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 augment its feature set. 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 [3,4,5,6,7,8,9]. 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 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]. 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 [13] or load-time binary rewriting [14] 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 [13,15] 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 [16]. In a ***capability type system*** [17,18], 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 [19,20] 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* [21,22,23,24], 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 [25], 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*** [26,27]: 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 [13,15]. 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 [28,27]. 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.

Organization. Sect. 2 motivates DCC by an example. Sect. 3 outlines the main type constraints. Sect. 4 states the confinement guarantees. Sect. 5 discusses extensions and variations. The paper concludes with related work and future work. Appendix A reports implementation experiences.

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 character whose behavior is a function of the state of the hero to which it is associated. The intention is that a sidekick augments the power of its hero. The number of sidekicks that may be attached to a hero is a function of the hero's experience. A hero may adopt or orphan a sidekick at will. New hero and sidekick types may be introduced in future releases.

A possible design is to employ the Observer pattern [29] 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 `Sidekicks`. This set up would have worked had it not been the following requirement: *users may download new hero or sidekick types from the internet during a game play.* Because of dynamic extensibility, we must actively ensure fair game play by eliminating the possibility of cheating through the downloading of malicious characters. Two prototypical cheats are the following.

```
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)
         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 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 `Sidekicks`, or stealing existing instances from unprotected sources, and then attaching these instances to itself.

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 `Heros` may be accessed securely by `Sidekicks`. 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 sidekicks to the `Hero` object (i.e., capability leakage).

Solution Approach. To control capability propagation, DCC assigns the `Hero` and `Sidekick` interfaces to two distinct ***confinement domains*** [21], 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 `Heros` and `Sidekicks`, and as such it requires the rights to acquire instances of `Heros` and `Sidekicks`. This motivates the need to have a notion of ***trust*** to discriminate the two cases.

To further control the granting of capabilities, a ***capability granting policy*** [25] 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 leakage between `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 them, and then require that their subtypes also conform to the constraints. DCC achieves this via a mechanism known as ***hereditary mutual suspicion***.

3 Discretionary Capability Confinement

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. B ***grants*** a reference of type C to A when B invokes a method[1] declared in A, passing an argument via a formal parameter of type C.
2. B ***shares*** a reference of type C with 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 ***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

[1] 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.

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. We 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:

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

Acquiring non-capability references is harmless. Capability acquisition, however, is restricted by a number of constraints, the first of which is the following:

($\mathcal{DCC}2$) The following must hold:
1. A can generate a reference of type C only if $C \triangleright A$. [No capability generation is permitted.]
2. B may share a reference of type C with A only if $C \triangleright A \vee A \bowtie B$. [Capability sharing is not permitted across domain boundaries.]

In other words, capability acquisition only occurs as a result of explicit capability 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[2].

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.

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

[2] 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.

[3] 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 ($\mathcal{DCC}6$).

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

A method may be tricked into invoking another method that does not honor the caller's capability granting policy. This classical anomaly is known as the Confused Deputy [30]. The following constraint ensures that capability granting policies are always preserved along a call chain.

($\mathcal{DCC}4$) A method m may invoke another method n only if $n \rhd m$.

We now turn to the interaction between capability confinement and 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. The following constraint is imposed to ensure that widening does not turn a non-capability into a capability.

($\mathcal{DCC}5$) If $A <: B$ then $B \rhd A$.

When an instance method $B.n$ is invoked, the method that 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 intended for B rather than B'. The following constraint prevents impersonation.

($\mathcal{DCC}6$) Suppose $B.n$ is overridden by $B'.n'$. The following must hold:
1. $n' \rhd n$. [Overriding never relaxes capability granting rights.]
2. If the method return type is C, then $C \rhd B \vee B \bowtie B'$. [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 $C \rhd B' \vee B \bowtie B'$. [A method may be granted a capability only if it does not override a method declared in a different domain.]

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

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

($\mathcal{DCC}7$) results in a strong form of static separation of duty [27]. Firstly, as $\rhd$ is reflexive, no reference type can be a subtype of both A and B. This renders A and B mutually exclusive roles. Secondly, Sect. 4.2 shows that a class of collusion between A and B can be provably eliminated.

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 dominance hierarchy for the hero-sidekick game application is given in Fig. 2. 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 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., $l(\texttt{update}) = \texttt{SidekickDomain}$. Consequently, even if a `Sidekick` obtains a `Hero` reference, it is still not allowed to attach any sidekick to the `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.

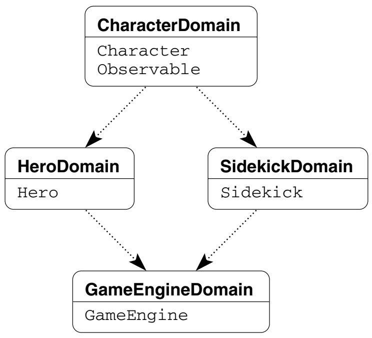

Fig. 2. Dominance hierarchy for the hero-sidekick application. Arrows represent "dominated-by" relationships (▶).

