Discretionary Object Confinement: A Minimalist Approach to Capabilities for the JVM

Philip W. L. Fong Boting Yang

Technical Report CS-2004-13
December 23, 2004

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

ISBN 0-7731-0506-9
ISSN 0828-3494

Discretionary Object Confinement: A Minimalist Approach to Capabilities for the JVM

Philip W. L. Fong Boting Yang
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2
{pwlfong,boting}@cs.uregina.ca

First Draft: December 23, 2004
First Revision: January 24, 2005
Second Revision: February 11, 2005

Abstract

Secure cooperation is the problem of protecting mutually suspicious code units from one another. The notion of capabilities is an effective means for facilitating secure cooperation in dynamically extensible software systems, in which both trusted and untrusted code may run alongside each other. This paper proposes a lightweight, statically enforceable type system, Discretionary Object Confinement (DOC), for modeling capabilities with abstract interface types. The type system can be seen as a discretionary variant of confined types. Formulated at the bytecode level, the type system is enforceable at link time, by the code consumer. Type checking does not involve any iterative flow analysis, and is therefore highly efficient. A link-time type checker has been implemented for the Java platform under the framework of Pluggable Verification Modules. The simplicity of the type system imposes only a modest increase in size to the trusted computing base. Although DOC enjoys an efficient type checking procedure, the inference of DOC annotations from legacy code base is NP-complete. The practical implication of this negative result is discussed.

1 Introduction

Secure cooperation [27, 24] is the problem of protecting mutually suspicious code units within the same execution environment from one another. Peer code units collaborate by sharing object references. The challenge is to allow the owner of an object reference to impose access constraints over those object references that are shared with an untrusted peer. Secure cooperation is thus an enabling infrastructure for dynamically extensible software systems such as mobile code language environments, scriptable applications, and software systems with plug-in architectures, in which both trusted and untrusted code units may run alongside each other.

The notion of capabilities [7, 6] provides an effective means for supporting secure cooperation. A capability is traditionally understood as an object reference plus a set of access rights that can be exercised through the reference. One way of implementing capabilities in an object-oriented programming environment is to employ a combination of the proxy design pattern [15] and load-time binary rewriting [19]. Such a solution approach results in proliferation of small objects, non-trivial performance overhead, and confusion of object identity. A more elegant approach is to embed the notion of capabilities into a statically checkable type system. In a *capability type system* [3], every object reference is statically assigned a capability type, which prescribes what operations can be performed through the object reference. A capability type may impose on the object reference a set of operational restrictions that constrains the way the underlying object may be accessed.

In this paper, we explore the following question: *What is the least perturbation to the Java programming language that allows us to build a rudimentary but useful capability type system?* A naive approach would be to exploit abstract interface types (e.g., abstract classes and interfaces in Java) as capability types. An abstract interface exposes only a

limited subset of the functionalities provided by the underlying object, and thus an object reference with an abstract interface 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 a capability interface. The receiver of the reference may then access the underlying object through the constrained interface. This naive scheme, however, has two problems:

1. **Capability Amplification.** The possibility of unrestricted downcasting allows a malicious receiver to amplify access rights. All the functionalities of the underlying object can be fully exposed if its concrete class identity is known to the a malicious peer.

2. **Capability Leaking.** A peer may freely acquire capability instances (e.g., through instantiation). This is problematic when capabilities are supposed to be owned and managed by an abstraction, through which clients obtain legitimate capability instances.

A careful examination of the research problem reveals its deeper tie with the recent line of work on rethinking the notion of encapsulation in object-oriented programming languages [23, 5, 29, 17, 4, 2, 1, 26, 25]. Influenced by the work of Vitek *et al* [29, 17, 30], this paper proposes a lightweight, statically enforceable type system called Discretionary Object Confinement (DOC) that fully supports the use of abstract interface types as capability types. The type system can be seen as a discretionary variant of confined types [29, 17, 30], in which the boundaries of confinement domains are semi-permeable. Specifically, the space of Java reference types is stratified into a hierarchy of confinement domains, partially ordered by a binary trust relation. Java reference types declared inside a confinement domain are seen from the outside as capability types. So long as a capability remains inside its defining confinement domain, it has the same semantics as a regular Java reference. However, once a capability escapes to an untrusted domain, its concrete class identity will be concealed. From then on, it can only be accessed through a non-bypassable abstract interface, thereby avoiding capability amplification. To control capability propagation, an untrusted domain is not allowed to acquire a capability unless it is explicitly granted one by argument passing. Capability transfer is thus moderated through a combination of mandatory and discretionary control. The DOC type system supports a useful form of secure cooperation in the presence of dynamically loaded software extensions. In a related work [14], it is also shown that DOC can be used as the foundation for building more sophisticated capability type systems in the Java platform.

The contributions of this paper are the following:

1. A lightweight, statically enforceable type system, DOC, was proposed to support the use of abstract interface types to model capabilities. We demonstrate the utility of the DOC type system in facilitating a form of secure cooperation in the context of dynamically extensible software systems.

2. The DOC type system is formulated at the bytecode level, and is thus enforceable at link time by the code consumer. This feature is crucial in the context of dynamically extensible software systems, in which untrusted extensions may be produced by a malicious compiler.

3. The DOC type system is designed to be extremely lightweight, so that type checking involves only a linear-time scan of the target classfile: no iterative flow analysis is involved. This lightweight design is desirable because it incurs only a modest increase to the size of the trusted computing base.

4. The link-time type checking procedure has been implemented in an open source JVM, the Aegis VM [8], which provides an extensible protection mechanism, Pluggable Verification Modules [10], for the safe introduction of third-party verification services into the dynamic linking process of Java.

5. The problem of inferring DOC annotations from legacy code has been characterized to be NP-complete. Preliminary experience in employing heuristic approaches to solve the DOC type inference problem suggests that the current formulation of the problem may be under-constrained. We believe that a more constrained formulation is in order.

The rest of this paper is organized as follows. Sect. 2 gives an overview of DOC, and demonstrates its utility. Sect. 3 presents a bytecode-level formulation of DOC. Sect. 4 reports an implementation of a link-time type checker for DOC. Sect. 5 explores the problem of inferring DOC annotations from legacy code. The paper concludes with discussions, related work, future work, and a summary.

```
public abstract class Character { /* Common character behavior */ }

public interface Observable {
  Context getContext();       // e.g., forest, river, bad land
  Mode getMode();             // e.g., combating, regenerating, scouting
  Activity getActivity();     // e.g., defend, attack, move towards
  Character getTarget();      // e.g., target of activity
}

public abstract class Hero extends Character implements 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 (int i = 0; i < observers.length; i++)
      if (observers[i] != null)
        observers.update(this);
  }
}

public abstract class Sidekick extends Character {
  public abstract void update(Observable hero);
}

public class GameEngine {
  // Manage life cycle of characters
}
```

Figure 1: Regular set up of hero-sidekick game.

2 Discretionary Object Confinement

2.1 A Motivating Example: Observer Pattern

2.1.1 Setting the Stage.

Suppose we are developing a role-playing game. Over time, a hero (e.g., Bat Man) may acquire an arbitrary number of *sidekicks* (e.g., Robin). A sidekick is an AI-controlled character whose behavior is a function of the state and behavior of the hero to which it is associated. 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 its type and experience. A hero may also choose to adopt or orphan a sidekick at any point of time. New sidekick and/or hero types may be introduced in future releases of the game.

A standard set up of the game is to employ the Observer pattern [15] 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.

2.1.2 Complications.

The set up in Fig. 1 would have worked beautifully 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.

