國立交通大學

資訊科學系

碩 士 論 文

以定性的有限狀態機為基礎的 Java 程式混淆

DFA-based Obfuscation of Java Programs

研 究 生：趙博民

指導教授：楊　武　教授

中 華 民 國　九 十 四　年 一 月

以定性的有限狀態機為基礎的 Java 程式混淆

學生：趙博民　　指導教授：楊　武　博士

國立交通大學資訊科學研究所

摘要

為了達到跨平台的特性，Java 程式會被編譯成一種以堆疊操作為導向，與硬體無關的中間型式，這種中間型式稱為 bytecode。然而採用此種型式的指令集，雖造就了跨平台的特性，卻衍生了新的問題：bytecode 及 Java 虛擬機器的特性使得反編譯 Java 程式變得更加容易。

本論文提出一種新的混淆技術來保護 Java 軟體。使得攻擊者難以僅利用反編譯器就可以直接獲取正確無誤的原始碼，迫使攻擊者一定要以手動或其它輔助工具進行反編譯，進而增加反編譯的難度，促使其感到灰心、氣餒，因而放棄反編譯之非法意圖。

誌謝

首先，要感謝的是我的指導教授，楊武博士，老師不斷地指導我去學習，並督促我實作和寫論文，假如沒有老師的鼓勵和指導，學生很難順利完成這篇論文，在此，向老師致上最深的謝意。

接下來，我要感謝的是程式語言及系統實驗室的所有同學，有著大家的陪伴，研究生涯方是多采多姿；累的時候互相鼓勵，快樂的時候一同分享。在我研究、寫論文的過程之中著實提供不少心靈上的慰藉、也讓生活更加充實、有趣。此外，還有滄智常常陪我一同運動、討論程式、分享工作經驗…，沒有他，論文也不會如此順利，真的非常感謝。

最後，要將這篇論文獻給我的家人、女友，如果沒有他們的支持、經濟上的援助，我是不可能取得這個學位、完成自己心願，在此，由衷地感謝他們。

目錄

表目錄

圖目錄

一、緒論

1.1 研究動機與背景

Java，由美國昇陽電腦公司(Sun Microsystems, Inc)於 1995 年 5 月發表，其主要特色有「write once, run anywhere」、刪除指標操作、安全性、增加物件導向設計觀念、大量的類別庫支援…等。而這些特色，促使 Java 成為當今世上最受歡迎、最普及的程式語言之一。

然而，在這些特色之後，所隱藏的是少為人知的系統結構。Java 為什麼能「write once, run anywhere」？其背後的機制為何？為何能確保在各個作業平台皆有相同的執行結果？而這些經過特殊考量、設計的系統存在著什麼樣的優點、什麼樣的缺點？其中又有什麼是我們必須了解、重視的？

在這篇論文之中，我們將專注於 Java 軟體的保護，探討問題的成因及相關保護方法，以保護自身權益、免受他人侵害智慧財產權。

1.2 研究目的

與其說 Java 是一個程式語言，不如說它其實是個系統、平台。在簡單易用、跨平台的背後其實有著許多技術的結合。今日我們所最熟悉者莫過「Java 虛擬機器」，雖然「虛擬機器」的概念並非 Java 首創，然而多數人卻可能是因為 Java 的出現方才了解、注意到什麼是虛擬機器以及它的原理、架構。

為了達成「跨平台」的目的，虛擬機器上採用了兩種架構：一是指令集是與底層平台無關的 bytecode 及避免使用平台特有暫存器的 Stack Machine 架構。而其中 bytecode 這種中間碼(intermediate representation)表示法，採用的是符號式連結(symbolic reference)，以利於虛擬機器執行時的動態繫結(dynamic binding)，而此種方式也代表著原本存於原始碼中的若干資訊經過編譯後仍會存於 bytecode 之中。亦即，如果我們能夠了解 bytecode 的組成格式，就可以了解 bytecode 之中所擁有的資訊，諸如此類別繼承自那個類

別、宣告了什麼欄位、方法…等；說的更具體一些，就是可以將bytecode反編譯為原始碼。而這就是問題的所在，代表著任何人只要對Java有所了解，只要到網路上下載個反編譯器(Decompiler)，就可以輕易地「看到」別人費盡心血所寫的程式，進而從事非法意圖。

今日，Java反編譯器的普及、免費、容易取得，加上優化變數命名、縮排等功能出現，轉換後的程式碼逼近原始碼，甚至有過之而無不及。亦即，軟體開發商、程式設計師若使用Java開發軟體，並發行、散佈於市面、網路上之際就必須了解軟體有被人利用反向工程竊取智慧財產權的危機。

本論文主要探討如何混淆原始碼以期達到「反反編譯」(Anti-decompiler)之目的，期使我們的敵手、或是有任何有不軌企圖之人士無法僅使用反編譯器即可達到他們之意圖，迫使他們必須親自手動進行反編譯，進而感到難以達成、不符經濟效益而主動放棄反編譯。

1.3 章節概要

本論文共分五章，第二章介紹各種現存軟體保護技術。第三章提出一種新的混淆法，介紹其原理、優、缺點及應用方向。第四章介紹實作混淆器及系統的過程。第五章提出結論及未來改進方向。

二、Java 軟體保護技術

一般而言，當一個系統建立之後，在其生命週期之中免不了做些修改，例如錯誤修正、調整使用方式、新增功能…等。然而，有些問題往往會在此時產生—如果當初的原始碼已不可得那該如何修改系統？為此，反向工程(Reverse Engineering)因運而生。

隨著時光流轉，反向工程的使用目的似乎已變成現今眾人所熟知的破解「註冊碼」以利複製販賣謀取利益、竊取系統中具有商業價值之演算法、資料結構或是破解安全相關設施。

傳統上對軟體進行反向工程是一件困難的任務，原因在於傳統語言的編譯方式多與平台系統相依，為求速度之提昇多充份利用平台架構上的特點進行最佳化處理，如使用記憶體位置呼叫副程式、使用暫存器及特殊指令集等。再者，軟體的架構越來越複雜、體積越來越大皆使反向工程之難度大大提昇。

然而，Java 跨平台的優點，卻使其易受反向工程威脅。Java 為了達成跨平台的目的，使用了堆疊指令集(Stack Machine Instruction Set)架構，完全摒除了暫存器、特殊指令集而採用 bytecode 此種中間表示法(Intermediate Representation)之指令集。此表示法有幾項特色：使用一個位元組代表 opcode，也就是最多只能定義 256 種指令，此外為了達成跨平台及物件導向語言特性，使用符號連結(symbolic linking)及動態繫結(dynamic binding)之機制，以符號代替了記憶體位置呼叫副程式。上述這幾種原因，促成了原始碼中的相關資訊必須儲存於編譯過的檔案之中，亦即俗稱的類別檔(class file)。

類別檔本身的組成有一定的格式[1-3]，簡單而言就是一個位元組串流，只要了解其組成，就可以輕易地達成反編譯之目的，是以 Java 發表僅僅一年之後就有第一個反編譯器 Mocha[4]的出現。今日，出現了更多更快更好的反編譯器，不過相對的，也產生了許多為了抗衡的保護技術。

軟體保護技術的目的在於要讓反向工程變得不切實際，逼使懷有不良意圖之人必須花費很高的代價(時間或是金錢)才可以達成其目的；也就是說要盡可能的拖延攻擊的行為，讓他感到灰心、挫折，進而萌生放棄之念頭。

2.1 軟體保護技術之分類

當 Java 易遭反向工程的弱點被揭露出之後，許多研究紛紛出現，為了研究上的方便，以下先針對相關軟體保護技術予以分類再加以討論。

為了方便討論，此處將保護技術分成下列四大類：

1. 加密(Encryption)
2. (Partial) Server-side Execution
3. 混淆(Obfuscation)
4. 其它

其中混淆(Obfuscation)一項又可以予以細分，在接下來的各小節之中，將予以細部的討論。

2.1.1 加密(Encryption)保護技術

如圖 2.1 所示，將程式加密後再傳送給使用者執行，然而依目前的狀況而言，難以實行，一來解密會花費許多時間，降低效能，此外將解密的金鑰隱藏於軟體之中易遭破解。除非未來能以硬體實作全部的加/解密執行過程或搭配其它更安全的保護金鑰措施，不然此技術的運用將會受到一定之限制。

