移除動態繫結的程式碼混亂化

Obfuscation Based on Removing Dynamic Binding

賴政良 周敬斐 廖賀田*
Jeng-Liang Lai Ching-Fei Chou Heh-Tyan Liaw
淡江大學資訊管理研究所 淡江大學管理科學研究所 淡江大學資訊管理研究所
註: *為通訊作者, Email: htliaw@mail.tku.edu.tw

摘要

物件導向語言 Java 已是非常流行的程式語言，但它的**類別檔**（class file）中含有大量的程式資訊，因而容易被**逆編譯**（de-compile）。本論文提出一套程式碼混亂器。它以去除**動態繫結**（dynamic binding）為手段來進行**混亂化**（obfuscation），使混亂後的程式碼更不容易被研讀，也使智慧財產能得到更好的保護。

關鍵詞：動態繫結、程式碼混亂化。

Abstract

Java, the object-oriented language, is a very popular programming language. But its class file, containing vast code information, is easy to be de-compiled. An obfuscation tool based on removing dynamic binding is proposed in this paper. After the obfuscation, code is more difficult to be studied, and the intellectual property will be protected in a better manner.

Keyword:Dynamic binding、Code Obfuscation.

一、緒論

（一）研究動機

有許多軟體系統開發者會擔心他們的程式（application）由於**逆向工程**（reversed engineering）而導致關鍵技術遭竊，相關的侵權訴訟案件也時有所聞[7]。

物件導向（object-oriented）技術[9]是掌握程式碼複雜度的優良工具，這種技術主要是包含**封裝**（encapsulation）、**繼承**（inheritance）與**動態繫結**（dynamic binding, 又稱為虛擬呼叫, virtual call)三大部份。

Java 語言已經相當普遍地被用來開發應用程式[17]，它支援**物件導向**技術，但編譯後的**類別檔**（class file）[18]包含有完整的資訊，如檔名（file names）、**繼承**（inheritance）關係、**覆寫**（overriding）關係等。類別檔和**執行檔**（executables）相較，更容易被**逆編譯**（de-compile）。這使得運用物件導向技術的細節都公開於世。

Java的**實體函數**（instance method）-- 除了配置**私有存取權**（private accessibility）的以外-- 都是動態繫結。

實體函數的呼叫是經由**物件**（object）的**指標**（pointer, java文件將它稱為reference）來發動。**動態繫結**是等到執行期才根據指標所指物件的真實類別而決定要被叫用的是哪一個函數。這種機制使程式碼的可讀性大幅提高[9]。

本論文以去除動態繫結為手段來對Java程式原始碼進行混亂化。混亂化後的程式碼語意不變（semantics preserving），但不再直接使用動態繫結的機制，這使程式碼的複雜度增加，也就提高了有心人士對程式碼的研讀成本，因此軟體的智慧財產就能得到更強的保護[5][7]。另外，經本系統處理過的程式碼仍可再透過其它的混亂化技術加以處理。

（二）相關研究

當前在程式碼混亂化的研究當中，可以歸納出下列幾種作法[7]：(i)**語彙轉換**（layout transformation），例如：混亂程式碼的排版結構、更名、刪除註解等。 (ii)**控制流程轉換**（control flow transformation），例如改變迴圈條件、或是扁平化整個控制流程[3]。 (iii)**資料轉換**（data transformation），例如分割變數（split variables）、重建陣列（restructure arrays）等[6][13]。

目前一些混亂化的軟體如yGuard [19]、Jarg [10]、與Marvin Obfuscator [8]等，都是以語彙轉換為主。另外如RetroGuard [15]、Zelix KlassMaster [20]、DashO [14]等，它們都只是著重於控制流程轉換和資料轉換。

物件導向技術減輕了**程序導向**（procedure-oriented）時代的**軟體危機**（software crisis），是當代軟體界的重點技術[9]。但以上的相關研究與工具都只是在程序導向的範圍內操作。就Java語言來說，程式的技術重點是在於繼承及動態繫結。程式碼中的各個函數內容通常並不長，這使得目前一些混亂化軟體的手法變成英雄無用武之地。