4 Confinement Properties

Given a discretionary access control mechanism such as DCC, safety analysis [31,32,33] must be conducted to characterize the conditions under which access rights are not granted to unintended parties. This section reports the confinement properties that have been established for DCC [34].

4.1 Featherweight JVM

Our confinement results are formalized in a lightweight model of the JVM called Featherweight JVM (FJVM) [35]. FJVM is a nondeterministic production system that describes how the JVM state evolves 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. FJVM manipulates *object references*. Every object reference is an instance of exactly one class. An object has an arbitrary number of fields, each of which is declared either

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

Fig. 3. FJVM states

in the class of the object or one of the supertypes. Each field in turn stores an object reference. A field may only be initialized once but never updated.

Fig. 3 summarizes the structure of a VM state $\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle$. The *object pool* Π is a finite set of *allocations* $r : C$, recording the objects allocated by the VM, together with their class membership. The *link graph* Γ is a finite set of *links*. A link $p : B \leadsto q : C$ asserts that p has a field declared in B, with field type C, storing the object reference q. The *stack frame* Φ is a finite set of *labeled references* $r : C$. The set Φ models the references accessible in a JVM stack frame, and tracks the type interfaces that are visible to the execution context. The *execution context* $A.m$ is the currently executing method. The *proper stack* σ models the call chain that leads to the current VM state. Specifically, σ is either an *empty stack*, $\diamond$, or a *non-empty stack*, $\mathit{push}(\Phi, A.m, C, \sigma)$, where Φ is the caller stack frame, $A.m$ is the caller execution context, C is the callee return type, and σ is another proper stack.

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.

Fig. 4 defines the state transition relation $\rightarrow_\Sigma$, which is parameterized by a *safety policy* Σ. Intuitively, Σ specifies for each execution context $A.m$ the set $\Sigma[A.m]$ of permitted events. The transition rules ensure that $\rightarrow_\Sigma$ observes Σ. We model the type rules of DCC by the policy in Fig. 5. ($\mathcal{DCC}5$) and ($\mathcal{DCC}7$) are not modeled: ($\mathcal{DCC}5$) is implicitly assumed in the proofs [34], and ($\mathcal{DCC}7$) is orthogonal to the confinement results.

4.2 Confinement Theorem

To help articulate confinement guarantees, a family of *Accessible* judgments are defined in Fig. 6 to assert that a labeled reference ($r : C$ or $q : C$) is accessible from a domain ($\mathcal{D}$) in a given VM state. The main confinement theorem is stated below (consult [34] for a detailed proof).

Theorem 1 (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:*

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

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

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

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

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

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

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

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

Fig. 4. FJVM transitions

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. $C \rhd m \wedge \mathcal{D} \blacktriangleright l(m)$ *(controlled capability propagation)*

The theorem above ensures that capability propagation honors the capability granting policy of a method. In the following, we describe how one may annotate methods with capability granting policies to preserve useful confinement properties. Specifically, a method $A.m$ is said to be ***safe*** iff $m \rhd A$. *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

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

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

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

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

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

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

Fig. 5. A safety policy for DCC

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

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

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

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

Fig. 6. Accessibility judgments

makes capabilities unforgeable. Theorem 1 entails that executing safe methods always guarantees the absence of capability theft.

Corollary 2 (No Theft). *Suppose* $m \rhd A$ *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* $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma'\rangle)$, *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)*

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

Corollary 3 (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* $\mathit{Accessible}_{[\mathcal{D}]}(r : C \mid \langle \Pi', \Gamma'; \Phi', A'.m', \sigma' \rangle)$*, 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)*

Mutual Suspicion. Suppose A and B are mutually suspicious, and a safe method $A.m$ is invoked. By Corollary 2, no reference of type B will be acquired by A as a result of the invocation. Similarly, by Corollary 3, 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. A solution is to allow the aliasing of carriers across domain boundaries so long as the acquisition of capability carriers is categorically denied. This can be enforced easily by minor revisions to the type constraints in Sect. 3.2: (a) assume $C \bowtie C[\,]$; (b) adapt ($\mathcal{DCC}3$) to forbid the granting of capability carriers across domain boundaries. Further details concerning the treatment of arrays can be found in [34].

Genericity. Genericity does not present any security challenge to the present design of DCC. 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 implications to this set up: (1) since generic containers such as `Set` belong to the root domain, DCC permits the acquisition and transmission of capability containers; (2) 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 any additional type constraint to account for genericity.

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 these two issues by examining a reformulation of ($\mathcal{DCC}7$) that facilitates ***modular enforcement***. A conservative solution is adopted. Specifically, we want to check 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:

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

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 ($\mathcal{DCC}7$) in a form that facilitates modular enforcement:

($\mathcal{DCC}7''$) If $A <: B$, then $B :\triangleright A$.

The companion technical report [34] shows that ($\mathcal{DCC}7$) follows from ($\mathcal{DCC}7''$). That is, ($\mathcal{DCC}7''$) is sound but incomplete: 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.

