)*
3. $\underline{l(C) \blacktriangleright p(m) \ \wedge \ \mathcal{D} \blacktriangleright t(m)}$ *(controlled capability propagation)*

The proof of this theorem is a straightforward adaptation of the proof for Theorem 4. The theorem above legitimize the annotations suggested in the beginning of this subsection.

```
@Domain
public interface CharacterDomain extends Root { }
@Domain( allowSubtyping = { CharacterDomain.class } )
public interface HeroDomain extends CharacterDomain { }
@Domain( allowSubtyping = { CharacterDomain.class } )
public interface SidekickDomain extends CharacterDomain { }
@Domain
public interface GameEngineDomain extends HeroDomain, SidekickDomain { }
```

Fig. 11. An interface hierarchy representing the dominance hierarchy of the hero-sidekick game application

6 Implementation Experience

6.1 Source-Level Annotations

Although DCC is formulated and enforced at the bytecode level, a specification mechanism has been devised to facilitate the annotation of Java source files with such DCC typing information as domain membership ($l(C)$), capability granting policy ($l(m)$), dominance relationship ($\blacktriangleright$), and strong dominance relationship ($:\blacktriangleright$). These source-level annotations are encoded using the JDK 5.0 metadata facility. For example, Fig. 11 illustrates how the domain hierarchy in Fig. 8 is encoded at the source level as an interface hierarchy. Specifically, a confinement domain is represented as an empty public interface with a `@Domain` annotation. The subsumption relation is represented by interface extension: if a domain interface $\mathcal{E}$ extends another domain interface $\mathcal{D}$, then $\mathcal{D} \blacktriangleright \mathcal{E}$. The root domain $\top$ is represented by the predefined domain interface `Root`, which must be a superinterface of every user-defined domain interface. Strong subsumption is specified via the `allowSubtyping` element of a `@Domain` annotation. Specifically, the value of an `allowSubtyping` element is a list of domain interfaces. If domain interface $\mathcal{D}$ appears in the `allowSubtyping` list of domain interface $\mathcal{E}$, then we intend it to mean $\mathcal{D} :\blacktriangleright \mathcal{E}$. If no `allowSubtyping` element is supplied, then, by default, the domain interface is strongly dominated only by `Root`. Lastly, domain membership and capability granting policies (without the extension of Sect. 5.3) are indicated by the `@Confined` and `@Grants` annotations respectively. For example, the following declaration confines the `Robin` class to `SidekickDomain` and sets the capability granting policy of the `update` method to `SidekickDomain`:

```
@Confined( SidekickDomain.class )
public class Robin implements Sidekick {
   @Grants( SidekickDomain.class )
```

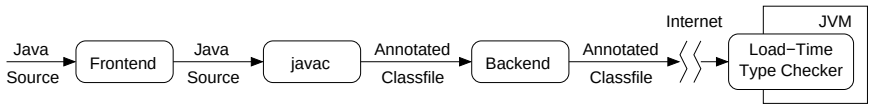

Fig. 12. The DCC software development environment

```
    public void update(Observable hero);
}
```

6.2 Type Checkers

The present design of DCC is optimized for enforcement efficiency, and as such it requires no iterative analysis of method bodies. All type constraints are enforced by a linear-time scan of classfiles.

We envision a programming environment (Fig. 12) in which Java source files embedded with DCC annotations are partially validated by a compiler frontend, and subsequently translated into annotated classfiles by the JDK 5.0 compiler. The annotated classfiles are then type-checked at the bytecode level by a compiler backend prior to shipping. To guard against malicious code generators, type checking is also conducted by the JVM at load time, against classfiles, at the bytecode level. All the three DCC type checkers depicted in Fig. 12 have been implemented. The ***frontend*** component is a source-level type checker based on the JDK 5.0 annotation processing tool (`apt`). It ensures that the type interface of Java classes and interfaces conform to type constraints ($\mathcal{DCC}5$), ($\mathcal{DCC}6$) ($\mathcal{DCC}7''$), as well as the $\mathcal{HMS}$ rules. The ***backend*** component is an offline, bytecode-level type checker based on the Apache ByteCode Engineering Library (BCEL). It ensures that classfiles or JAR files conform to *all* the type constraints. Lastly, the ***load-time type checker*** is obtained by embedding the backend type checking engine into a Java class loader, which type-checks classfiles as they are loaded into the JVM.