（三）論文概觀

本論文的主題是藉著自動移除物件導向技術中的動態繫結機制來對Java程式碼做混亂化。我們已實作一套**程式碼混亂器**（以下稱本系統）來進行這個工作。我們所處理的Java原始碼都已先假設為能夠通過編譯且正確執行。

要去除程式碼中的動態繫結機制，就要將實體函數轉為類別函數。並且將**虛擬呼叫**（virtual call）[4]改為**靜態呼叫**（static call）。由於類別函數都是用**靜態繫結**（在編譯期就確定所要呼叫的函數），因此本系統另外製作**發配函數**（dispatch function）與**預發配函數**（pre-dispatch function）來模擬動態繫結。

轉換後的程式執行起來的功能和先前相同，但執行的流程變得非常混亂。

在討論上，本文將程式碼區分為**內部**（internal）與**外部**（external）。內部程式碼是指專案（project）內的程式碼，外部程式碼是指那些被使用到的**程式庫**（libraries）。這些程式庫中的類別必定歸屬於一些**套件**（packages）之中。

程式庫原則上不該被更動，但若使用者堅持要更動，就要自己負責取得程式庫的原始碼並將它們納入為專案的一部份。

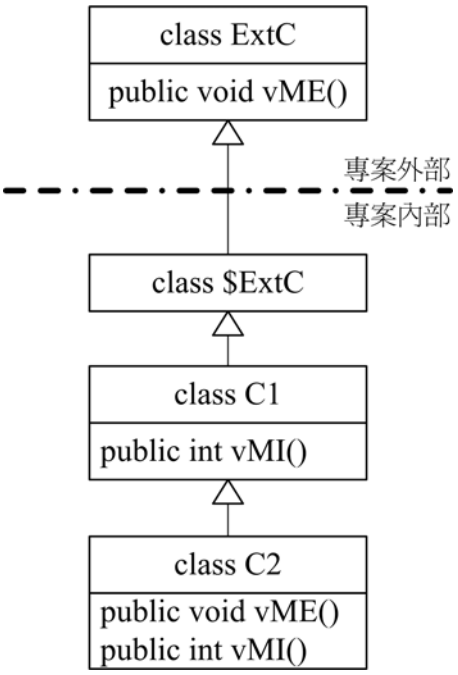

圖 1 實體函數植根類別說明圖

以下我們在第二節討論如何對純屬於內部的實體函數移除動態繫結。第三節討論繼承攔截，這是為了要處理跨越專案邊界的實體函數。第四節討論如何對牽涉到外部的實體函數去除動態繫結。第五節為系統測試與評估。最後一節是結論與展望。

二、去除植根於內部的實體函數

為了便於討論，本論文定義**植根類別**（rooted class）。對寫在類別 C 中的實體函數 VM()，考慮 C 及所有 C 的祖先（ancestor）類別，若 VM()的抽象宣告或實作最高是出現在類別 C_r，則稱 VM()**植根於** C_r，又稱 C_r 為 VM()的**植根類別**。以圖 1 為例，ExtC 為 vME()的植根類別，而 C1 為 vMI()的植根類別。

本節討論如何移除植根於內部的實體函數。首先是由實體函數抄寫出對應的類別函數，然後製作發配函數，接著將虛擬呼叫轉成發配函數的呼叫，最後刪除實體函數。整個程序分述如下。

（一）由實體函數抄製類別函數

我們由實體函數抄寫出對應的類別函數。若某個類別 C 中有 N 個實體函數，則在抄寫後，類別 C 將新增 N 個對應的類別函數。圖 2 是抄寫動作完成後的一個典型實例，其中函數下方標底線者為新增的類別函數。

以圖 2 的類別 B 為例，對實體函數 VM1()抄寫出類別函數的步驟如下： (a)在同個類別內新增一個同名的類別函數 VM1()。參數除了原先實體函數 VM1()的參數以外，還新增一

個參數“$that”，型別是實體函數 VM1()所屬的類別 B。(b)將實體函數 VM1()的內容抄寫至類別函數 VM1()中。(c)調整類別函數 VM1()中的程式碼，將 this 指標更換成參數$that。(d) 調整類別函數 VM1()中涉及 *super* 指標的程式碼，包括：(i)對於以 *super* 存取欄位的情況，將 *super* 改成$that，並向上轉型(up-cast)成為父類別的指標來進行存取。(ii) 對於以 *super* 存取函數的情況，則將經由 *super* 指標的函數呼叫轉成父層中所對應的類別函數的呼叫。程式碼實例請參閱表 1，粗體字部分為抄製後的類別函數。