Capability Amplification: Upon receiving a `Hero` object through the `Observable` argument of the `update` method, a malicious `Sidekick` may then downcast the `Observable` reference to a `Hero` reference, thereby exposing the *sidekick management interface* of the `Hero` object. This allows the malicious `Sidekick` to attach powerful sidekicks to the `Hero` object, thereby turning the `Hero` object into a more potent character.

Capability Leaking: A malicious `Hero` can augment its own power by simply creating new instances of concrete `Sidekick` subclasses and attaching these instances to itself.

A number of lessons may be learned from the above archetypical cheats.

Capabilities. Notice that a `Hero` exports two 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 by the `GameEngine`, the latter is a limited interface through which `Sidekick`s interact with `Hero`s. In short, an `Observable` reference to a `Hero` object is supposed to be a *capability*: i.e., a statically typed reference with a concealed class identity. Unfortunately, standard Java provides no provision for protecting capabilities. Access rights can be readily amplified by downcasting, as we have seen in the first cheat.

Confinement. To the `Hero`s, `Sidekick`s are also capabilities: the `Sidekick` interface exposes the `update` method, the invocation of which augments the behavior of a `Hero`. The second cheat exploits the weakness of standard Java semantics, whereby `Hero`s may freely acquire `Sidekick` capabilities through object instantiation. What is needed is a notion of *confinement domain*, so that leaking of capabilities can be controlled.

Trust. It is intended that `Hero` and `Sidekick` belong to two distinct confinement domains. Amplification and leaking of capabilities should therefore be restricted. In contrast, `GameEngine` is by design responsible for managing the life cycle of `Hero`s and `Sidekick`s, and as such it requires full access to everything belonging to the two confinement domains. That is, restrictions on amplification and leaking of capabilities should not apply to `GameEngine`. This highlights the need to have a discriminating confinement domain boundary, so that full access may be given to trusted confinement domains.

Capability Granting by Discretion. There are times in which capability granting should be allowed even between mutually distrusting confinement domains. For example, suppose `Observable` belongs to the same confinement domain as `Hero`. Then `Observable` is a capability type from the perspective of `Sidekick`, and crossing of confinement boundary is not supposed to be allowed. Broadcasting of change would then be impossible. Therefore, a discretionary means for capability granting via argument passing should be allowed.

2.2 Enter Discretionary Object Confinement

2.2.1 Basic Concepts.

Discretionary Object Confinement formalizes the features suggested in the analysis above. Specifically, DOC is based on the following concepts:

1. The space of Java reference types is partitioned into a number of *confinement domains*, so that every Java reference type belongs to exactly one confinement domain.

2. If Java reference types A and B both belong to the same confinement domain, then A may freely *acquire* a reference of type B. In this case, B is said to *trust* A. Formally, reference acquisition occurs when one of the following happens:

   - An exception handler in A with catch type B catches an exception.

- A creates an instance of B.
- A casts a reference to type B.
- A invokes a method with return type B.
- An object reference is passed to a method declared in A via a formal parameter of type B.
- A reads a field with field type B.
- An object reference is stored into a field of A with field type B.

3. The confinement domains are organized into a partially ordered *subsumption hierarchy*. If one confinement domain subsumes another, then reference types belonging to the subsumed domain trust those belonging to the subsuming domain. This means that reference type A may freely acquire a reference of type B if the confinement domain to which B belongs is subsumed by that of A.

4. If C does not trust A, then a reference of type C is said to be a *capability* for A. A Java reference type A may acquire a capability of type C only through the following means:

   - *Capability granting:* A Java reference type B invokes a method declared in A, passing an argument via a formal parameter of type C.
   - *Capability sharing:* A Java reference type B belonging to the same confinement domain as A shares a capability of type C with A when one of the following happens:
     - B invokes a method declared in A, passing an argument via a formal parameter of type C.
     - A invokes a method declared in B with return type C.
     - A reads a field declared in B with field type C.
     - B stores a reference into a field declared in A with field type C.

   In no other ways shall A acquires a capability.

5. Capabilities provide the only means for untrusted types to access methods declared in the capability type. Specifically, if C is a capability for A (i.e., C does not trust A), then A shall not invoke static methods declared in C.

In summary, the following invariant is maintained:

Capability Confinement. *A class must not acquire a capability unless it is explicitly granted one via argument passing, or it acquires the capability through sharing with another class belonging to the same confinement domain.*

When a capability is granted, it roams freely within the confinement domain. However, 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.

2.2.2 Addressing the Security Challenges.

The security challenges of capability amplification and leaking can be fully addressed by DOC. Specifically, the confinement domains and subsumption hierarchy as shown in Fig. 2 can be defined for the hero-sidekick game application. Because **HeroDomain** and **SidekickDomain** are incomparable in the subsumption hierarchy, `Hero` and `Sidekick` are capabilities of each other. Consequently, not only are `Sidekick`s not allowed to amplify an `Observable` reference to a `Hero` capability (i.e., capability amplification prevented), `Hero`s are also forbidden to create new `Sidekick` capabilities (capability leaking prevented). Furthermore, the subsumption hierarchy also renders **GameEngineDomain** the most subsuming confinement domain, thereby allowing `GameEngine` to have full access to the reference types belonging to the rest of the confinement domains. Lastly, the provision for capability granting makes it legitimate to pass `Observable` references from `Hero` to `Sidekick` when an update is broadcasted.

Notice that we have eschewed giving concrete syntactic devices for specifying domain membership and subsumption relationship. A diagram such as Fig. 2 is considered sufficient for now. More details concerning the embedding of such information into Java classfiles will be given in Sect. 4.1.

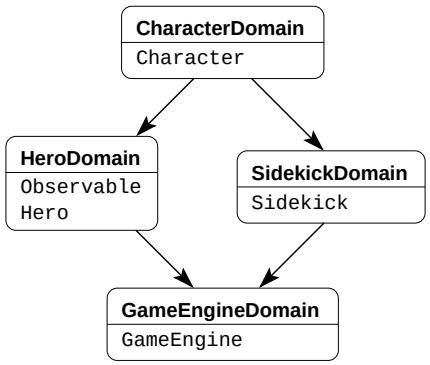


Figure 2: Subsumption hierarchy for the hero-sidekick application. Arrows represent "subsumed-by" relationships, meaning that unconstrained reference acquisition may occur along the direction of the arrows.

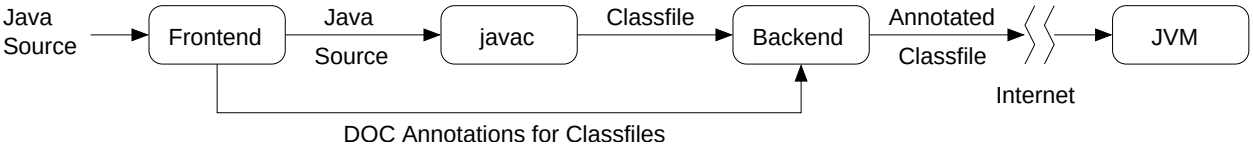


Figure 3: Processing of DOC Annotations

3 The DOC Type System for Java Bytecode

In the context of the JVM, in which 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 Discretionary Object Confinement is to be used for enabling secure cooperation, it must be enforceable at the bytecode level, by the code consumer. This section provides a systematic presentation of the bytecode-level typing discipline for DOC. The section begins with the fixing of assumptions and notations (Sect. 3.1). The notions of confinement domains, subsumption hierarchy, trust relation, and capabilities are then introduced formally (Sect. 3.2). The bulk of the section outlines the type constraints for DOC (Sect. 3.3–3.4).