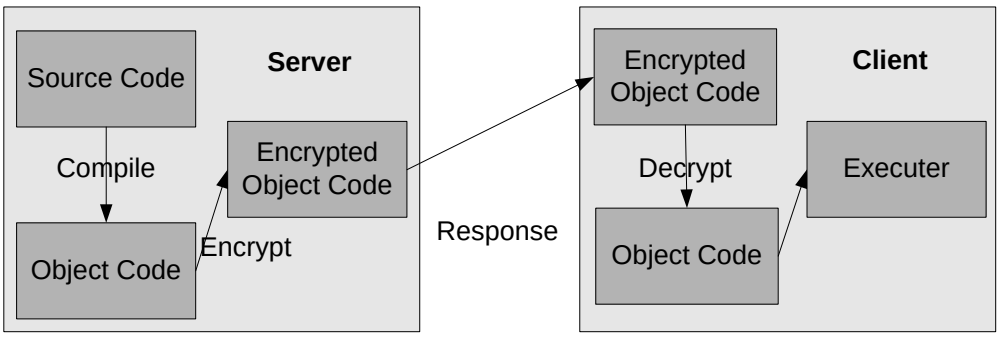


圖 2.1 加密保護技術

2.1.2 (Fully/Partial) Server-side Execution

將全部(或是部份)的程式放於伺服端，再依使用者之請求執行，執行完成後

再將結果傳回使用者，如圖 2.2、2.3 所示：

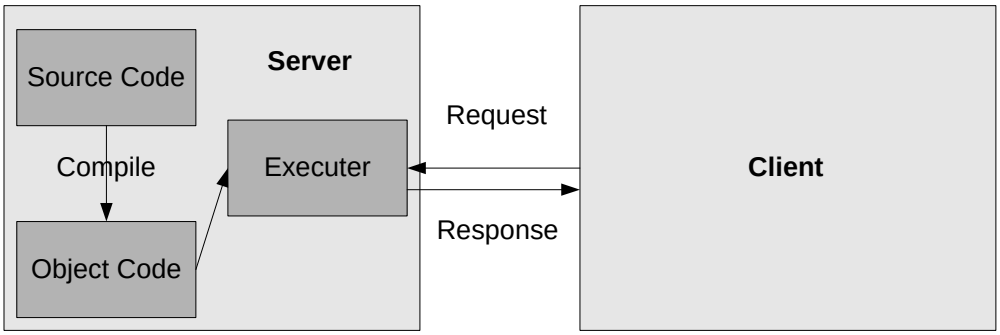


圖 2.2 Fully Server-side Execution

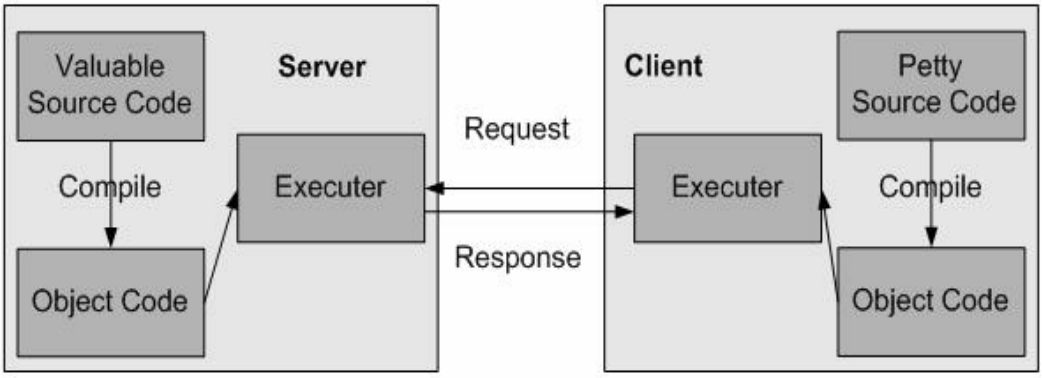


圖 2.3 Partial Server-side Execution

此外，Partial Server-side Execution 亦可結合電子簽章予以進階的保護。

當未來網路頻寬、安全性等相關措施進一步成長時，此一保護方式有可能變成主流應用[5]。此一趨勢就如同往日的單機版電玩遊戲因不堪「破解」之擾，而紛紛轉型為線上遊戲一般。

2.1.3 混淆(Obfuscation)[6]

混淆的觀念大致上可以敘述如下：轉換程式，使得它仍維持與原程式相同的功能、行為，但是卻更難以了解、反編譯。

目前已經有不少的混淆技術被提出，有的已完成實作，有的僅止於觀念上的提出，接下來幾個小節將介紹幾個較具重要性的技術。

2.1.3.1 識別子重新命名(Identifier rename)[6]

識別子重新命名是最早應用於 Java 的保護方法之一，原理是將程式之中的識別子，如類別名稱、方法名稱、欄位名稱及區域變數名稱…等，改變成無意義的名稱，如：

類別名稱：MainApp → A

方法名稱：showGraph() → b()

欄位名稱：birthday → c

如此一來，即使程式被反編譯了，攻擊者也難以明瞭原本識別子所代表的意義，增加反編譯的難度。

此外，在本實驗室陳建財學長的研究[7]之中，將此技術提昇至另一個境界，其概念可以圖 2.4 表示：

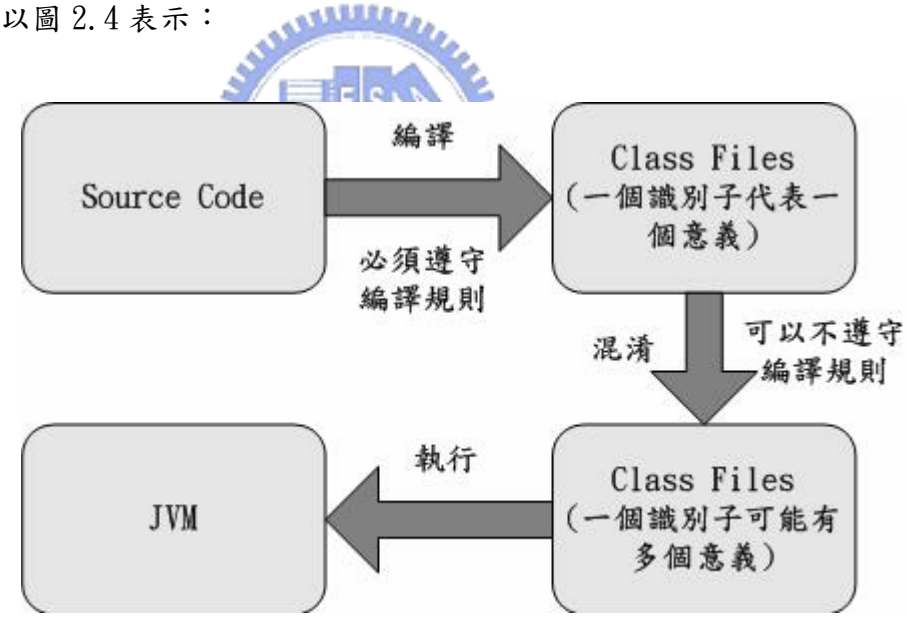


圖 2.4 進階識別子重新命名的基礎概念

基本的概念就是盡可能重複使用識別子，讓一個識別子盡可能代表多重意義，藉此混淆反編譯器，促使其當機或是產生無法重新編譯的原始碼，逼使攻擊者必須手動除錯、消耗大量時間、心力。此外使用此保護方式可能會帶來另一項優點：當識別子重複用之時，壓縮檔案(通常是 jar 檔)的效果會更好，會進一步減少程式碼所需空間。

2.1.3.2 移除註解(Remove comment[6])

在 Java 類別檔之中存有許多除錯資訊(如 LineNumberTable、LocalVariableTable)，不過這些資訊對於執行而言並非必要，可以除去讓反編譯器、攻擊者讓難以達成反編譯的目的。

2.1.3.3 改變編碼(Change encoding[6])

如圖 2.5 所示：左方的整數 i 原本為 1 經過轉換後成為右方的 13，在 while 迴圈之中的存取指標亦做相對應之調整，當攻擊者看到右方的程式碼，勢必要多花一些時間於解析迴圈的存取規則。

```
int i =1;                                  int i =13;
while(i< 10){                              while(i < 103){
… A[i]…;                    ⟶              … A[(i-3)/10]…;
i++;                                       i+=10;
}                                          }
```