（二）在植根類別建立發配函數

動態繫結在執行期根據物件實體的型別，呼叫正確的實體函數。本系統透過**發配函數**（dispatch method）來模擬動態繫結，使去除動態繫結後程式的功能不變。我們將所有的內部類別都各別配置一個不會重複的整數欄位 clsID 當作**類別識別編號**。這個欄位將由各個類別的建構元（constructor）負責設值。發配函數在執行期透過這些編號來判定物件的類別，然後轉呼叫對應的類別函數。以圖 2 為例，類別 B、A、Base 的識別編號分別為 1, 2 與 3。

植根於內部的實體函數若沒有被內部其他類別覆寫（override），就沒有動態繫結的需求。於是我們就不替它建立發配函數。

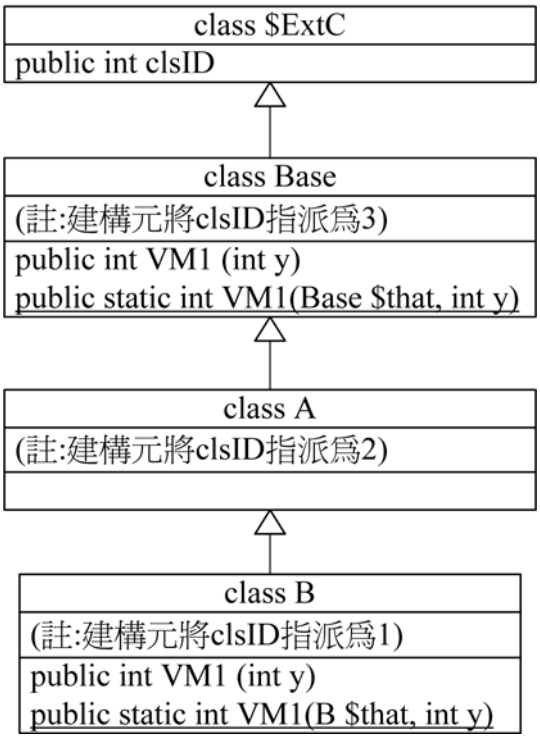

圖 2 植根於內部的實體函數

在圖 2 中的實體函數 VM1(int y)植根於類別 Base，只有類別 B 覆寫 VM1(int y)。 於是我們在植根類別 Base 建立一個發配函數，它的本體是一個 *switch* **敘述**（statement）。本系統從植根類別開始，以**深度優先**（depth-first）的方式探訪各個子類別。子類別若有覆寫實體函數，則它的**類別識別編號**就作為 *case* 的條件值。讓發配函數在執行期時，能夠根據物件的類別編號決定要轉呼叫哪個類別函數。實例請參閱表 2。

在建完發配函數後，原先動態繫結的功能都可改由發配函數負責模擬。

（三）更改呼叫點

實體函數若未被覆寫（override），則不發生動態繫結的需求。因此本系統將這種函數的呼叫點改成呼叫由它抄製的類別函數。

其餘的實體函數呼叫點可以分為三種形式：(i)以普通指標呼叫實體函數。(ii)以 *this* 指標呼叫實體函數。(iii)以 *super* 指標呼叫實體函數。(i)與(ii)情況相同，本系統將它們改成呼叫對應的發配函數。

依據 Java 的規格書，(iii)的情況在編譯期就可以確定所繫結(bind)的實體函數。本系統從父類別往上查找，將原先由 *super* 發動的實體函數呼叫改成呼叫由繫結目標所抄製的類別函數。

我們以一個實例來展示：表 3 是原先的程式碼，表 4 是更換函數呼叫後的結果。

（四）刪除實體函數

去除動態繫結的過程至此，植根於內部的實體函數將不再被呼叫。我們就可以刪除不再被需要的實體函數。

但還有兩種情況須特別考慮：(i)實體函數負責**實作**（implement）Java 語言的介面（interface）[11]、(ii) 實體函數是**事件處理函數**（event-handling method）[12]。由於此類函數必須保留，本系統就將它們的函數本體改成只有一行，只負責轉呼叫**發配函數**。單單由外貌很難判定一個函數是否是個事件處理函數（例如負責重畫 swing 視窗的標準方法 paintComponent）。所以本系統要求使用者在這種函數前加上特別的註記（annotation）以資識別[11]。