7 Concluding Remarks

7.1 Related Work

Language-Based Capability Systems. Previously proposed language-based capability systems [17, 14, 15] lack confinement guarantees. This work combines the ideas of confinement domains [22] and capability granting policies [26] to achieve capability confinement.

Object Reference Confinement. The design of DCC has been influenced by the notion of confined types [22–25], the systematic enforcement of which could have eliminated a security breach in JDK 1.1. While the confinement boundaries of confined types are absolute and uniform, those in DCC are semi-permeable and discriminatory, allowing reference acquisition through dominance and capability granting through discretion. This difference is due to the fact that confined types is designed to uniformly confine all object references of a given *concrete class*, but DCC is designed to selectively confine those object references that would otherwise escape with a privileged *static type*. While Vitek *et al* identify confinement domains with Java packages, thereby reusing the access control semantics of package private members, the dominance hierarchy of DCC is independent of the Java package semantics. Also, the type rules for confined types block reference leakages at their originating sites, while the type rules of DCC target illegal reference acquisition at the receiving ends. This shift of focus is motivated by the need of discriminatory confinement induced by the trust relation. Lastly, bytecode-level implementation of confined types involves iterative flow analysis; link-time type checking of DCC does not.

The static type system pop [37] supports the reference-as-capability metaphor in an inheritance-less object calculus. Contrary to "communication-based" schemes of object confinement (e.g., confined types), an "used-based" approach has been adopted by pop to impose a custom "user interface" over an object. The user interface specifies how individual protection domains may access the object. DCC can be seen as a hybrid of communication-based and use-based approaches to capabilities: use-based views are modeled as static types imposed on references, and references may only escape from a confinement domain so long as they do not escape with a view that grants privileged accesses to the receiving domain.

Encapsulation Policies. Modern object-oriented programming languages provide access modifiers such as `protected` and `private` to control the visibility of reference types and their members. DCC could be seen as a static access control scheme in which the accessibility of a reference type or its members can be controlled more precisely than the traditional set of access modifiers. In this respect, this work shares with Composable Encapsulation Policies (CEP) [38, 39] the concern of providing alternative interfaces to multiple client categories. Therefore, the role of encapsulation policies is analogous to capabilities in DCC. Schärli *et al* [38] rightly observe that Java interfaces could have been used for modeling

encapsulation policies had it not been the fact that encapsulation policies modeled as such are not enforceable. Under DCC, encapsulation policies expressed as Java interfaces *can* indeed be enforced as capabilities. DCC can therefore be seen as a non-intrusive augmentation to Java that turns type interfaces into enforceable encapsulation policies.

Concealment of Concrete Type. The Emerald programming language [40] is an early object-oriented language for distributed programming. It provides a `restrict` type qualifier for concealing the concrete type of an object reference (i.e., preventing downcasting). This feature provides partial support for the reference-as-capability metaphor.

Stack Inspection. Stack inspection [5] is an access control model for program execution that involves code units belonging to distinct protection domains. A common assumption behind the many models of stack inspection [5–7, 9] and its variants [8] is that the binding of permissions to code units is performed statically. While this simplification has the clear advantage of supporting declarative reasoning of access control, it does not support ***dynamic access control policies*** [41, Sect. 5.6], in which the set of permissions associated with a protection domain may evolve over time. DCC identifies protection domains with confinement domains. While the binding of code units to their protection domains (i.e., confinement domains) is performed statically, the granting of permissions to protection domains occurs dynamically through capability acquisition. Notice, however, *the right to grant capability* is still modeled statically in DCC. A close examination of the *SafeStack* invariant reveals that DCC maintains a stack invariant very similar to that of stack inspection. Precise formulation of this connection belongs to future work (see next section).

Language-Based Information Flow Control. Although this work is primarily concerned with access control, and thus orthogonal to language-based information flow control [42], one may see the **No Theft** and **No Leakage** properties as playing the analogous roles of **Simple Security** and ***-Property** in information flow control.

Separation of Duty. The principle of separation of duty is foundational in ensuring system integrity [27]. The original intention is to avoid an untrusted external agent from single-handedly contaminating the integrity of the system. Postulating mutually-exclusive roles is a popular means [43, 29] for enforcing separation of duty statically [28]. By requiring multiple external agents to be involved in carrying out a task, it is assumed that

collusion between external agents is *unlikely*. Similarly, our work aims to avoid the collusion of untrusted code units. The hereditary mutual suspicion constraint not only support mutual exclusion of roles, it *provably* eliminates collusion between them. To the best of the author's knowledge, this is the first work to enforce such a strong form of separation of duty in a language-based environment.