3.1 Assumptions and Notations

To fix thoughts, notice that *three* type systems are involved in the present discussion: (1) the standard Java type system, (2) the source-level DOC annotation scheme as described in the previous section, and (3) the bytecode-level DOC type system to be presented in this section.

Firstly, there is the standard Java type system. We use A, B and C to denote a Java reference type, which is a class, interface, or array type in the standard Java type system. Methods and fields are denoted by m and f respectively. A constant pool reference[1] is denoted by its referent's signature delimited by angled brackets ($\langle\cdot\rangle$). For example, $\langle B\rangle$ denotes a constant pool class reference that resolves to the Java reference type B. Along a similar vein, $\langle B.f : C\rangle$ denotes a constant pool field reference that resolves to a field f declared in B with field type C, while $\langle B.m : C\rangle$ denotes a constant pool method reference that resolves to a method m declared in B with method return type C.

Secondly, there is the source-level DOC annotation scheme. We assume that Java source files are properly annotated so that the confinement domain to which a class belongs is indicated by some backward-compatible syntactic devices such as `javadoc` taglets/doclets or JSR 175 metadata facility.

Thirdly, there is the bytecode-level DOC type system. We envision a programming environment (Fig. 3) in which domain membership and subsumption relationship information embedded in Java source files is extracted by a compiler frontend, translated into classfile annotations, and subsequently injected into the classfiles generated by the

[1]The symbol table of a classfile is called a constant pool [22].

standard `javac` compiler. Type checking will be conducted by the JVM at link time, against classfiles, at the bytecode level. It is the detailed typing discipline of this bytecode-level DOC type system that is presented in this section.

3.2 Confinement Domains, Subsumption, Trust, and Capabilities

Confinement Domains. Every Java reference type belongs to exactly one *confinement domain*. We use $\mathcal{D}$ and $\mathcal{E}$ to denote confinement domains, and write $C \in \mathcal{D}$ when Java reference type C belongs to confinement domain $\mathcal{D}$.

Subsumption Hierarchy. A binary *subsumption relation* $\blacktriangleright$ is defined over the class of confinement domains. The relation $\blacktriangleright$ is a partial ordering, and as such it is reflexive, transitive, and antisymmetric. We say that $\mathcal{D}$ *is subsumed by* $\mathcal{E}$ whenever $\mathcal{D} \blacktriangleright \mathcal{E}$. Intuitively, instances of Java reference types belonging to $\mathcal{D}$ may be freely acquired by a class belonging to $\mathcal{E}$. 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 classes from all confinement domains[2].

Trust Relation. The subsumption hierarchy induces a natural ordering of Java reference types. Specifically, if $B \in \mathcal{D}$, $A \in \mathcal{E}$ and $\mathcal{D} \blacktriangleright \mathcal{E}$ then we write $B \triangleright A$, and say that B *trusts* A, meaning that A may freely acquire instances of B. By definition $\triangleright$ is reflexive and transitive (but not antisymmetric). We thus write $A \bowtie B$ if both $A \triangleright B$ and $B \triangleright A$.

Capabilities. An object reference with type C is said to be a *capability* for A if $C \triangleright A$ does not hold. Capability acquisition must take the form of capability granting or capability sharing. A and B may share capabilities if $A \bowtie B$.

3.3 Type Constraints

To enforce the Capability Confinement Invariant, the following constraints are imposed on classfiles.

C1 If A extends or implements B then $B \triangleright A$ must hold.

> ***Commentary.*** This rule forces the trust relation to be consistent with the Java type hierarchy. As a result, Java type widening never amplifies capability. This design renders iterative flow analysis unnecessary at link time, thereby allowing us to obtain a very small increase in the size of trusted computing base in the implementation of a link-time type checker (Sect. 4.2).

C2 If a bytecode method declared in A provides an exception handler for exception type $\langle B \rangle$, then $B \triangleright A$ must hold[3].

> ***Commentary.*** Exception handlers may not intercept capabilities.

C3 If a bytecode method declared in A contains the bytecode instruction "**new** $\langle B \rangle$", then $B \triangleright A$ must hold.

> ***Commentary.*** Capability instantiation is not permitted.

C4 If a bytecode method declared in A contains the bytecode instruction "**checkcast** $\langle B \rangle$", then $B \triangleright A$ must hold.

> ***Commentary.*** Capability amplification is not allowed.

C5 If a bytecode method declared in A contains the bytecode instruction "**invokestatic** $\langle B.m : C \rangle$", then $B \triangleright A$ must hold.

> ***Commentary.*** Untrusted types may only access methods declared in a capability type via a capability instance.

[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 subsumption hierarchy therefore reflects this concern: Java platform classes and application core classes belong respectively to the least and the most subsuming domain.

[3]The `catch_type` field [22, Sect. 4.7.3] of a `Code` attribute specifies the target exception type of an exception handler in the corresponding bytecode method. If the field assumes a zero value, then the handler is a "*catch-all*" handler. The corresponding exception type is therefore `java.lang.Throwable`, which belongs to the root confinement domain. In this case, the subsumption requirement trivially holds.

C6 If a bytecode method declared in A contains the bytecode instruction "**invoke*** $\langle B.m:C\rangle$, where **invoke*** is one of **invokeinterface**, **invokespecial**, **invokestatic** or **invokevirtual**, then either $C \triangleright A$ or $A \bowtie B$ must hold.

Commentary. Reference acquisition through method return is permitted only if the the acquired reference is not a capability, or if the acquisition is achieved through capability sharing. Notice that no data flow restriction is placed on argument passing: capability granting is enabled.

C7 If a bytecode method declared in A contains the bytecode instruction "**get*** $\langle B.f:C\rangle$", where **get*** is one of **getfield** or **getstatic**, then either $C \triangleright A$ or $A \bowtie B$ must hold.

Commentary. Reference acquisition through field reading is permitted if the acquired reference is not a capability, or if the acquisition is achieved through capability sharing.

C8 If a bytecode method declared in A contains the bytecode instruction "**put*** $\langle B.f:C\rangle$", where **put*** is one of **putfield** or **putstatic**, then either $C \triangleright B$ or $A \bowtie B$ must hold.

Commentary. Reference acquisition is permitted through field writing if the acquired reference is not a capability, or if the acquisition is achieved through capability sharing.

Notice from the above list that all means of capability acquisition are rejected except for capability granting and capability sharing. The Capability Confinement Invariant is therefore enforced.

3.4 Handling Array Types

The array type $C[\,]$ is considered to be a carrier of Java reference type C, and as such it is considered to be as capable as C. Therefore, the following relation is imposed to avoid leaking of C via a $C[\,]$ carrier:

$$C \bowtie C[\,]$$

With this, the type constraints in the previous section will work with array types also. Although propagation of capability carriers is carefully moderated, creation of capability carriers does not amplify access rights. Consequently, empty arrays can be freely instantiated using bytecode instructions **anewarray** or **multianewarray** without violating the Capability Confinement Invariant. By a similar token, retrieval of a capability from a carrier or storage of a capability into a carrier should be freely allowed once the carrier is acquired (this is just a form of capability sharing). Therefore, there is no need to impose further type constraint on bytecode instructions **aaload** and **aastore**. An implementation, however, may (out of paranoia) choose to treat instantiation of capability carriers in the same way as capability instantiation, thereby obtaining a slightly more stringent set of type rules. The implementation efforts described in this paper assume the more liberal posture.

4 Type Checking Procedure

This section reports the implementation strategy we have employed to realize the bytecode-level DOC type system in a JVM. Two subtasks are involved. Firstly, one has to decide how confinement domains, the subsumption hierarchy, and domain membership are embedded in Java classfiles and represented at run time inside a JVM. Secondly, one has to decide how to incorporate a link-time type checking procedure into the dynamic linking process of the JVM so as to enforce DOC type constraints. The two topics are covered in Sect. 4.1 and 4.2 respectively.