三、繼承攔截

當實體函數植根於專案外部時，去除動態繫結就必須跨越專案邊界。由於**程式庫**（library）本來就不應該被更動，本系統在專案內部新增一些**攔截類別**（interception class），讓這些攔截類別成為內外部之間繼承關係的關卡。處理的步驟分四小節敘述如下。

（一）建立攔截類別

建立攔截類別的步驟包括：(i)搜尋**跨邊界類別**：跨邊界類別是指那些被內部類別直接繼承的程式庫類別。以圖 3 為例，類別 ExtC 就是跨邊界類別。(ii)建立**攔截類別**：在專案內部新增相對應的攔截類別，這些攔截類別的名稱將依照跨邊界類別的名稱加上“$”命名。攔截類別將繼承跨邊界類別。

Java 語言中的所有類別都直接或間接繼承了類別 java.lang.Object[11]. Object 當然是專案外部的類別，因此本系統也將為 java.lang.Object 建立一個名為$Object 的攔截類別。$Object 還用來放置在第四節討論的預發配函數。

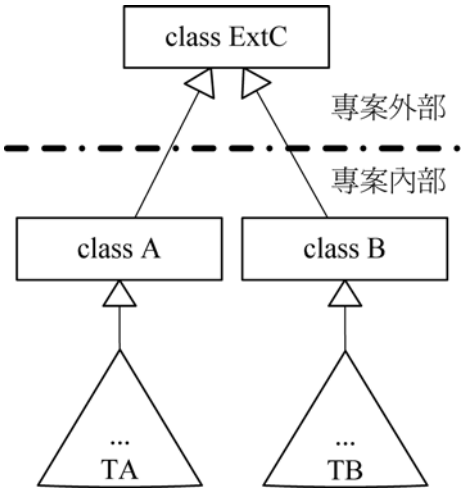

圖 3 繼承攔截前類別繼承樹示意圖

（二）調整繼承關係

攔截類別建立後，原先繼承跨邊界類別的內部類別將改為繼承攔截類別。以圖 3 為例，原本類別 A、B 繼承類別 ExtC，在建完攔截類別之後，就改繼承攔截類別$ExtC。轉換後的繼承關係如圖 4。

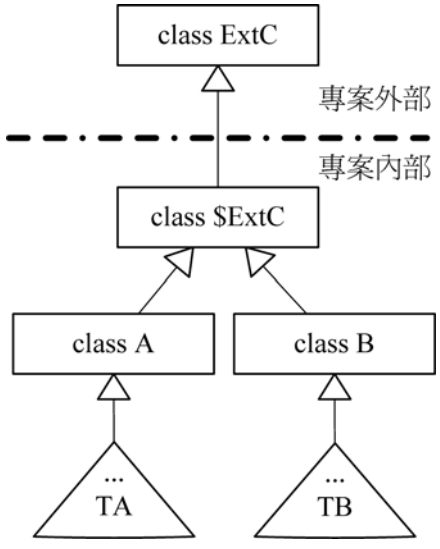

圖 4 繼承攔截後類別繼承樹示意圖

（三）為外部的保護型欄位建存取函數

在去除跨越專案的動態繫結後，實體函數將轉為類別函數。因類別函數沒有 this 或是 super 可存取外部父類別的保護型欄位。因此我們在攔截類別中建立各種存取函數，如**設值函數**（setter）及**取值函數**（getter），使專案內部的類別函數得以存取父類別的保護型欄位。

（四）為外部的實體函數建純上轉函數

類別函數沒有 super 指標可叫用專案外部（公開或保護型）的實體函數。為了配合 Java 語言對使用 super 指標呼叫實體函數的特殊規定，我們還必須在攔截類別中新增一些純上轉函數（細節略）。實例請參閱表 5。