7.2 Future Work

In Sect. 5.3, adopting a finer-grained representation of capability granting policies leads to interesting capability communication idioms. An extension is to explore alternative representations of capability granting policies, and study the collaboration idioms thus enabled.

Typing rule ($\mathcal{DCC}4$) mandates that the right to grant capability is always diminishing along a call chain. This requirement not only restricts the reusability of methods, but also causes methods deep in a call chain to be deprived of capability granting rights. Is it possible to permit amplification of capability granting right while preserving the confinement properties? One helpful observation is that the reasoning of capability granting rights in DCC is akin to stack inspection: introduction of every new stack frame further restricts the right of capability granting, and we seek a mechanism that allows privileged frames to tentatively amplify this right. It is likely that the extensive experience of the research community on stack inspection can be exploited to help address this challenge.

A limitation of this work is the lack of support for capability revocation. It is obviously impossible to "revoke" a capability reference that has already been acquired by a protection domain. The lack of revocation can be alleviated by carefully regulating authority delegation. Constrained delegation is a well-studied topic in the context of role-based access control and trust management (see, particularly, [44–46]). First ideas of how delegation constraints may be integrated into a capability type system can be found in [47].

7.3 Conclusion

A lightweight, non-intrusive, static annotation system, DCC, has been proposed to model capabilities in a Java-like language environment. We have shown that, unlike previous language-based capability systems, DCC enforces capability confinement guarantees such as no theft and no leakage. Leveraging the confinement guarantees, DCC can model a collaboration protocol known as hereditary mutual suspicion, which can be seen as

a strong form of role separation. A suite of development tools, including a load-time type checker, have been implemented. This work suggests a number of future directions, including the connection of DCC with stack inspection and constrained delegation.

Acknowledgments. This work was supported by an NSERC Discovery Grant.

References

1. Carzaniga, A., Picco, G.P., Vigna, G.: Designing distributed applications with mobile code paradigms. In: Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA (1997) 22–32
2. Schneider, F.B., Morrisett, G., Harper, R.: A language-based approach to security. In: Informatics: 10 Years Back, 10 Years Ahead. Volume 2000 of LNCS. Springer (2000) 86–101
3. Edjlali, G., Acharya, A., Chaudhary, V.: History-based access control for mobile code. In: Proceedings of the 5th ACM Conference on Computer and Communications Security, San Francisco, California, USA (1998) 38–48
4. Gong, L., Schemers, R.: Implementing protection domains in the Java development kit 1.2. In: Proceedings of the Internet Society Symposium on Network and Distributed System Security, San Diego, California, USA (1998) 125–134
5. Wallach, D.S., Appel, A.W., Felten, E.W.: SAFKASI: A security mechanism for language-based systems. ACM Transactions on Software Engineering and Methodology **9**(4) (2000) 341–378
6. Úlfar Erlingsson, Schneider, F.B.: IRM enforcement of Java stack inspection. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, Berkeley, California (2000) 246–255
7. Fournet, C., Gordon, A.D.: Stack inspection: Theory and variants. ACM Transactions on Programming Languages and Systems **25**(3) (2003) 360–399
8. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the 10th Annual Network and Distributed System Security Symposium, San Diego, California, USA (2003)
9. Pottier, F., Skalka, C., Smith, S.: A systematic approach to static access control. ACM Transactions on Programming Languages and Systems **27**(2) (2005) 344–382
10. Dennis, J.B., Horn, E.C.V.: Programming semantics for multiprogrammed computations. Communications of the ACM **9**(3) (1966) 143–155
11. Miller, M.S., Yee, K.P., Shapiro, J.: Capability myths demolished. Technical Report SRL2003-02, System Research Lab, Department of Computer Science, The John Hopkins University (2003)
12. Schroeder, M.D.: Cooperation of Mutually Suspicious Subsystems in a Computer Utility. Ph.D. thesis, MIT (1972)
13. Rees, J.A.: A security kernel based on the lambda-calculus. A. I. Memo 1564, MIT (1996)
14. Wallach, D.S., Balfanz, D., Dean, D., Felten, E.W.: Extensible security architectures for Java. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint Malo, France (1997) 116–128