4.1 Type Annotations

4.1.1 Representation.

It is desirable not to introduce additional run-time data structure for tracking domain membership and subsumption relationship in the JVM. Confinement domains are therefore represented as subinterfaces of the interface

```
public interface CharacterDomain implements org.aegisvm.doc.Root { }
public interface HeroDomain implements CharacterDomain { }
public interface SidekickDomain implements CharacterDomain { }
public interface GameEngineDomain implements HeroDomain, SidekickDomain { }
```

Figure 4: Interface hierarchy representing the subsumption hierarchy of the hero-sidekick game application.

`org.aegisvm.doc.Root`, which in turn represents the root domain $\top$. Subsumption is therefore naturally induced by subinterfacing, so that $\mathcal{E}$ being a subinterface of $\mathcal{D}$ signifies $\mathcal{D} \triangleright \mathcal{E}$. For example, the subsumption hierarchy depicted in Fig. 2 can be codified in the interface hierarchy shown in Fig. 4. The acyclicity of subinterfacing automatically guarantees that subsumption represented in this way is antisymmetric.

We use a class attribute [22, Sect. 4.7] with attribute name `DOC` to declare the confinement domain of a given class. Specifically, a classfile may be annotated in three ways:

1. A classfile with no `DOC` attribute defines a reference type that belongs to the root domain $\top$. Such a reference type is not a confinement domain.
2. An empty `DOC` attribute identifies a reference type to be a confinement domain. Such a reference type belongs to the root domain $\top$.
3. A non-empty `DOC` attribute defines a reference type that does not correspond to a confinement domain. The `DOC` attribute holds a valid index into the `interfaces` array of the classfile structure. The referenced interface identifies the confinement domain of the reference type[4].

To ensure that classfile annotations are well-formed, an additional constraint is introduced:

C0 The following must hold:

1. The classfile corresponding to the root domain must be explicitly identified as a confinement domain (i.e., by an empty `DOC` attribute).
2. The declared confinement domain of a classfile must indeed be tagged as a confinement domain.
3. A confinement domain must be a public interface that declares no field or method.
4. The direct superinterfaces of a confinement domain must also be confinement domains.
5. A confinement domain has no superinterface iff it is the root domain.
6. If a classfile contains a non-empty `DOC` attribute, then except for the direct superinterface identified in the `DOC` attribute, none of the direct superinterfaces shall be a confinement domain.

4.1.2 Annotation Tool.

A simple Linux command-line tool has been developed to take the role of the backend component in Fig. 3, providing a convenient means for manual annotation of classfiles.

4.2 Link-Time Type Checking

4.2.1 Pluggable Verification Modules.

A type checking procedure that enforces the constraints specified in Sect. 3.3 must be incorporated into the dynamic linking process of the JVM in order to enforce the DOC type system at the bytecode level. The Aegis VM [8] is an open source JVM that offers an extensible protection mechanism called Pluggable Verification Modules (PVMs)

[4]This design has two benefits: Firstly, the standard bytecode verification procedure will make sure that the declared confinement domain of a class is indeed an interface. Secondly, the standard dynamic linking semantics guarantees that the confinement domain interface is accessible from within the class when it is prepared. This property is crucial in guaranteeing proof obligations can be discharged properly (see Sect. 4.2.2).

Type System	TCB Size Increase (LOC)	% wrt DOC
DOC	*788*	*100%*
JAC	2647	336%
Confined Types	2926	371%

Figure 5: TCB size increase resulting from bytecode-level implementation of various type systems.

[10, 11]. Unlike other implementations of the JVM, in which the link-time bytecode verification service is a fixture that cannot be extended conveniently, the PVM framework is based on a modular verification architecture whereby bytecode verification is a pluggable service that can be readily replaced, reconfigured and augmented. Third-party verification services can be safely incorporated into the dynamic linking process as a PVM. The verification service exported by a PVM will be invoked prior to the preparation of a classfile [22, Sect. 5.4.2]. The PVM may also schedule programmer-defined checks, called *proof obligations*, to occur at various points of the dynamic linking process. The PVM framework also provides reusable facilities for easing the construction of link-time static program analyzers.

4.2.2 Proof Obligations.

A DOC PVM was implemented to enforce the type constraints specified in Sect. 3.3. When the verification service exported by the DOC PVM is applied to an incoming classfile for Java reference type A, it performs the following tasks:

1. It parses any `DOC` attribute that may be attached to the class, and verifies that the attribute is well-formed.
2. It scans through the class interface and also the body of each bytecode method once in order to formulate proof obligations corresponding to type constraints **C0**–**C8**. Specifically, two types of proof obligations are formulated:
   (a) A proof obligation that encapsulates type constraints **C0**–**C1** is scheduled to be discharged when a class is prepared [22, Sect. 5.4.2].
   (b) For each exception handler, a proof obligation encapsulating type constraint **C2** is scheduled to be discharged when a class is prepared.
   (c) For each of the constraints **C3**–**C8**, a proof obligation is scheduled to be discharged when the corresponding constant pool reference is resolved [22, Sect. 5.4.3]. For example, in the case of **C7**, if the bytecode instruction "**getstatic** $\langle B.f:C\rangle$" is found to be in the body of a bytecode method, then the proof obligation $(C \triangleright A \vee A \bowtie B)$ will be scheduled for discharging at the point when the constant pool field reference $\langle B.f:C\rangle$ is resolved in A.

It is easy to see that this scanning process takes time proportional to the size of the classfile. Notice also the discharging of proof obligations incur a performance overhead comparable to those of the checks enforcing standard Java access control (e.g., `public`, `protected`, etc).

4.2.3 Trusted Computing Base.

The resulting implementation is extremely compact, consisting of only 788 lines of moderately commented C code. The simplicity of DOC's design results in a type checking procedure that does not involve an iterative static analysis component. Consequently, the increase in the size of *trusted computing base (TCB)* is quite acceptable. This compares favorably with the increase of TCB size resulting from previously published [10, 9] bytecode-level implementation of JAC [21] and confined types [29, 17, 30] (See Fig. 5).

5 Type Inference

Suppose one is given a legacy Java application (e.g., a `jar` file) that does not come with DOC annotations. Is there a way to automate the annotation of the legacy code base? More interestingly, is it possible to design a development environment in which the programmer obtains automatically generated advice on how her code should be annotated? Such are the possible applications of the DOC type inference problem. In general, there are multiple ways to organize the confinement domain hierarchy for a given code base. One desires a hierarchy that satisfies certain security-related criteria. This section reports preliminary exploration of DOC type inference with respect to a specific criterion. It turns out that, although DOC type checking is highly efficient, the formulation of DOC type inference as explored here is computationally intractable (Sect. 5.1). Preliminary experiences in employing heuristic approaches to solve the problem are discussed (Sect. 5.2).

5.1 A Formulation of the Type Inference Problem

The goal of DOC type inference is to infer from a legacy code base a set of confinement domains and to identify their member classes. Notice that there is always a degenerate solution: all reference types are assigned to the root domain. Intuitively, a finer-grained allocation of classes to confinement domains is more desirable. Therefore, the number of inferred confinement domains should be *maximized*. The inference process consists of two stages:

1. Scan through the code base to obtain a set of type constraints mandated by **C1**–**C8**. The constraints have the form of either $(B \triangleright A)$ or $(C \triangleright A \vee A \bowtie B)$.