四、去除植根於外部的實體函數

在函數植根於外部類別的情況下，指標所指的物件在執行期的真實型態是植根類別的某個子類別 -- 這個子類別可能屬於某個內部繼承樹，也可能是在外部。因此在執行流程進入發配函數之前還要先有**預發配函數**（pre-dispatch function）來判定應該轉呼叫哪個攔截類別中的發配函數，當真實型態在外部時，還必須交由外部進行動態繫結。

對植根於外部的實體函數去除動態繫結，在程序上我們也分四個小節討論。

（一）由實體函數抄製類別函數

本節在作法上與第二節相同。

（二）建立發配函數與預發配函數

植根於外部的實體函數若從未被內部程式所覆寫（override），就不為它建立發配函數或是預發配函數。

對植根於外部的實體函數，本系統一律在內部繼承樹的樹根（也就是**攔截類別**）中為它建立對應的**發配函數**（dispatch method）。發配函數的作法和植根於內時相同，差別只在發配函數所歸屬的類別。

除此之外，我們還需要**預發配函數**（pre-dispatch method）。這個函數是攔截類別$Object 的類別函數，它的本體（function body）在執行期利用 *instanceof* **運算子**（operator）來判斷指標所指的物件的真實類別是否在某個內部繼承樹之中。若是，就轉呼叫樹根(攔截類別)中所對應的發配函數；而若所指的物件不屬於任何一棵內部繼承樹，就交還給外部做動態繫結。

以圖 5 的實體函數 VM2(int x)為例，在本系統為它產生的**預發配函數**（實例請參閱表 6）有一個名為 obj 的參數。這個參數的型別就是 VM2()所屬的繼承樹中最高的公開宣告類別 OA，預發配函數中的第二個參數 x 就是原先 VM2()的參數。在這個預發配函數的本體中，*if* 敘述以 *instanceof* 來判斷 obj 在執行期的真實型別。如果 obj 屬於內部某個繼承樹，就將 x 當作參數轉呼叫對應的發配函數。否則就交由外部進行動態繫結。

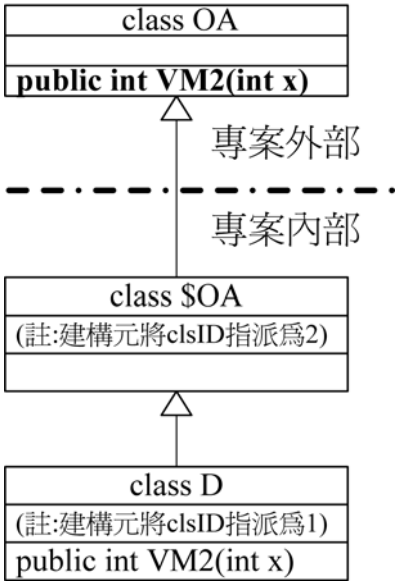

圖 5 植根於外部的實體函數

（三）更換函數呼叫點

（植根於外部的）實體函數若沒被覆寫，它的呼叫點就不更動。

我們在前面討論過一般的實體函數呼叫點有三種格式：(i)以普通指標呼叫實體函數。(ii)以 *this* 指標呼叫實體函數。(iii)以 *super* 指標呼叫實體函數。形式(i)與(ii)的情況相同，就是將實體函數呼叫改成呼叫**預發配函數**。

依據 Java 的規格書，(iii)的情況在編譯期就可以確定。若 *super* 指標所繫結的實體函數是在內部，則將實

體函數呼叫改成對應的類別呼叫。若 *super* 指標所繫結的實體函數是在外部，則須將實體函數呼叫改成呼叫對應的**純上轉函數**。

表 7 的實例展示了原先的程式碼，經由本系統更換實體函數呼叫後的程式碼請參閱表 8。

（四）刪除實體函數

對植根於外部類別的實體函數來說，它在內部可能有許多個覆寫的(overriding)函數。除了那些不再被需要的實體函數可以被刪除之外，還需要特別考慮以下兩種情況：(i)實體函數負責實作 Java 語言的**介面**[11]、(ii)實體函數是**事件處理函數**[12]。由於此類函數必須保留，本系統就將它們的函數本體改成只有一行，只負責轉呼叫**預發配函數**。本系統同樣要求使用者在**事件處理函數**前加上特別的（annotation）以資識別[11]。