圖 2.5 改變編碼範例

2.1.3.4 Modify object-oriented relation[9]

Java 是個物件導向語言，在開發系統時往往在架構的設計、撰碼階段使用一些著名的設計模式[10]，以保持開發時的模組化及後續修改的彈性，雖然這麼做對程式設計師而言有其正面價值，然而相對而言當對手認出了架構、設計模式之時，就會降低反編譯的難度，因此，此方法利用混淆物件導向的特性來增加程式的複雜度，例如增加類別之間的繼承深度、類別分裂(class splitting)、類別融合(class coalescing)、型別隱藏(type hiding)…等。總之，只要能增加程式複雜度的方法都值得一試，不過，使用之前需要考量諸如程式碼增加大小、降低執行速度等代價。

2.1.3.5 控制流程混淆(Control Flow Obfuscation)[11]

此方法主要建立於 opaque constructs(曖昧不明的結構)之上。在不影響程式執行結果的條件下插入不易預測運算結果的分支指令來達成混淆之效果。如圖

2.6 所示，在敘述 $S_1S_2\cdots Sn$ 之間插入分支指令 P，其分支結果難以預測，但實際上為 true。

當整個程式大量地插入 opaque constructs 之後，攻擊者勢必要花費更多的時間去了瞭、分析該流程所代表的意義。想要從既有的程式碼之中擷取出演算法就將會變得更難、更複雜。

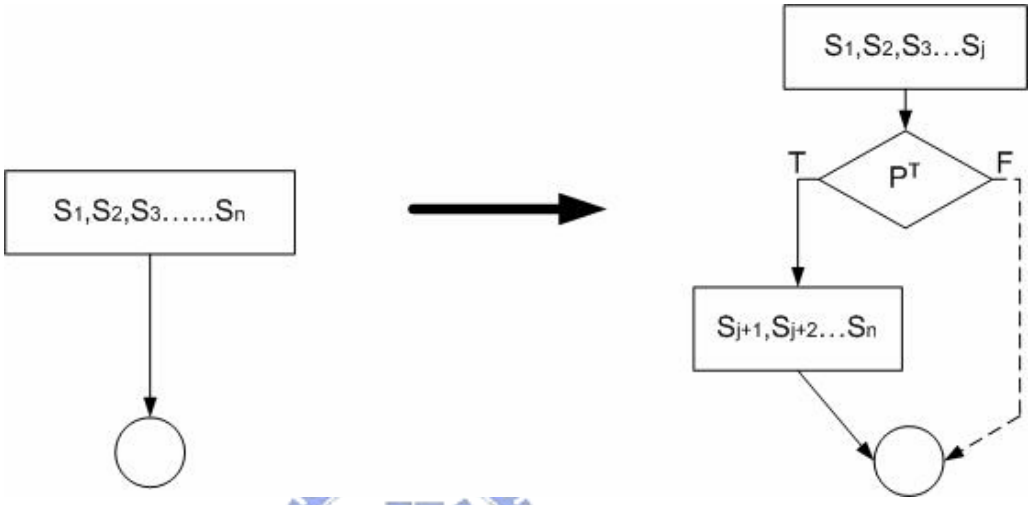


圖 2.6 控制流程混淆示意圖

2.1.3.6 置換 goto 混淆法[12]

Zelix[13]是一套商業的混淆器，除了識別子重新命名、字串加密外，最大的特色在於其特有的流程混淆(Flow Obfuscation)，經由觀察混淆過的程式碼之後，可以發現其使用了一種技術專門破壞迴圈結構，使用的原理在將迴圈開頭的 goto 指令置換成條件分支指令，而置換過的執行路徑仍保持原有程式的執行路徑。

不過 Zelix 這套混淆器對於以控制流程圖(Control Flow Graph)為建構基礎的反編譯器效果不彰，在蔡明修[12]的研究之中有對置換 goto 混淆法作進一步的改良，有興趣者可自行參閱、研究。

2.1.4 其它相關保護技術

除了上述的保護技術之外，事實上還有許多相關技術可供利用，在 Christian

Collberg 及 Douglas Low 眾人的研究[15-20]之中有許多的介紹、分類；此外，亦有人[21]提供了一些手動混淆的建議事項。凡是能增加軟體複雜度的方法，本質上都值得一試。當然，使用的代價必須列入考量，不然在還沒有防到他人反編譯之前可能就會先累壞自己。

2.2 反混淆(Deobfuscation)技術

當保護技術不停演進之際，反向陣營的人也努力提昇反混淆的技術，比起第一個反編譯器 Mocha 而言，現在的反編譯器可以說是種類更多、發展速度快、功能更強。往往一個新的保護技術被提出來沒多久就有相對應的破解技術因應而生。

早期的反編譯器只針對 Sun 的編譯器(javac)所產生的類別檔利用樣式比對(Pattern Matching)的技術進行反編譯，但是當各家軟體廠商推出自己的編譯器之時，樣式比對的方法往往成效不彰；因此，許多反編譯技術陸陸續續地被研發出來，以期提升反編譯能力。今日已有反編譯器使用控制流程圖分析(Control Flow Graph Analysis)的方式進行反編譯，致使一些保護技術遭受破解。在研究、發展保護技術之際，有必要注意、了解反編譯技術的進展、建構原理才能針對其弱點進行攻擊進而保護智慧財產權。

在「Of Graphs and Grounds: Decompiling Java」[23]這份技術文件之中，作者使用 Soot framework[24]為基礎進行 Java 程式反編譯研究。Soot framework 定義了三種不同的中間碼表示法:Baf、Jimple、Grimp。由於 Grimp 相當接近 Java，因此很適合作為反編譯器的研究基礎。在這份報告之中作者藉由建立控制流程(Control-flow)的圖形再透過三個時期的化簡圖形程序，最後得到 Java 原始碼。整個過程圖 2.7 所示。

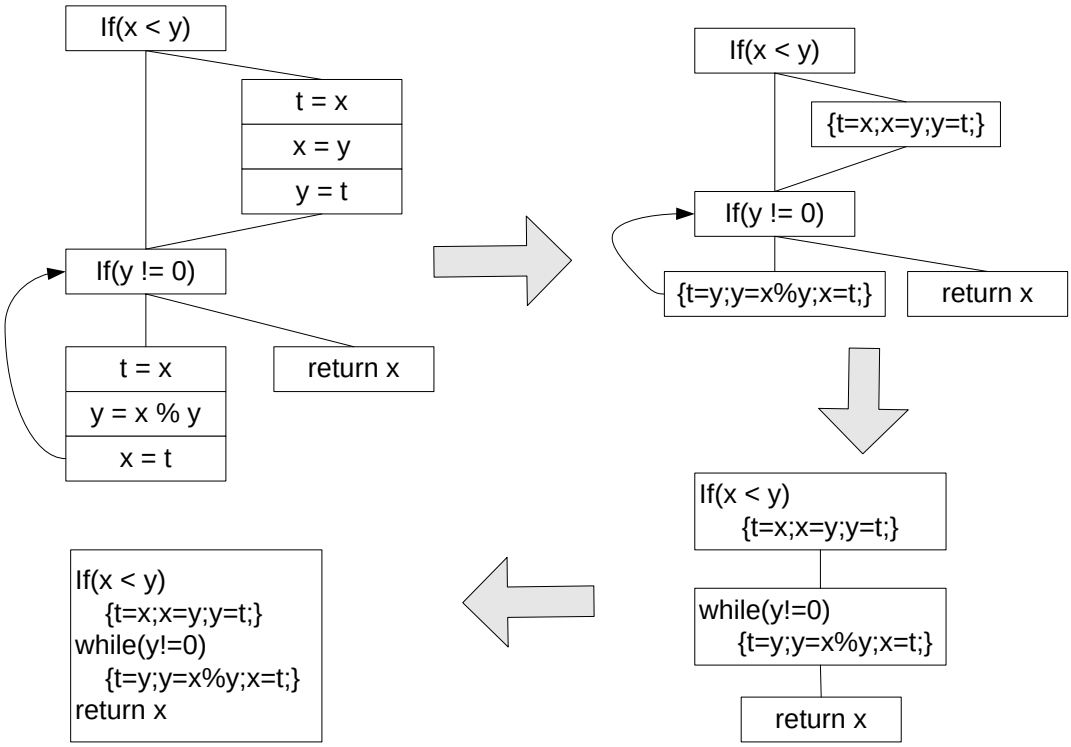


圖 2.7 控制流程圖化簡過程(資料來源：[23])