2. Find a solution to the constraints that maximizes the number of equivalence classes induced by the binary relation $\bowtie$. Recall that the binary relation $\triangleright$ over reference types is reflexive and transitive but not antisymmetric. The binary relation $\bowtie$ therefore defines an equivalence relation over reference types. The equivalence classes induced by $\bowtie$ correspond to the confinement domains we seek to infer.

The first stage is easy to deal with. The second stage, however, involves solving a combinatorial optimization problem.

Specifically, each reference type can be represented as a node in a digraph, and the relationship $B \triangleright A$ as an arc from node B to node A. Under this representation, constraint solving becomes equivalent to the construction of a digraph out of the arcs prescribed in the constraints. Along the same vein, an equivalence class induced by $\bowtie$ corresponds to a strongly connected component (SCC) in the digraph. Given a set of constraints, the optimization objective is therefore to maximize the number of SCCs in the constructed digraph.

A unary constraint of the form $B \triangleright A$ mandates that the arc (B, A) must go into the constructed digraph. So constraint solving may very well begin with an initial graph constructed out of the unary constraints. A disjunctive constraint of the form $(C \triangleright A \vee A \bowtie B)$ requires more care. In particular, $(C \triangleright A \vee A \bowtie B)$ can be rewritten as $(C \triangleright A \vee A \triangleright B) \wedge (C \triangleright A \vee B \triangleright A)$. Generalizing, let us assume that the disjunctive constraints are given as a list of primitive disjunctive constraints of the form $(A_1 \triangleright B_1 \vee A_2 \triangleright B_2)$. That is, a primitive disjunctive constraint requires that we place into the resulting digraph at least one arc from the pair $\langle (A_1, B_1), (A_2, B_2) \rangle$.

Seen in this light, stage two of type inference is equivalent to solving the following optimization problem:

MAX-SCC **Problem (Optimization Version):** Given a digraph G and a list L of pairs of arcs, at least one arc from each pair in L is to be selected and added to G. Which combination of arc selection will result in a graph with the maximum number of strongly connected components?

A decision version of the optimization problem can be formulated in the standard way. It turns out that the decision problem is NP-complete. The proof of this result can be found in Appendix A

Theorem 1 MAX-SCC *(decision version) is NP-complete.*

It is therefore unlikely that there is a good algorithm for the present formulation of the type inference problem. We thus set out to look for a good heuristic solutions.

5.2 Preliminary Experience

Since MAX-SCC is NP-complete, we do not expect good algorithms to exist for the constraint problem. We do, however, have some preliminary experience in employing a branch-and-bound algorithms to solve MAX-SCC instances. Based on the Byte Code Engineering Library[5] (BCEL), a DOC type inference tool has been developed. Given one or more `jar` packages, the tool construct a corresponding instance of MAX-SCC, and apply a branch-and-bound algorithm to search for an optimal subsumption hierarchy for the input packages. Some non-trivial graph-theoretic optimizations have allowed us to prune the search space significantly. A number of heuristic search strategies were developed to further improve performance. A combination of optimizations and heuristics[6] allowed us to fully infer the optimal subsumption hierarchies of medium-sized software packages such as JRuby[7] (468 classfiles, 832KB) and Jython[8] (336 classfiles, 720KB). Our implementation, however, failed to complete the inference task in a reasonable amount of time when presented with Kawa[9] (746 classfiles, 1.4MB). Although further experimentation would have allowed us to come up with increasingly competitive heuristics, two observations disturbed us:

1. *The optimal subsumption hierarchies contain too many confinement domains to be comprehensible for human programmers.* For example, the optimal subsumption hierarchy for JRuby contains 317 confinement domains (out of 468 classfiles), while the optimal subsumption hierarchy for Kawa features 469 confinement domains (out of 746 classfiles). Admittedly, there are exceptions in which the numbers are slightly more reasonable, (e.g., 189 confinement domains out of 336 classfiles for Jython), but they are exceptions rather than the norm.

2. *Better heuristics probably would not improve the tractability of constraint solving much.* One thing we tried was to use the same heuristic strategy to solve a constraint problem twice, and used the optimal solution obtained in the first round as an initial solution for the branch-and-bound algorithm in the second round. We would have expected that, given the aggressive initial bound, the second round of constraint solving would be significantly more efficient than the first round. To our surprise, no significant performance differential was observed. This means that the branch-and-bound algorithm was not able to perform much pruning until it got very deep into the search tree.

Both of the observations suggest that the current formulation of DOC type inference is under-constrained, resulting in the proliferation of confinement domains and the ineffectiveness of heuristic search strategies.

We still believe that maximization of the number of confinement domains is a security-wise sound criterion. What is missing from the formula is probably additional criteria such as natural-ness and comprehensibility. We conjecture that formalization of these criteria will result in a denser initial digraph G, which will in turn lead to more efficient constraint solving and subsumption hierarchies of more manageable complexities. Validation of these conjectures clearly belongs to future work.

6 Concluding Remarks

6.1 Discussion

DOC was originally discovered during the study of a more sophisticated capability type system [14], in which the data flow trajectory of an object reference can be constrained via a capability type with a structure resembling the sequential fragment of CSP [18]. DOC was superimposed on this capability type system to counter a class of capability spoofing attacks. It was later recognized that capability amplification and capability leaking are two general issues that capability type systems similar to [14] must wrestle with. The successful experience of constructing the capability type system in [14] on top of DOC leads us to believe that DOC could serve as a basic building block for more sophisticated capability type systems, providing the infrastructure for capability confinement.

[5] `http://jakarta.apache.org/bcel`.
[6] Details to be given in an up-coming technical report.
[7] `http://jruby.sourceforge.net`.
[8] `http://www.jython.org`.
[9] `http://www.gnu.org/software/kawa`.

6.2 Related Work

Confined Types. DOC can be considered a discretionary variant of confined types [29, 17, 30]. The first point of comparison is that, while the confinement boundaries of confined types are absolute and uniform, those in DOC are semi-permeable and discriminatory, allowing capability granting through discretion and reference acquisition through subsumption. A unique feature of our type system is that references may escape from a confinement domain so long as it assumes a certain non-bypassable abstract interface. While Vitek *et al* identify confinement domains with Java packages, thereby reusing the access control semantics of package private members, the subsumption hierarchy of DOC 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 DOC 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 an iterative flow analysis element [9]; link-time type checking of DOC does not.

Ownership Types. Ownership types [23, 5, 4, 2, 1] is a family of typing disciplines for preventing leaking of references to the internal representation of an aggregate object. It works at a granularity that is finer than confined types: whereas confined types prevent reference leaking at the package level, ownership types prevent leaking of component references outside of an aggregate object. A particular relevant formulation of ownership types is the recent proposal of ownership domains [1]. Our work shares with [1] the approach of using multiple encapsulation boundaries (aka domains) to specify non-uniform aliasing policies. [1] also allows fine-grained inter-domain alising constraints to be specified through the *link* directive. A similar role is filled in this work by the subsumption hierarchy. While our trust relation as induced by the subsumption hierarchy is much less flexible than the user-specified link relation in [1], it nevertheless represents a natural solution for a wide varieties of modeling problems.

Composable Encapsulation Policies. This work shares with Composable Encapsulation Policies (CEP) [26, 25] the concern of providing alternative interfaces for multiple client categories. The role of encapsulation policies is analogous to capabilities in DOC. Schärli *et al* [26] offer an insightful comparison between CEP and Java interfaces. Most of the analysis also applies to DOC, except for one. They [26] rightly observe that abstract interfaces could have been used for moduling encapsulation policies had it not been the fact that encapsulation policies modeled as such are not enforceable. Under DOC, encapsulation policies expressed in Java interfaces *can* indeed be enforced as capabilities. DOC can therefore be seen as the least intrusive augmentation to Java that turns interfaces into enforceable encapsulation policies. CEP, however, offers a more flexible language for specifying encapsulation policies than Java interfaces: provision of fine-grained access rights (i.e., called, overridden, reimplemented) and composition of encapsulation policies, etc. Whereas CEP offers per-client class customization of encapsulation policies, DOC customizes the capability of an object reference using confinement domains and subsumption relations. Whereas the implementation of CEP involves dynamic checks, DOC is a statically enforceable type system.