五、系統測試與評估

本系統已經實作完成，內部的**資料結構**（data structure）是建構在 Java 的**語法樹**（syntax tree）上[2]。藉由語法樹所提供的應用程式介面（Application Programming Interface, API）實作本論文的演算法。

系統的運作是將 Java 原始檔**剖析**（parse）並轉成語法樹。再依照去除動態繫結的程序修改語法樹，最後輸出混亂化後的程式碼。過程如圖 6 所示。

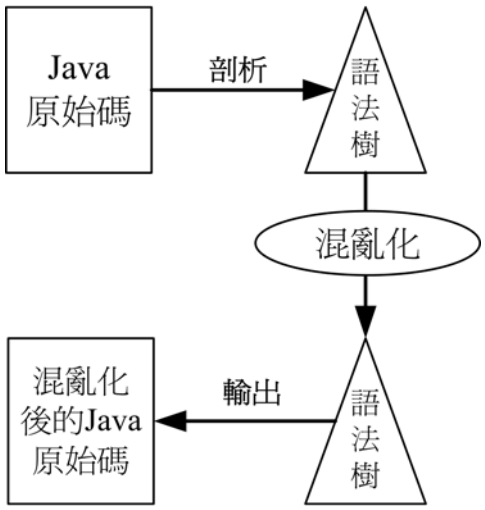

圖 6 程式碼混亂器運作架構圖

我們以一套“天文觀星系統—TwinkleSky”[1] 來進行測試，該系統模擬天球的運作，將星體投影在視窗上，讓使用者可以指定任何時間地點，觀測任何方向的天空。這套觀星系統在第十屆全國大專院校資訊管理實務專題競賽中獲得 ICT2 資訊技術組第一名。

“TwinkleSky”共有 47 個類別、42 條繼承線、164 個實體函數，五千餘行程式碼。經本系統混亂化後，程式碼仍能正確編譯並執行，執行時的畫面也很流暢。天文系統的寫作者本身對混亂化後的程式流程已經感到難以掌握。

在 Java **虛擬機器**（Java Virtual Machine）的**指令集**（instruction set）中，執行虛擬呼叫的指令為 *invokevirtual* [18]。這個指令在執行期完成動態繫結，所需花費的時間為常數時間 O(1)。

本系統的發配函數是以**類別識別編號**透過 *switch* **敘述**(statement)來模擬動態繫結。一般而言，*switch* 敘述的執行時間為線性時間 O(n)，n 為 *case* 的個數。但對負責執行 Java 程式的 Java 虛擬機(Java virtual machine)來說，有兩種不同的對應指令: 常數時間的 *tableswitch* 指令與線性時間的 *lookupswitch* 指令[18]。當原始碼中 *case* 的條件值較相近時，編譯器會將 switch 敘述譯成 *tableswitch* 指令，只有當條件值較遠離時才譯成 *lookupswitch* 指令。

本系統對專案內部的**類別識別編號**是以**深度優先**（depth-first）的次序指派。因此發配函數中各個 *case* 的條件值通常都彼此接近，而能被編譯成為 *tableswitch* 指令，因而發配函數的執行仍是屬於常數時間 O(1)。但對植根於外部的函數來說，我們的預發配函數是屬於線性時間 O(k)，k 是內部繼承樹的個數。在一般情況下，這個 k 並不會很大。

Java 的**編譯器**（compiler）通常都會進行**最佳化**（optimization）的程序。此外，官方的 Java **直譯器**（interpreter)在執行時還會進行一種稱為 JIT（Just-In-Time ）的加速動作，它會適度地將某些**位元組碼**（bytecodes）臨時編譯成**機器碼**（machine codes）[16]，因此我們不容易精確測量各個指令的實際執行時間。

為了進一步了解本系統的效能，我們另外設計如下的實驗。在測試程式裡，每個實體函數本體都有四條敘述，每條敘述會呼叫程式庫產生一個亂數值並將它設入物件的欄位。我們對每個受測函數反覆呼叫一千萬次，然後記錄混亂化前後的執行時間。

我們的實驗結果如下：對植根於內部的實體函數來說，混亂化之後所花費的平均時間和混亂化之前相比為 103.3%。對植根於外部的實體函數來說，混亂化之後所花費的平均時間和混亂化之前相比為 106.7%。