Dava decompiler[25]也使用 Soot framework 作為反編譯器的建構基礎，不過建構技術比上述技術更進一步，使用了六個階段的化簡圖形程序[26]。

2.3 評估準則

在介紹了許多軟體保護技術之後，有個議題其實值得我們一同思考：什麼樣的保護技術算好？不同的技術之間如何衡量、比較其優、缺點？很可惜，到目前為止並不存在一個好的方式能加以比較或是證明該保護技術能有效保護軟體。在「On the (im)possibility of obfuscating programs」[22]一文中，作者明確地指出，目前為止的保護技術在理論及實際應用上皆沒有足夠的證據可以證明能有效地保護軟體。亦即，「有法有破」，其實每種保護技術都有其致命的缺點，不可能存有一個完美的保護技術。

換個角度思考，如果不可能存在一個完美的技術，那就既有技術加以組合輔以一些新的技術成效會是如何？縱使還是有法可破，但是否有可能建構一個更複雜、難以破解的系統？

2.4 本章小結

反編譯技術與保護技術就像冷戰時期的軍備競賽一般，不停地相互競爭、找尋對方的弱點。然而，就保護技術而言，往往為了使用者的方便或是成本考量，不得不捨去一些保護性較佳的技術。如果有朝一日現今的保護技術不足以抗衡反編譯技術，則軟體保護技術勢必有一番重大革命，可能是更加利用硬體或是只提供服務而不提供軟體。

本章介紹了一些既有的保護方法，在下一章中，將會探討本論文所設計的保護方式。

三、以定性的有限狀態機為基礎的 Java 程式混淆之設計

3.1 軟體混淆之基本概念

所謂軟體混淆(Software Obfuscation)是指將原程式做轉換使其在保持原功能的狀態下，使人、工具更加難以了解其結構、語意。然而，不論其如何轉換，只要有足夠的時間、決心，混淆過的程式終將會因仍保有一切程式執行所需的資訊而被破解。

由此可以了解，其實我們所能做的並非完全的防堵惡意的攻擊者，而是一種「拖延」。如果一個程式沒有經過任何混淆，只須幾秒鐘，攻擊者就可以藉由反編譯器輕鬆獲得原始碼，進而從事非法意圖。但是如果一個程式經過高度混淆，讓反編譯器無法執行反編譯，或是執行後所得的原始碼與真正原始碼有很大的差異，無法再次編譯，那唯一之途就只剩手工打造，逐一還原；而這無疑是費時費力、折磨人的苦差事，也意味著破解之日遙遙無期，想破解就必須投入大量的時間、心力，進而迫使攻擊者感到不值得、灰心、氣餒進而放棄反編譯。

而一個好的混淆器應該具有什麼樣的特性呢？從眾多的研究[7]之中大致可以歸納出如下的要求：

1. 保持原程式的功能。
2. 讓攻擊者難以設計出自動化的工具。
3. 儘可能延遲反向工程所需的時間。
4. 保留或改善原程式的效率且以不增加過多的程式大小為限。

3.2 Java 軟體保護的設計基礎

在進行 Java 混淆器的設計之前，需要先行了解一些相關重要議題，以下小節將一一介紹。

3.2.1 類別檔的格式

類別檔的格式對編譯器、Java 虛擬機器(往後皆稱 JVM)、混淆器及反編譯器皆是一項很重要的議題，其重要性可以藉由圖 3.1 顯示。

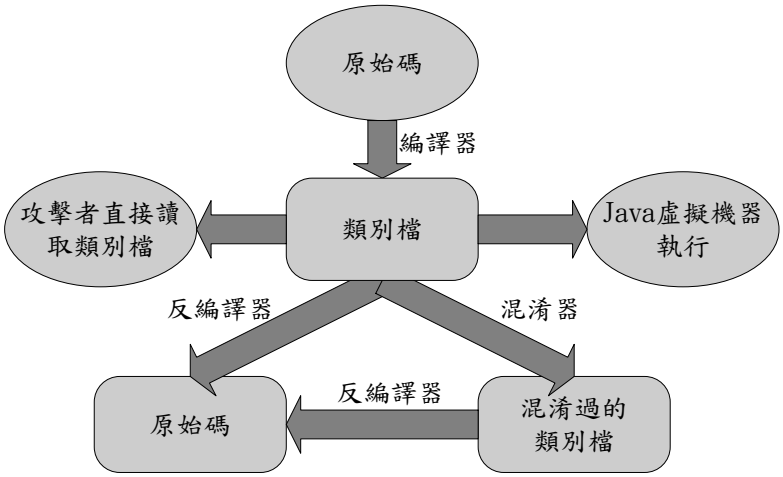


圖 3.1 類別檔之重要性

無論是混淆器或是反編譯器首先要處理的皆是類別檔，所以在深入探討混淆器之前有必要先了解類別檔的格式。

Java 類別檔是 Java 程式編譯後所產生的二元碼檔案，依據 JVM 格規書規定，每個類別檔只包含一個 Java 類別或是介面的定義。檔案以魔術數字「0xCAFEBABE」開始，依續為版本編號、常數區、類別存取權限…等，表 3-1 為類別檔的結構說明，在此處僅就較重要者加以說明，其它相關資訊請參閱 JVM 規格書。

表 3.1 類別檔結構

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
```

```
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

<1>常數區(constant_pool)：儲存此類別用到的符號和原詞(Literal)，包含字串、數字常數、類別指引、方法名稱、欄位名稱及型態描述子。

<2>方法數目(methods_count)：此類別所擁有方法的數目。

<3>方法群(methods)：此類別擁有的方法，不包含父類別的方法。每個項目儲存指向常數區的某項目的索引值，記錄方法名稱，其結構如下所示：

表 3.2 method_info 的結構

```
method_info {
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

<4>屬性(attributes)：在 Java 類別檔之中使用了多種屬性，包含 SourceFile、ConstantValue、Code 及 Exception…等。在本篇論文之中，最關心者當屬 Code 及 Exception 這兩個屬性，以下分別介紹之。

<A>Code Attribute：

如果屬性的 attribute_name_index 指向的名稱為 Code 的話，則此屬性為 Code Attribute。Code Attribute 被用來描述某個方法(method)，包含了該方法的 bytecode 及一些執行時所必須知道的相關資訊。其中的 code[code_length]陣列儲存的就是此方法的指令串流。

表 3.3 Code Attribute 的結構

```
Code_attribute {
            u2 attribute_name_index;
            u4 attribute_length;
            u2 max_stack;
            u2 max_locals;
            u4 code_length;
            u1 code[code_length];
            u2 exception_table_length;
            {       u2 start_pc;
                    u2 end_pc;
                    u2 handler_pc;
                    u2 catch_type;
            }    exception_table[exception_table_length];
            u2 attributes_count;
            attribute_info attributes[attributes_count];
    }
```

其中，exception_table 的資料結構如下所示：

表 3.4 exception_table 的結構

```
exception_table[exception_table_length]{
    u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
}
```

當 code[code_length]有經過調整時，exception_table 就需要重新調整。

<B>Exceptions Attribute：

指出所屬的 method_info 結構代表的 method 會丟出那一種 checked exception。

表 3.5 Exceptions_attribute 的結構