Pluggable Verification Modules. The PVM framework [10] of the Aegis VM [8] is based on a modular verification architecture called Proof Linking [11]. The correctness of Proof Linking, especially its interaction with lazy dynamic linking, has been studied rigorously [12]. The correctness proof has been generalized to account for multiple classloaders [13]. Both JAC [21] and confined types have been implemented under the PVM framework [10, 9]. As reported in Sect. 4.2, the simplicity of DOC results in an increase in size of TCB that is more competitive than a typical implementation of an alias control type system.

6.3 Future Work

We plan to extend this work along three directions. Firstly, we plan to establish the soundness of DOC in Featherweight Java [20]. Secondly, we plan to evaluate the utility of DOC in a medium-sized software development project involving the construction of extensible systems. Through this exercise, we plan to document any coding idioms and design patterns pertaining to the use of DOC. Thirdly, we plan to explore more constrained formulations of DOC type inference, and also the design of heuristics and/or approximation algorithms for these formulations. The goal is to

construct program development environments that can offer advice to developers regarding how DOC confinement domains should be organized.

6.4 Conclusion

A lightweight, statically enforceable type system, Discretionary Object Confinement, was proposed. Its utility in facilitating a useful form of secure cooperation was explored in the context of extensible systems. Formulated at the bytecode level, the type system is enforceable at link time by the code consumer. Type checking is highly efficient, involving only a linear-time scan of classfiles. A link-time type checker has been implemented, yielding only a modest increase to the size of the TCB. The problem of inferring type annotations from legacy code base was shown to be NP-complete. Preliminary experiences in the design and implementation of type inference algorithms were discussed.

A Proof of Theorem 1

The type inference problem of DOC can be viewed in abstract as the following optimization problem:

MAX-SCC **Problem (Optimization Version):** Given a digraph G and a list L of pairs of arcs, at least one arc from each pair in L is to be selected and added to G. Which combination of arc selection will result in a graph with the maximum number of strongly connected components?

The decision version of this problem can be formulated as follows.

MAX-SCC **Problem (Decision Version):** Given a digraph G, a list L of pairs of arcs, and a positive integer K, a set S of arcs is to be selected from L, such that for each pair of arcs in L, at least one arc belongs to S. Does the digraph G^* obtained by adding S to G has at least K strongly connected components?

This appendix provides a formal proof that MAX-SCC is NP-complete. The result is obtained by a reduction from the following intermediate problem.

Acyclicity Maintenance (AM**) Problem:** Let $G(V, A)$ be an acyclic digraph and L be a list of pairs of arcs. For each arc $(u, v) \in L$, $u, v \in V$ but $(u, v) \notin A$. Suppose a set S of arcs is to be selected from L, such that for each pair of arcs in L, at least one arc belongs to S. Can S be selected in a way that the digraph G^* obtained by adding S to G contains no directed cycle?

A special case of MAX-SCC, the problem AM can be shown to be NP-complete by a reduction from the following problem:

1-IN-3 3SAT **Problem:** Given a Boolean formula ϕ in conjunctive normal form such that each clause contains three positive literal, is there a satisfying truth assignment for ϕ such that each clause in ϕ has exactly one true literal?

1-IN-3 3SAT can be shown to be NP-complete by a reduction from 3SAT [16].

Theorem 2 AM *is NP-complete.*

PROOF. We first show that the AM problem belongs to NP. Suppose we are given an acyclic digraph G and a list L of pairs of arcs. A nondeterministic algorithm needs only guess a set S of arcs. For each arc in S, checking whether it is in L. For each pair of arcs in L, checking whether at least one arc from the pair belongs to S. Then, checking whether the digraph obtained by adding S to G does not have any directed cycle. It is easy to see that these checks can be accomplished in polynomial time. Thus, AM belongs to NP.

We then show that the AM problem is NP-hard by proving a reduction of 1-IN-3 3SAT to AM. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be an instance of 1-IN-3 3SAT problem, i.e., a Boolean formula in conjunctive normal form in which each clause contains three positive literals. Let $C_i = x_{i1} \vee x_{i2} \vee x_{i3}$, for $1 \leq i \leq m$.

We now construct an acyclic digraph $G = (V, A)$ and a list L of pairs of arcs. Let $V = \{v_1, v_2, \ldots, v_{6m}\}$. Construct the arc set A as follows:

$$A = \{(v_{6i-4}, v_{6i-3}), (v_{6i-2}, v_{6i-1}), (v_{6i}, v_{6i-5}) : 1 \leq i \leq m\}.$$

The intention is that the subgraph induced by $V_i = \{v_{6i-5}, v_{6i-4}, \ldots, v_{6i}\}$ represents the clause C_i, for $1 \leq i \leq m$.

We next construct a list L of pairs of arcs. For $1 \leq i \leq m$ and $1 \leq j \leq 3$, let the arc $a_{ij} = (v_{6(i-1)+2j-1}, v_{6(i-1)+2j})$ correspond to the literal x_{ij}. Also, let $\bar{a}_{ij} = (v_{6(i-1)+2j}, v_{6(i-1)+2j-1})$. The list L will be constructed in terms of a_{ij} and $\bar{a}_{ij}$. We first create $3m$ pairs of arcs for L as follows: For each clause $C_i = x_{i1} \vee x_{i2} \vee x_{i3}$, $1 \leq i \leq m$, we create a set A_i of 3 pairs of arcs:

$$A_i = \{\{a_{i1}, a_{i2}\}, \{a_{i2}, a_{i3}\}, \{a_{i3}, a_{i1}\}\}.$$

Then, for each variable that appears more than once in ϕ, e.g., k times, we create a set of $2k$ pairs of arcs for L. Let y be such a variable that appears k times such that $y = x_{i_1 j_1} = x_{i_2 j_2} = \cdots = x_{i_k j_k}$, where the subscripts $i_1 j_1, i_2 j_2, \ldots, i_k j_k$ form a lexicographic order. We create a set B_y of $2k$ pairs of arcs

$$\begin{aligned} B_y = \quad & \{\{a_{i_1 j_1}, \quad \bar{a}_{i_1 j_1}\}, \{\bar{a}_{i_1 j_1}, a_{i_2 j_2}\}, \{a_{i_2 j_2}, \bar{a}_{i_2 j_2}\}, \{\bar{a}_{i_2 j_2}, a_{i_3 j_3}\}, \ldots, \\ & \{a_{i_k j_k}, \quad \bar{a}_{i_k j_k}\}, \{\bar{a}_{i_k j_k}, a_{i_1 j_1}\}\}. \end{aligned}$$

We now finish the construction of L, which is the union of A_i ($1 \leq i \leq m$) and B_y for each variable y appears more than once in ϕ. In order to show the relations between the elements in L, we introduce an undirected graph $G(L)$, called the graph of L, such that each arc in L corresponds to a vertex in $G(L)$, and each element (i.e., pair of arcs) in L corresponds to an edge in $G(L)$. Fig. 6 illustrates the construction of G and L. It is easy to see that we can construct the above G and L in polynomial time.