就實務上來說，函數的呼叫暨回返程序（calling sequence）多多少少總會耗用一些執行時間。但很少有人會為了省這種時間而將函數呼叫寫成塞入型（inline）的程式碼。就混亂化的目標來說，本系統雖然增加了一點執行時間，但幅度還在可接受的範圍之內。我們是以內含四條敘述的函數來做實驗。在實務上函數通常會含有更多的敘述，所以呼叫的時間耗費就會因分攤的效果而變成更不顯著。

六、結論與展望

物件導向（object-oriented）技術[9]使程式碼的複雜度更容易被掌握。而**動態繫結**（dynamic binding）是其中的重要機制。本論文提出去除程式碼中動態繫結機制的方法，並依此方法實作了**程式碼混亂器**。

當實體函數植根在專案內部時，原先的動態繫結機制改由呼叫**發配函數**（dispatch function）來模擬。發

配函數在**執行期**（runtime）藉著判定物件的**類別識別編號**轉呼叫對應的類別函數。

當實體函數植根於專案外部時，去除動態繫結就必須跨越專案邊界。本系統在專案內將新增一些**攔截類別**（interception class）當成內外部之間繼承關係的關卡。

若實體函數植根於專案外部時，本系統一律在內部繼承樹的樹根（也就是**攔截類別**）中建立對應的**發配函數**。除此之外，我們還需要**預發配函數**（pre-dispatch function）在執行期判斷指標所指的物件真實類別是否在內部繼承樹中。若是，就轉呼叫樹根中的發配函數；否則就交還給外部做動態繫結。

去除動態繫結之後，程式碼變得混亂，也使得研讀的成本提高，於是軟體的智慧財產就得到更強的保護。經過我們系統混亂化後的程式碼仍舊可以再施行其他的混亂化手段。

我們在本論文之後將繼續研發去除**繼承**（inherit）關係的程式碼混亂化方法。

參考文獻

[1] 鄭立婷、劉定衡等，“天文觀星系統--Twinkle Sky”，第十屆全國大專院校資訊管理實務專題暨資訊服務創新競賽，中華民國資訊管理學會，經濟部工業局合辦，http://www.csim.org.tw，2005/12/17。

[2] 郭肇安，“Java 的語法樹與直譯機制”，淡江大學資訊管理學系碩士論文，2006/6。

[3] Alex Kalinovsky, “Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering”, Sams, May 2004.

[4] Bjarne Stroustrup, “The C++ Programming Language 3rd Edition”, Addison-Wesley, May 1998.

[5] C. Collberg, C. Thomborson, and D. Low, “A Taxonomy of Obfuscating Transformations”, Technical Report #148, Department of Computer Science, University of Auckland, July 1997.

[6] C. Collberg, C. Thomborson, and D. Low, “Breaking abstractions and unstructuring data structures”, Proc. IEEE International Conference on Computer Languages p 28-38, May 1998.

[7] C. S. Collberg, and C. Thomborson, “Watermarking, Tamper-proofing, and Obfuscation - Tools for software protection”, In IEEE Transactions on Software Engineering, volume 28, pages 735–746, August 2002.

[8] Dr. Java, “Marvin Obfuscator”, http://www.drjava.de/obfuscator/, 2007

[9] Grady Booch, “Object-Oriented Analysis and Design”, Addison-Wesley 1994.

[10] Hidetoshi Ohuchi, “Jarg”, http://jarg.sourceforge.net/, 2003.

[11] K. Arnold, J. Gosling, and D. Holmes, “The Java Programming Language 4th edition”, Addison-Wesley, November 2005.

[12] K. Walrath, M. Campione, A. Huml, S. Zakhour, “The JFC Swing Tutorial: A Guide to Constructing GUIs, 2nd Edition”, Prentice Hall PTR, February 2004.

[13] L. Ertaul, S. Venkatesh, “Novel Obfuscation Algorithms for Software Security”, Proceedings of the 2005 International Conference on Software Engineering Research and Practice, SERP’05, Las Vegas, June 2005.

[14] PreEmptive, “DashO”, http://www.preemptive.com/products/dasho/index.html, 2007.

[15] Retrologic Systems, “RetroGuard”, http://www.retrologic.com/retroguard-main.html, 2007.