```
Exceptions_attribute {
        u2 attribute_name_index;
        u4 attribute_length;
        u2 number_of_exceptions;
        u2 exception_index_table[number_of_exceptions];
    }
```

3.2.2 指令的分類

在解析類別檔的時候，指令的種類、長度扮演一個非常重要的角色，如果誤判了其中一個指令，後續指令的解析也會接連出錯，這個特性除了對混淆器分析類別檔非常重要之外，也可以利用來使反編譯器判斷錯誤，在後面的章節之中會再利用到這個特性來對抗反編譯器。

一般而言一道 Java 指令可以下列方式表示：

opcode + [operand…]

在 JVM 規格書中定義了 202 個 opcode，並保留了三個 opcode，其中除了 tableswitch 及 lookupswitch 兩者的指令長度須搭配後續的運算元(operand)計算外，其餘的指令長度皆可由 opcode 得知。

除了指令的長度之外，亦可先依指令的特性分類再做後續的處理，附錄一中有本論文採用的分類方式，有興趣者可自行參閱。

3.3 以定性的有限狀態機為基礎的混淆設計原理

了解了第二章所提的相關保護方法之後，相信大家對混淆已經有了一些基本概念，在接下來的這一小節之中，將會探討本論文所提「以定性的有限狀態機為基礎的混淆」(DFA-Based Obfuscation，後續皆以英文表示)設計原理。

3.3.1 設計基礎

在本論文之中，所設計的 DFA-Based Obfuscation，主要的概念來自於「randomized instruction-set approach」[28]及有限狀態機(Finite State Machine)[29]。簡單地說就是改變 opcode 與語意之間的對應關係，讓攻擊者或是反編譯器誤認或是難以判定。

在 JVM 規格書之中定義了 202 個 opcode，每一個都有其對應的語意，當執行時，直譯器(Interpreter)就依照事先定義的語意執行相關的原生程式碼。不過這種對應可以適當的加以改變、運用，例如 bytecode 之中的 iadd 可以做如圖 3.2 之轉換：

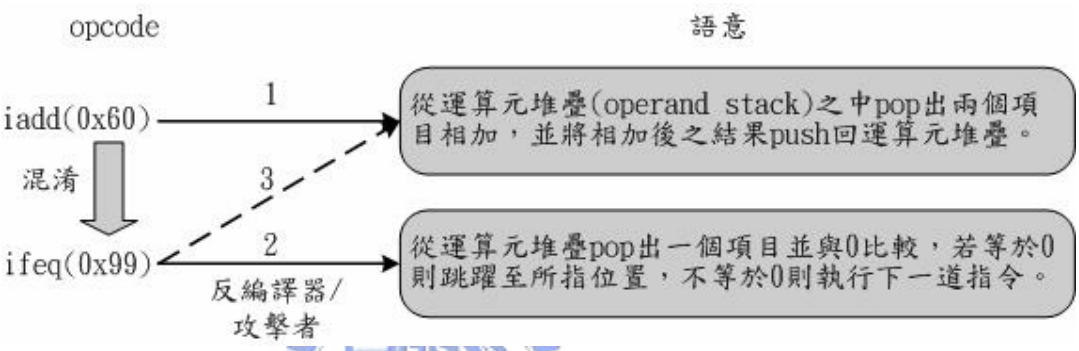


圖 3.2 指令轉換範例

原本「iadd」的語意為：從運算元堆疊(operand stack)之中 pop 出兩個項目相加，並將相加後之結果 push 回運算元堆疊。今故意讓「ifeq」也代表相同的語意，則當反編譯器或是攻擊者看到此指令之時，會認為它是代表定義在 JVM 規格書中的「ifeq」這道指令，進而將該 opcode 的下兩個位元組當做 offset 處理，亦即，原本的指令被誤認、錯誤處理了，結果不是反編譯器處理錯誤、當機就是攻擊者抓狂、失心耐心。此外，就算知道了這是混淆過的指令，反編譯器及攻擊者也難以找出其中的轉換規則。

整體的概念可以藉由圖 3.3 及圖 3.4 之比較獲得更深入的了解。圖 3.3 代表的是一般的 Java 程式執行流程，圖 3.4 則是本論文所提保護方法的執行流程。

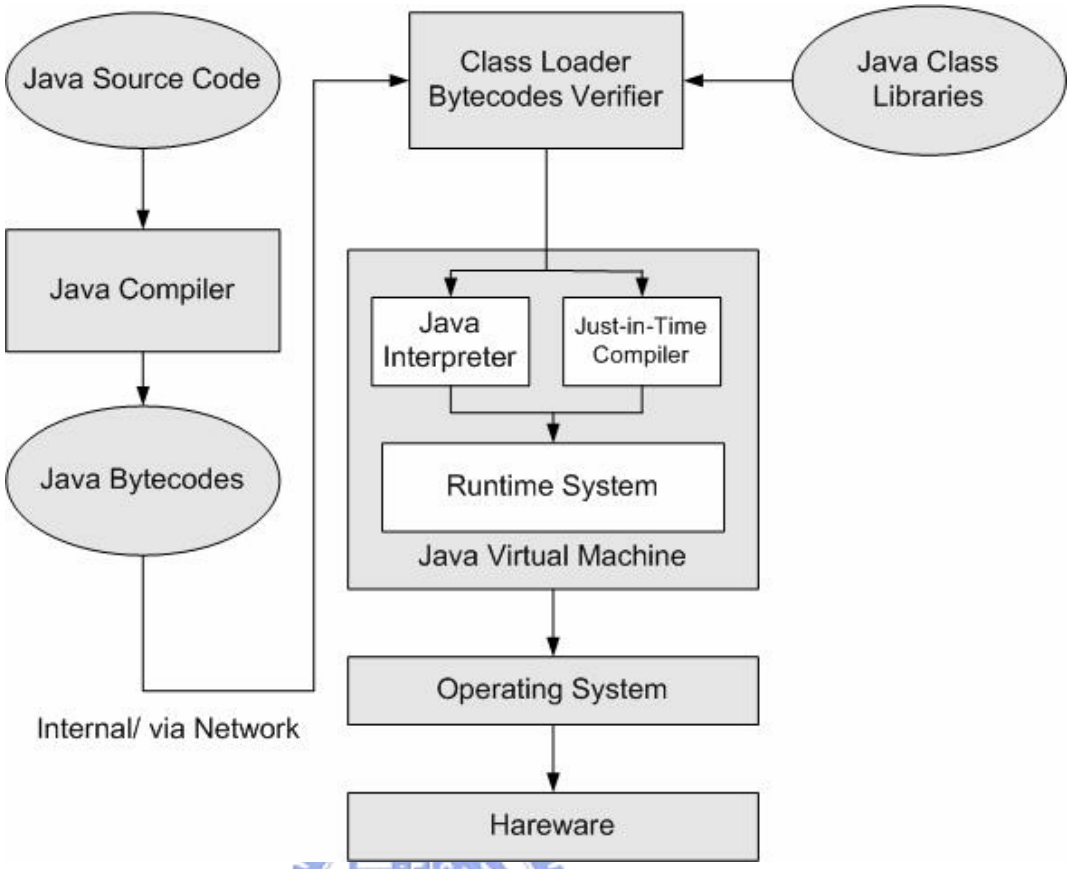


圖 3.3 一般的 Java 運作流程

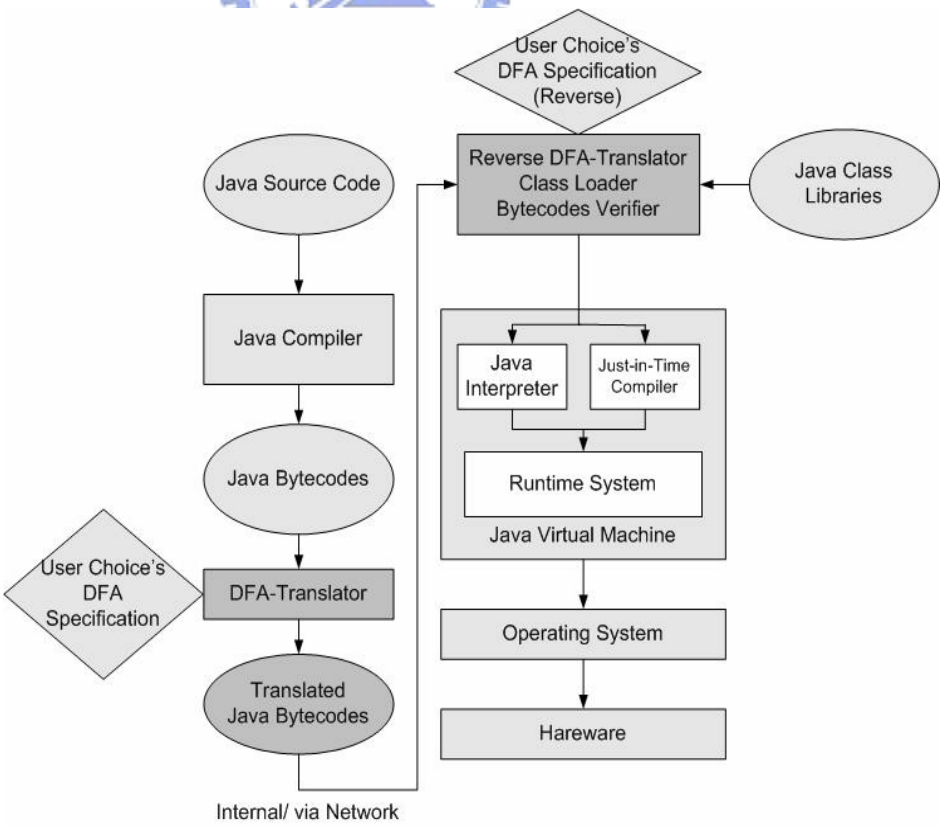


圖 3.4 DFA-Based Obfuscation 的 Java 執行流程圖

經由比較可以得知本論文所提的保護方法較原有的執行流程多了幾個元件：

1. User Choice's DFA Specification
2. DFA-Translator
3. Reverse User Choice's DFA Specification
4. Reverse DFA-Translator

關於這些元件的詳細功能將在接下來的章節之中加以介紹。

3.3.2 設計原理

如同在上一小節所提，本方法的觀念主要來自有限狀態機(Finite State Machine)及「randomized instruction-set approach」的結合以達到混淆程式之目的。其最大的特色在於指令之間的對應關係不是靜態而是動態，除了輸入(Input)之外，尚須以前一道指令所達狀態(State)，共同決定輸出(Output)及下一個狀態(State)。

一般的有限狀態機可用 5 個 tuple 加以描述[29]：

$(Q, \sum, q_0, A, \delta)$，其中：
Q：finite set
$\sum$ ： finite alphabet of input symbols
$q_0 \in Q$：(the initial state)
$A \subseteq Q$：(the set of accepting states)
δ ：is a function from $Q \times \sum \to Q$ (the transition function)

以此為基礎再做進一步的延伸，可以建構出如下的有限狀態機：

$(Q, \sum, \Psi, q_0, \delta, \lambda)$，其中：
Q、$\sum$ 、q_0及δ四個項目定義同上