We claim that there exists a satisfying truth assignment for ϕ such that each clause in ϕ has exactly one true literal if and only if there exists a set S of arcs satisfying the following three conditions: (1) Each arc in S is contained in L; (2) for each pair of arcs in L, at least one arc belongs to S; and (3) the digraph G^* obtained by adding S to G does not have directed cycles. We first suppose that there exists a satisfying truth assignment for ϕ such that each clause in ϕ has exactly one true literal. For each clause C_i, $1 \leq i \leq m$, containing exactly two false literals, say, x_{ij} and $x_{ij'}$, $j \neq j'$, we put the corresponding two arcs a_{ij} and $a_{ij'}$ into S (see Fig. 7). Since a_{ij} and $a_{ij'}$ are contained in A_i, condition (1) holds. Notice that each pair of arcs in A_i contains a_{ij} or $a_{ij'}$. For each variable y appearing $k \geq 2$ times in ϕ and its corresponding set B_y, if y is false, then S contains k arcs $a_{i_1 j_1}, a_{i_2 j_2}, \ldots, a_{i_k j_k}$. Thus, each pair of arcs in B_y contains at least one arc in S. If y is true, then we put k arcs $\bar{a}_{i_1 j_1}, \bar{a}_{i_2 j_2}, \ldots, \bar{a}_{i_k j_k}$ into S. Hence each pair of arcs in B_y contains at least one arc in the updated S. After we consider all clauses C_i and variables y, we obtain a set S of arcs which satisfies conditions (1) and (2). We now consider the digraph G^* obtained by adding S to G. We can partition $V(G^*)$ into m subsets $V_i = \{v_{6i-5}, v_{6i-4}, \ldots, v_{6i}\}$, $1 \leq i \leq m$ such that V_i corresponds to the clause C_i. It is easy to see that no arcs in G^* connecting two vertices in V_i and V_j ($j \neq i$) respectively. The digraph induced by V_i is denoted by $\langle V_i \rangle$. For any $\langle V_i \rangle$, it is easy to check this digraph has no directed cycle. For example, let x_{i1} be the true literal in C_i. If variable x_{i1} appears only once in ϕ, then $\langle V_i \rangle$ has the arc set

$$\{(v_{6i-4}, v_{6i-3}), (v_{6i-3}, v_{6i-2}), (v_{6i-2}, v_{6i-1}), (v_{6i-1}, v_{6i}), (v_{6i}, v_{6i-5})\}.$$

If variable x_{i1} appears more than once in ϕ, then $\langle V_i \rangle$ has the arc set

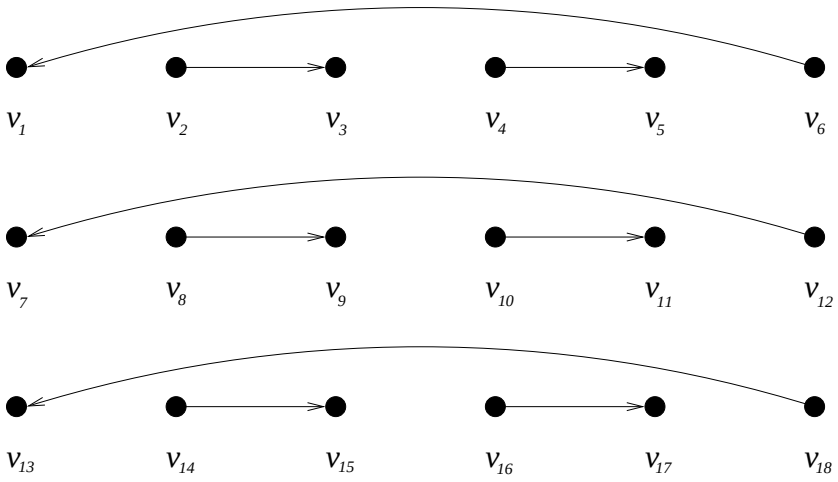
$$\{(v_{6i-4}, v_{6i-3}), (v_{6i-3}, v_{6i-2}), (v_{6i-2}, v_{6i-1}), (v_{6i-1}, v_{6i}), (v_{6i}, v_{6i-5}), (v_{6i-4}, v_{6i-5})\}.$$

Thus, the digraph G^* has no directed cycle. Hence, condition (3) holds.

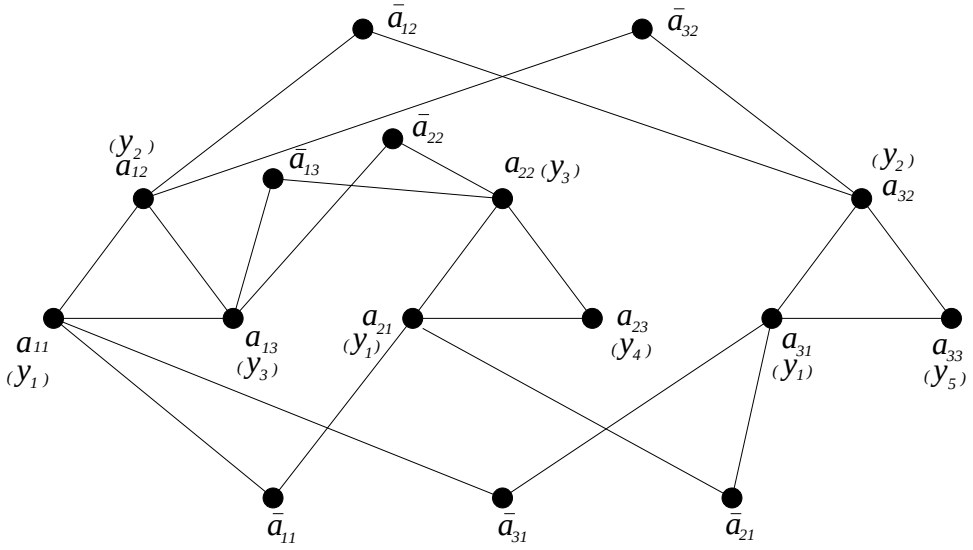
Conversely, suppose that there exists a set S of arcs satisfying the above three conditions. If S contains three different arcs in A_i, then G^* has a directed cycle

$$\{(v_{6i-5}, v_{6i-4}), (v_{6i-4}, v_{6i-3}), (v_{6i-3}, v_{6i-2}), (v_{6i-2}, v_{6i-1}), (v_{6i-1}, v_{6i}), (v_{6i}, v_{6i-5})\}.$$

This contradicts condition (3). If S contains only one arcs in A_i, then there exist a pair of arcs in $A_i \subseteq L$ in which both arcs are not in S. This contradicts condition (2). Thus, S contains exactly two different arcs in A_i, $1 \leq i \leq m$. We can set FALSE to the two literals corresponding to these two arcs, and set TRUE to the remaining literal in C_i. We now show that any variable y that appears $k \geq 2$ times in ϕ has the same truth assignment. Let $y = x_{i_1 j_1} = x_{i_2 j_2} = \cdots = x_{i_k j_k}$, where the subscripts $i_1 j_1, i_2 j_2, \ldots, i_k j_k$ form a lexicographic order. Assume that not all these literals have the same truth assignment. There must exist two literals $x_{i_p j_p}$ and $x_{i_{p+1} j_{p+1}}$ such that $x_{i_p j_p}$ =FALSE and $x_{i_{p+1} j_{p+1}}$ =TRUE, where $p + 1$ =mod(k). Thus, $a_{i_p j_p} \in S$ and $a_{i_{p+1} j_{p+1}} \notin S$. Since $(\bar{a}_{i_p j_p}, a_{i_{p+1} j_{p+1}})$ is a pair of arcs in $B_y \subset L$,