6 Concluding Remarks

Related Work. Previous language-based capability systems [16,13,14] lack confinement guarantees. This work combines the idea of confinement domains [21] with capability granting policies [25] to achieve confinement.

The design of DCC has been influenced by confined types [21,22,23,24]. While the confinement boundaries of confined types are uniform, those in DCC are 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 instances of a given *concrete class*, but DCC is designed to selectively confine those references that would otherwise escape with a privileged *static type*.

The static type system pop [36] 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.

Stack inspection [5] is an access control model for program execution that involves code units belonging to distinct protection domains. A common assumption behind most existing models of stack inspection [5,6,7,9] is that the binding of permissions to code units is performed statically. DCC identifies protection domains with confinement domains. While the binding of code units to their protection domains is performed statically, the granting of permissions to protection domains occurs dynamically through capability acquisition. Notice, however, *the right to grant capabilities* is still modeled statically in DCC in the form of a stack invariant.

Although this work is primarily concerned with access control, and thus orthogonal to language-based information flow control [37], 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 is foundational in ensuring system integrity [26]. Establishing mutually-exclusive roles is a popular means [28] for implementing separation of duty. The underlying assumption is that collusion between multiple agents is *unlikely*. In DCC, hereditary mutual suspicion not only establishes mutually exclusive roles, but *provably* prevents a class of collusion. 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.

Future Work. To ease exposition, a simple representation of capability granting policy has been adopted. A future direction is to explore finer-grained representations of capability granting policies, and study the collaboration idioms thus enabled. First ideas are reported in [34].

The right to grant capability is always diminishing along a call chain. This restricts the reusability of methods, and causes methods deep in a call chain incapable of granting capabilities. Can we allow the amplification of capability granting right while preserving confinement? A helpful observation is that the reasoning of capability granting right is akin to stack inspection. Exploring this connection belongs to future work.

A limitation of DCC is the lack of support for capability revocation. It is obviously impossible to "revoke" a reference that has already been acquired by a confinement domain. However, the lack of revocation can be alleviated by carefully regulating authority delegation. Constrained delegation is a well-studied topic in role-based access control and trust management (see, particularly, [38,39,40]). Controlling delegation in DCC belongs to future work.

Acknowledgments. This work is 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** (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** (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** (2005) 344–382
10. Dennis, J.B., van Horn, E.C.: Programming semantics for multiprogrammed computations. Communications of the ACM **9** (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. Rees, J.A.: A security kernel based on the lambda-calculus. A. I. Memo 1564, MIT (1996)
13. 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 (SOSP'97), Saint Malo, France (1997) 116–128
14. 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)

15. 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
16. Jones, A.K., Liskov, B.H.: A language extension for expressing constraints on data access. Communications of the ACM **21** (1978) 358–367
17. 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
18. 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
19. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. 3rd edn. Addison Wesley (2000)
20. ECMA: Standard ECMA-335: Common Language Infrastructure (CLI). 2nd edn. (2002)
21. Vitek, J., Bokowski, B.: Confined types in Java. Software - Practice & Experience **31** (2001) 507–532
22. 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
23. 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
24. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. Journal of Functional Programming **16** (2006) 83–128
25. 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
26. 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
27. 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
28. 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** (2001) 224–274
29. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley (1994)
30. Hardy, N.: The confused deputy: or why capabilities might have been invented. Operating Systems Review **22** (1988) 36–38
31. Lipton, R.J., Snyder, L.: A linear time algorithm for deciding subject security. Journal of the ACM **24** (1977) 455–464
32. Sandhu, R.S.: The schematic protection model: Its definition and analysis for acyclic attenuating schemes. Journal of the ACM **35** (1988) 404–432
33. Sandhu, R.S.: The typed access matrix model. In: Proceedings of the 1992 IEEE Symposium on Security and Privacy. (1992) 122–136
34. Fong, P.W.L.: Discretionary capability confinement. Technical Report CS-2006-03, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada (2006)

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. Skalka, C., Smith, S.: Static use-based object confinement. International Journal of Information Security **4** (2005) 87–104
37. Sabelfeld, A., Meyers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21** (2003) 5–19
38. Bandmann, O., Dam, M., Firozabadi, B.S.: Constrained delegation. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA (2002) 131–140
39. Li, N., Grosof, B.N., Feigenbaum, J.: Delegation logic: A logic-based approach to distributed authorization. ACM Transactions on Information and System Security **6** (2003) 128–171
40. 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

A Implementation Experience

A.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. 7 illustrates how the domain hierarchy in Fig. 2 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 dominance 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 dominance 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 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 )
   public void update(Observable hero);
}
```

```
@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. 7. An interface hierarchy representing the dominance hierarchy of the hero-sidekick game application

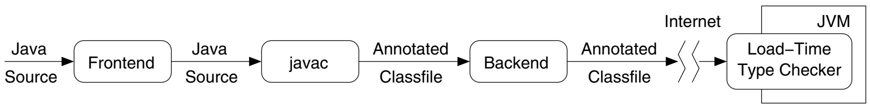

Fig. 8. The DCC software development environment

A.2 Type Checkers

We envision a programming environment (Fig. 8) 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. 8 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 DCC 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.

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.