Ψ：finite alphabet of output symbols
λ：is a function from $Q \times \sum \to \Psi$(the output function)

相較兩者之差異在於後者多了 output alphabet 及 output function 這兩個 tuple，而沒有 accepting states 這個 tuple。除此之外，尚有下列限制：

1. 僅考慮定性的有限狀態機(Deterministic Finite Automata，後續以 DFA 稱之)
2. $Q \equiv \Psi$
3. 先執行 $Q \times \sum \to \Psi$ 再運用 $Q \times \sum \to Q$(使用同一個 input symbol)

了解了有限狀態機的建構之後，接著看一個簡單的範例，如圖 3.5 所示，其中：

$$Q = \{q_0, q_1, q_2\}$$

$$\sum = \{iadd, isub, imul, idiv\}$$

$$\Psi = \{\underline{iadd}, \underline{isub}, \underline{imul}, \underline{idiv}\}$$

$$\delta_0(iadd) = q_0, \delta_0(isub) = q_1, \delta_0(imul) = q_1, \delta_0(idiv) = q_2$$
$$\delta_1(iadd) = q_1, \delta_1(isub) = q_1, \delta_1(imul) = q_2, \delta_1(idiv) = q_0$$
$$\delta_2(iadd) = q_1, \delta_2(isub) = q_0, \delta_2(imul) = q_0, \delta_2(idiv) = q_2$$

$$\lambda_0(iadd) = \underline{isub}, \lambda_0(isub) = \underline{imul}, \lambda_0(imul) = \underline{idiv}, \lambda_0(idiv) = \underline{iadd}$$
$$\lambda_1(iadd) = \underline{idiv}, \lambda_1(isub) = \underline{iadd}, \lambda_1(imul) = \underline{isub}, \lambda_1(idiv) = \underline{imul}$$
$$\lambda_2(iadd) = \underline{iadd}, \lambda_2(isub) = \underline{idiv}, \lambda_2(imul) = \underline{imul}, \lambda_2(idiv) = \underline{isub}$$

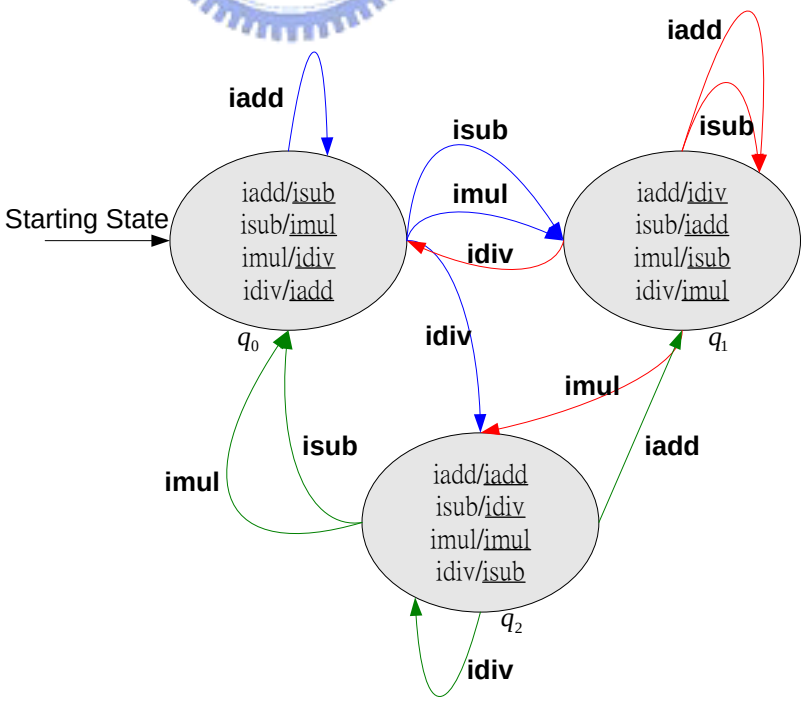


圖 3.5 範例所採用的 DFA

假設有一指令串流如表 3.6 左方所示，則經過圖 3.5 所示 DFA 轉換過可得右方之指令串流。首先，iadd 由 Starting State 出發，在q_0時 output 為 isub，state 轉換後依舊為q_0，處理完第一個指令之後接著處理第二道指令，imul 在q_0的 output 為 idiv，state 轉換後為q_1；第三道指令 isub 在q_1的 output 為 iadd，state 轉換後為q_1；第四道指令 imul 在q_1的 output 為 isub，state 轉換後為q_2；第五道指令在q_2的 output 為 iadd，state 轉換後為q_1。

表 3.6 範例一的指令串流轉換

1	iadd x, y		1	isub x, y
2	imul x, y	⟶	2	idiv x, y
3	isub x, 1		3	iadd x, 1
4	imul x, 2		4	isub x, 2
5	iadd x, 2		5	iadd x, 2

從上表之中我們可以觀察到第一道、第五道指令皆是 iadd，但是轉換之後的結果分別是 isub 及 iadd。而二、四道指令原本皆是 imul，經過轉換之後分別變為 idiv 及 isub。由此可以得知以 DFA 為基礎的轉換，其轉換之間的對應關係並非一對一的靜態關係、而是動態，依之前指令所達狀態加現在的指令所得之結果。而當反編譯器或是攻擊者看到右方的指令串流時就會產生困惑，無法依照一般 bytecode 的指令格式進行反編譯。

經過了一個簡單的範例之後，接著將探討如何建構 DFA-Based Obfuscation 以供軟體保護之用。

在 JVM 規格書之中定義了 202 道指令，如果我們將全部的指令皆放進 DFA 之中予以混淆，雖然混淆的程度達到了最大，不過轉換時間、所佔空間皆會以倍數成長，因此有必要先將指令分類再做後續處理，在此，可將指令分成兩大類：

1. 需混淆的指令($\sum$)：程式轉換時遇到此類指令需以 DFA 作轉換。
2. 不需混淆的指令(NOB)：遇到此類指令不作任何處理。

有了分類之後，後面就可以用來設計轉換器。

3.3.3 設計原理之運用

了解了 DFA 的模型之後，在此先將相關符號予以定義以利後續討論，如圖 3.6 所示：