[16] Sun Microsystems, Inc., “Java SE HotSpot at a Glance”, http://java.sun.com/javase/technologies/hotspot/, 2007.

[17] TIOBE Software BV, “TIOBE Programming Community Index for September 2007”, http://www.tiobe.com/tpci.htm

[18] T. Lindholm, and F. Yellin, “The Java Virtual Machine Specification 2nd Edition”, Addison-Wesley April 1999.

[19] yWorks, “yGuard”, http://www.yworks.com/en/products_yguard_about.htm, 2007.

[20] Zelix Pty Ltd, “Zelix KlassMaster”, http://www.zelix.com/klassmaster/features.html, 2007.

表 1 實體函數抄製成類別函數

```
class Base extends $ExtC
{
  /* 略 */
  public int w;
  public int VM1(int y) { return w + y; }
  public static int VM1
    (Base $that, int y)
  {
      return $that.w + y;
  }
}
class A extends Base { /* 略 */ }
class B extends A {
  public int x;
  public int VM1(int y) {
   return super.w+this.x+super.VM1(y);
  }
  public static int VM1(B $that, int y)
  {
    return ((B)$that).w +
      $that.x +
      Base.VM1($that,y);
  }
}
```

表 2 靜態發配函數之程式碼

```
class Base extends $ExtC {
  public static int dispatch$VM1
    (Base obj, int y)
  {
    switch ( obj.clsID ) {
      case 1:
        retrun B.VM1((B)obj, y);
      case 2:
        return Base.VM1(obj, y);
      case 3:
        retrun Base.VM1(obj, y);
      default:
        throw new

RuntimeException( “Err!” );
    }
  }
  /* 略 */
}
```

表 3 更換內部實體函數呼叫之前

```
class C extends B{
  public int VM1(int y){ /*略*/ }
  public static int VM1(C $that, int y)
    { /*略*/ }
  public void testVM1(B obj){
    obj.VM1(1);
    this.VM1(2);
    super.VM1(3);
  }
  /* 略 */
}
```

表 4 更換內部實體函數呼叫之後

```
class C extends B{
  public int VM1(int y){ /*略*/ }
  public static int VM1(C $that, int y)
    { /*略*/ }
  public void testVM1(B obj){
    Base.dispatch$VM1(obj, 1);
    Base.dispatch$VM1(this, 2);
    Base.VM1(this, 3);
  }
  /* 略 */
}
```

表 5 建立純上轉函數之結果

```
//: 外部程式碼
class ExtC {
  public int VM1(int x) { /* 略 */ }
}
//: 內部程式碼
class $ExtC extends ExtC {
  public int super$VM1(int x){
    return super.VM1(x);
  }
  /*略*/
}
class A extends $ExtC {
  public void VM2() {
    //: 原來爲 super.VM1(10);
    this.super$VM1(10);
    /*略*/
  }
}
```

表 6 建立預發配函數之結果

```
class $Object extends Object{
  public static int
    preDispach$OA$VM2(OA obj, int x)
  {
    if(obj instanceof $OA){
      return
       $OA.dispatch$VM2( ($OA)obj, x);
    }
    return obj.VM2(x);
  }
  /* 略 */
}
```

表 7 更換外部實體函數呼叫之前

```
//: 外部類別
class OA {
  public int VM2(int x){ /*略*/ }
}
//: 內部類別
class D extends OA{
  public int VM2(int x){ /*略*/ }

  public void testVM2(OA obj){
    obj.VM2(1);
    this.VM2(1);
    super.VM2(1);
  }
}
```

表 8 更換外部實體函數呼叫之後

```
class $Object extends Object{
  public static int preDispatch$OA$VM2
    (OA obj, int y)
  { /*略*/ }
}

class $OA extends OA{
  public int super$VM2(int x){
    return super.VM2(x);
  }
  public static int dispatch$VM2
    ($OA obj, int x)
  { /*略*/ }
}

class D extends $OA{
  public int VM2(int x){ /*略*/ }
  public static int VM2($OA obj, int x)
  { /*略*/ }

  public void testVM2(OA obj){
    $Object.
      preDispatch$OA$VM2(obj, 1);
    $Object.
      preDispatch$OA$VM2(this, 1);
    this.super$VM2(1);
  }
}
```