15. Hawblitzel, C., Chang, C.C., Czajkowski, G., Hu, D., von Eicken, T.: Implementing multiple protection domains in Java. In: Proceedings of the USENIX Annual Technical Conference, New Orleans, Louisiana, USA (1998)
16. Chander, A., Dean, D., Mitchell, J.C.: A state-transition model of trust management and access control. In: Proceedings of the 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada (2001) 27–43
17. Jones, A.K., Liskov, B.H.: A language extension for expressing constraints on data access. Communications of the ACM **21**(5) (1978) 358–367
18. Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: A generalization of uniqueness and read-only. In: Proceedings of the 2001 European Conference on Object-Oriented Programming, Budapest, Hungary (2001) 2–27
19. Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, USA (1999) 262–275
20. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. 3rd edn. Addison Wesley (2000)
21. ECMA: Standard ECMA-335: Common Language Infrastructure (CLI). 2nd edn. (2002)
22. Vitek, J., Bokowski, B.: Confined types in Java. Software - Practice & Experience **31**(6) (2001) 507–532
23. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tampa Bay, FL, USA (2001) 241–253
24. Zhao, T., Palsberg, J., Vitek, J.: Lightweight confinement for featherweight Java. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, California, USA (2003) 135–148
25. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. Journal of Functional Programming **16**(1) (2006) 83–128
26. Gong, L.: A secure identity-based capability system. In: Proceedings of the 1989 IEEE Symposium on Security and Privacy, Oakland, California, USA (1989) 56–63
27. Clark, D.D., Wilson, D.R.: A comparison of commercial and military computer security policies. In: Proceedings of the 1987 IEEE Symposium on Security and Privacy. (1987) 184–194
28. Li, N., Bizri, Z., Tripunitara, M.V.: On mutually-exclusive roles and separation of duty. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, Washington DC, USA (2004) 42–51
29. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security **4**(3) (2001) 224–274
30. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley (1994)
31. Hardy, N.: The confused deputy: or why capabilities might have been invented. Operating Systems Review **22**(4) (1988) 36–38
32. Lipton, R.J., Snyder, L.: A linear time algorithm for deciding subject security. Journal of the ACM **24**(3) (1977) 455–464
33. Sandhu, R.S.: The schematic protection model: Its definition and analysis for acyclic attenuating schemes. Journal of the ACM **35**(2) (1988) 404–432

34. Sandhu, R.S.: The typed access matrix model. In: Proceedings of the 1992 IEEE Symposium on Security and Privacy. (1992) 122–136
35. Fong, P.W.L.: Reasoning about safety properties in a JVM-like environment. Technical Report CS-2006-02, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada (2006)
36. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE **63**(9) (1975) 1278–1308
37. Skalka, C., Smith, S.: Static use-based object confinement. International Journal of Information Security **4**(1–2) (2005) 87–104
38. Schärli, N., Ducasse, S., Nierstrasz, O., Wuyts, R.: Composable encapsulation policies. In: Proceedings of the 18th European Conference on Object-Oriented Programming, Oslo, Norway (2004)
39. Schärli, N., Black, A.P., Ducasse, S.: Object-oriented encapsulation for dynamically typed languages. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, B.C., Canada (2004) 130–149
40. Hutchinson, N.C., Raj, R.K., Black, A.P., Levy, H.M., Jul, E.: The Emerald programming language report. Technical Report 87-10-07, Department of Computer Science, University of Washington (1987)
41. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 Platform Security. 2nd edn. Addison Wesley (2003)
42. Sabelfeld, A., Meyers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1) (2003) 5–19
43. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer **29**(2) (1996) 38–47
44. Bandmann, O., Dam, M., Firozabadi, B.S.: Constrained delegation. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Berkeley, California (2002) 131–140
45. Li, N., Grosof, B.N., Feigenbaum, J.: Delegation logic: A logic-based approach to distributed authorization. ACM Transactions on Information and System Security **6**(1) (2003) 128–171
46. Wainer, J., Kumar, A.: A fine-grained, controllable, user-to-user delegation method in RBAC. In: Proceedings of the 10th ACM Symposium on Access Control Models and Technologies, Stockholm, Sweden (2005) 59–66
47. Fong, P.W.L., Zhang, C.: Capabilities as alias control: Secure cooperation in dynamically extensible systems. Technical Report CS-2004-03, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada (2004) ISBN 0-7731-0479-8.