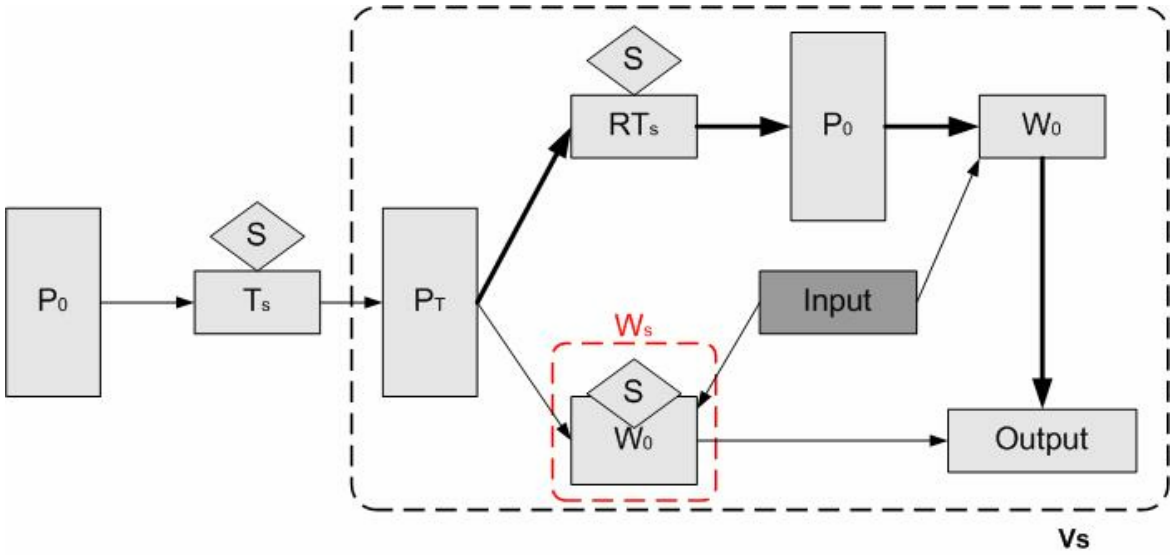


圖 3.6 DFA-Based Obfuscation 的符號定義

P_0：原始程式，亦即要保護的程式。

S：DFA 的轉換規格，定義了需混淆指令與其語意之間的動態對應。

T_S：使用轉換規格 S 的程式轉換器(Translator)，目的是將 P_0 轉成 P_T。

P_T：轉換過的程式，亦即一般使用者所能獲得的程式。

RT_S：反轉換器，目的是將 P_T 轉成 P_0。

W_0：一般 VM 之中的直譯器。

W_S：使用轉換規格 S 建構的直譯器。

Input：程式的輸入。

Output：程式的輸出。

V_S：Java 虛擬機器。

圖中 P_0 經過轉換為 P_T 之後，有兩種方式可以執行此轉換過的程式：

1. 在虛擬機器的類別載入(Class loading)過程之中將利用 RT_S 將 P_T 轉換回 P_0，以供直譯器執行程式的使用。
2. 建構一個 DFA-Based 的直譯器，其功能是直接讀取已轉換過的指令執行程式，在執行過程中以 S 所定義的語意對應關係做為執行依據。

上述這兩種建構方式，在本質上並無重大差異，皆是利用轉換規格所定義的指令與語意之間的對應關係來執行程式，至於細部的差異將在下一章中予以討論。

3.3.4 轉換器的演算法

經過了上一小節的介紹之後，緊接著我們將探討轉換器的建構原理，如同以往，先介紹相關符號定義，如圖 3.7 所示，其中：

PC：程式 P_0 的行號(Line number)
$InstP_0(PC)$：P_0 中 PC 處的指令
$InstP_T(PC)$：P_T 中 PC 處的指令
q_s： DFA 目前的 state
STATE(PC)：當 $InstP_0(PC)$ 這道指令被轉換後的 state

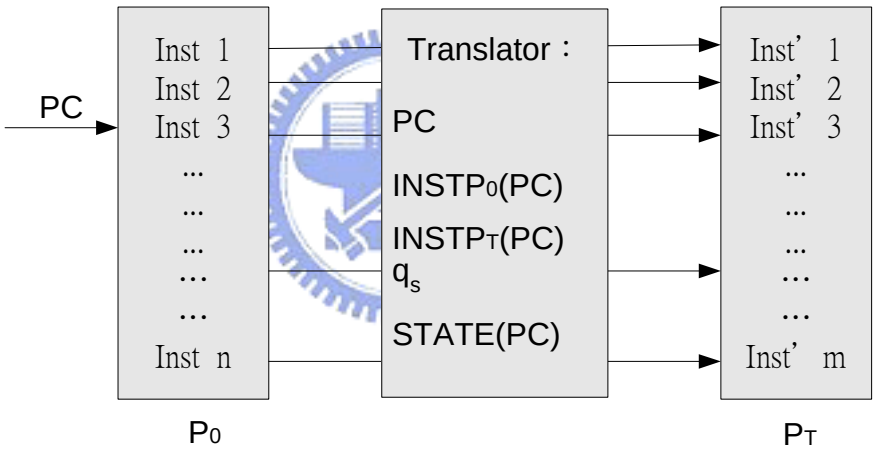


圖 3.7 轉換器的建構示意圖

有了這些定義之後，接著我們先來看一個最簡單的轉換器演算法(僅適用於以類別載入為執行基礎的方式，如需 DFA-Based 直譯器者，請參閱[30])。如圖 3.8 所示，先將類別檔之中的 method 結構解析出來，使之成為指令串流(如圖 3.7 左方所示)，其次設定初始參數，接著就是進入迴圈依序處理每道指令，判別是否為需要混淆的指令，若是，則以 DFA 作轉換，若否，則不進行任何處理。至於圖中所示的 Change Operand 一項亦是使用相同原理予以建構，將在第四章予以更詳細的說明。

至於反轉換器(RTs)的建構，本質上是與轉換器(Ts)一樣，此處需加以注意者

如下所示：

1. 僅考慮定性的有限狀態機(Deterministic Finite Automata，後續以 DFA 稱之)
2. 先執行 $Q\times\sum\rightarrow\Psi$ 再運用 $Q\times(Q\times\sum\rightarrow\Psi)\rightarrow Q$

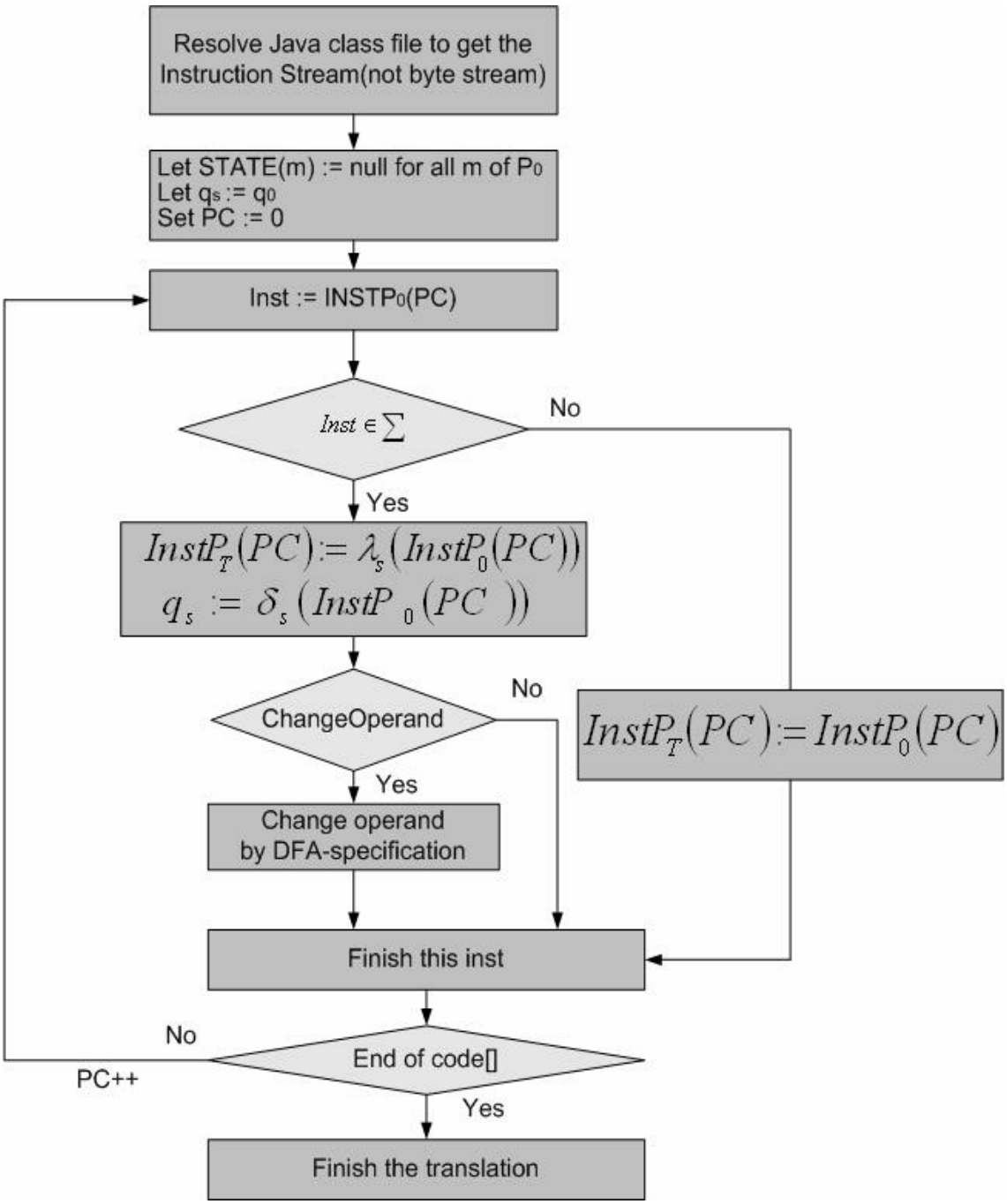


圖 3.8 轉換器演算法

3.3.5 轉換器範例

在這一小節之中，我們以一個簡單的程式做為範例。原始程式如表 3.7 所示，依照整數參數的特性輸出相對應的字串(無意義的字串)。表 3.8 所示為原始程式編譯過後的 bytecode，表 3.9 所示是轉換器依表 3.10 所列轉換規格轉換後的 bytecode。

比較表 3.8 與表 3.9 所列 bytecode，可以得知轉換後的 bytecode 雜亂無序，即使反編譯器、攻擊者是如何有能力、經驗，也難以僅憑表 3.9 所列的 bytecode，反編譯回原來的原始碼。

觀察表 3.8 及 3.9 所列指令，可得如下要點：

1. 表 3.9 的指令數目較表 3.8 為多，不過 printMessage 這個方法所佔的位元組數(code length)依舊為 55(0 至 54)。
2. 表 3.9 之中出現了表 3.8 之中沒有的指令，例如：nop、dconst_0。
3. 表 3.8 之中有些分支指令在轉換之後被取代了，亦即解析表 3.9 的 bytecode 無法得出原來的流程。

緊接著我們來分析上列三項代表什麼意義。首先，第一項指令數目變多但總長度不變度代表有指令被錯誤解析，以表 3.8 之中第三道指令為例：原本為「ifne 29」，佔三個位元組(0x9a 0x00 0x1a)，但經過轉換器轉後過後，ifne(0x9a)變為 iconst_1(0x04)，但 iconst_1 是一個不需 operand 的 opcode，因此原本 ifne 後的第一個位元組現在依其數值「0x00」被解析為「nop」這個 opcode，而 nop 亦是一個不需 operand 的 opcode，因此，原本 ifne 後的第二個位元組「0x1a」亦被當成指令的開頭解析，解析後所得為「iload_0」這道指令。而在這種錯誤解析的過程之中，往往會因將 operand 當成指令的開頭解析因而產生了原來沒有的指令，例如剛剛講解的「nop」就是箇中代表。

而第三項代表轉換過後的程式擁有類似流程混淆的功用，有可能將原本的分支指令變成非分支指令或是將原本的非分支指令變成分支指令。

表 3.7 轉換器範例：原始程式碼