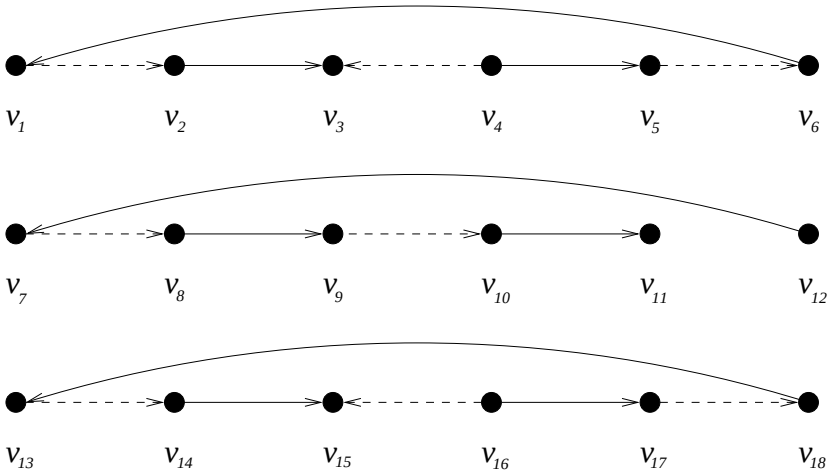


(a) G

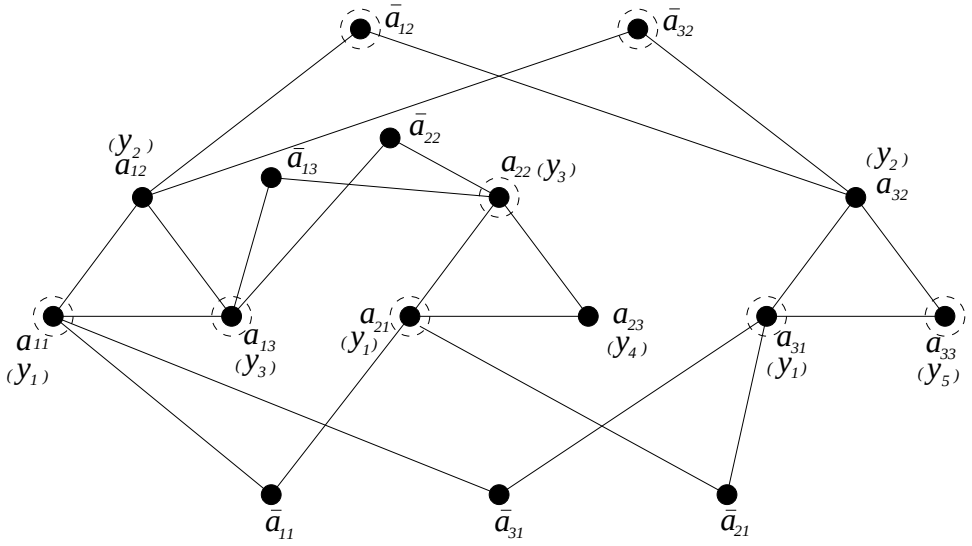


(b) The graph of L

Figure 6: The graph G and $G(L)$ constructed from an instance of 1-IN-3 3SAT $(y_1 \vee y_2 \vee y_3) \wedge (y_1 \vee y_3 \vee y_4) \wedge (y_1 \vee y_2 \vee y_5)$.



(a) G*



(b) The graph of L

Figure 7: Suppose $y_2 = y_4 =$TRUE and $y_1 = y_3 = y_5 =$FALSE for formula $(y_1 \vee y_2 \vee y_3) \wedge (y_1 \vee y_3 \vee y_4) \wedge (y_1 \vee y_2 \vee y_5)$. The dashed arcs in (a) belong to S. The dashed vertices in (b), which correspond to the false literals, also correspond to the arcs in S.

it follows from condition (2) that $\bar{a}_{i_p j_p} \in S$. Since $a_{i_p j_p}$ and $\bar{a}_{i_p j_p}$ form a directed cycle, this contradicts condition (3). Hence, any variable that appears more than once in ϕ has the same truth assignment. Therefore, there exists a satisfying truth assignment for ϕ such that each clause in ϕ has exactly one true literal. ■

Theorem 3 MAX-SCC *is NP-complete.*

PROOF. We first show that the MAX-SCC problem belongs to NP. Suppose we are given a digraph G, a list of pairs of arcs L, and an positive integer K. A nondeterministic algorithm needs only guess a set S of arcs. For each arc in S, checking whether it is in L. For each pair of arcs in L, checking whether at least one arc from the pair belongs to S. Since computing the strongly connected components of a digraph can be performed in linear time [28], It is easy to see that SCC belongs to NP.

We now show that the MAX-SCC problem is NP-hard by establishing a reduction of AM to MAX-SCC. Given an instance of the AM problem, that is, an acyclic digraph G and a list L of pairs of arcs, we construct an instance of MAX-SCC by setting $\bar{G} = G$, $\bar{L} = L$ and $K = |V|$. This construction needs linear time. Let S be a set of arcs selected from L such that for each pair of arcs in L, at least one arc belongs to S. Let G^* be the digraph obtained by adding S to G. We can set $\bar{S} = S$ and $\bar{G}^* = G^*$ for $\bar{G}$ and $\bar{L}$. It is easy to see that G^* is acyclic if and only if $\bar{G}^*$ has at least K strongly connected components. ■

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004.

[2] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 213–223, New Orleans, Louisiana, USA, January 2003.

[3] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, pages 2–27, Budapest, Hungary, July 2001.

[4] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 292–310, Seattle, Washington, USA, November 2002.

[5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64, Vancouver, BC, Canada, October 1998.

[6] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the 5th ACM Symposium on Operating System Principles*, pages 141–160, Austin, Texas, USA, November 1975.

[7] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[8] Philip W. L. Fong. The Aegis VM Project. `http://aegisvm.sourceforge.net`., 2004.

[9] Philip W. L. Fong. Link-time enforcement of confined types for JVM bytecode. Technical Report CS-2004-12, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada, December 2004. ISBN 0-7731-0505-0.

[10] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 404–418, Vancouver, B.C., Canada, October 2004.

[11] Philip W. L. Fong. *Proof Linking: A Modular Verification Architecture for Mobile Code Systems*. PhD dissertation, School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada, January 2004.

[12] Philip W. L. Fong and Robert D. Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology*, 9(4):379–409, 2000.

[13] Philip W. L. Fong and Robert D. Cameron. Proof linking: Distributed verification of Java classfiles in the presence of multiple classloaders. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 53–66, Monterey, California, USA, April 2001.

[14] Philip W. L. Fong and Cheng Zhang. Capabilities as alias control: Secure cooperation in dynamically extensible systems. Technical Report CS-2004-3, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada S4S 0A2, 2004. ISBN 0-7731-0479-8.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[17] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 241–253, Tampa Bay, FL, USA, October 2001.

[18] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[19] Chris Howblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, USA, June 1998.

[20] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[21] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, May 2001.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[23] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, Brussels, Belgium, July 2004.

[24] Jonathan A. Rees. A security kernel based on the lambda-calculus. A. I. Memo 1564, MIT, 1996.

[25] Nathanael Schärli, Andrew P. Black, and Steéphane Ducasse. Object-oriented encapsulation for dynamically typed languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 130–149, Vancouver, B.C., Canada, October 2004.

[26] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Composable encapsulation policies. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004.

[27] Michael D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. thesis, MIT, 1972.

[28] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[29] Jan Vitek and Boris Bokowski. Confined types in Java. *Software - Practice & Experience*, 31(6):507–532, May 2001.

[30] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 135–148, Anaheim, California, USA, October 2003.