```
    private static void printMessage(int i){
        if(i%2 == 0){
            for(int j = 0; j < 2*i; j++){
                System.out.println("Hello!");
            }
        }else{
            if(i%3==0)
                System.out.println("Hi!");
            else
                System.out.println("No!");
        }
    }
```

表 3.8 轉換器範例：原始程式的 bytecode

```
0 iload_0
1 iconst_2
2 irem
3 ifne 29 (+26)
6 iconst_0
7 istore_1
8 iload_1
9 iconst_2
10 iload_0
11 imul
12 if_icmpge 54 (+42)
15 getstatic #2 <java/lang/System.out>
18 ldc #3 <Hello!>
20 invokevirtual #4 <java/io/PrintStream.println>
23 iinc 1 by 1
26 goto 8 (-18)
29 iload_0
```

```
30 iconst_3
31 irem
32 ifne 46 (+14)
35 getstatic #2 <java/lang/System.out>
38 ldc #5 <Hi!>
40 invokevirtual #4 <java/io/PrintStream.println>
43 goto 54 (+11)
46 getstatic #2 <java/lang/System.out>
49 ldc #6 <No!>
51 invokevirtual #4 <java/io/PrintStream.println>
54 return
```

表 3.9 轉換器範例：經過轉換後的 bytecode

```
0 iload_0
1 iconst_2
2 imul
3 iconst_1
4 nop
5 iload_0
6 iconst_0
7 istore_1
8 iconst_1
9 iconst_2
10 iload_0
11 iconst_1
12 if_icmpge 54 (+42)
15 getstatic #2 <java/lang/System.out>
18 ldc #3 <Hello!>
20 ifne 24 (+4)
23 iinc 1 by 1
26 goto 8 (-18)
29 iload_0
30 iconst_3
```

```
31 iload_1
32 iconst_1
33 nop
34 dconst_0
35 getstatic #2 <java/lang/System.out>
38 ldc #5 <Hi!>
40 imul
41 nop
42 iconst_1
43 if_icmpge 54 (+11)
46 getstatic #2 <java/lang/System.out>
49 ldc #6 <No!>
51 ifne 55 (+4)
54 return
```

表 3.10 轉換器範例：轉換規格(S)

State	Input/output	transition	State	Input/output	transition
0	goto/ if_icmpge	4	4	goto/ if_icmpge	5
	if_icmpge/ goto	7		if_icmpge/ goto	4
	iload_1/ ifne	5		iload_1/ invokevirtual	2
	iconst_1/ iload_1	2		iconst_1/ irem	1
	ifne/ iconst_1	1		ifne/ imul	3
	invokevirtual/ irem	6		invokevirtual/ iload_1	1
	imul/ invokevirtual	3		imul/ iconst_1	7
	irem/ imul	0		irem/ ifne	6
1	goto/ goto	1	5	goto/ goto	0
	if_icmpge/ if_icmpge	3		if_icmpge/ if_icmpge	1
	iload_1/ iconst_1	4		iload_1/ irem	4
	iconst_1/	2		iconst_1/ imul	2
	ifne/ iload_1	0		ifne/ iload_1	3
	invokevirtual/ imul	6		invokevirtual/ ifne	6
	imul/ irem	7		imul/ invokevirtual	5
	irem/ ifne	5		irem/ iconst_1	7
2	goto/ if_icmpge	7	6	goto/ if_icmpge	2
	if_icmpge/ goto	0		if_icmpge/ goto	5
	iload_1/ imul	6		iload_1/ iload_1	4
	iconst_1/ iconst_1	5		iconst_1/ iconst_1	1
	ifne/ invokevirtual	4		ifne/ ifne	6
	invokevirtual/ifne	3		invokevirtual/invokevir	7
	imul/ iload_1	1		imul/ irem	0
	irem/ irem	2		irem/ imul	3
3	goto/ goto	3	7	goto/ goto	6
	if_icmpge/ if_icmpge	6		if_icmpge/ if_icmpge	2
	iload_1/ iconst_1	2		iload_1/ iconst_1	5
	iconst_1/ ifne	5		iconst_1/ invokevirtual	3
	ifne/ irem	1		ifne/ imul	4
	invokevirtual/ imul	7		invokevirtual/ irem	1
	imul/ invokevirtual	4		imul/ iload_1	7
	irem/ iload_1	0		irem/ ifne	0

3.4 DFA-Based Obfuscation 的特性

在本章一開始的時候，介紹了一個好的混淆器應該具有什麼樣的特性，現在我們就來進一步檢視 DFA-Based Obfuscation 是否具有那些特性：

1. 保持原程式的功能：
   只有在特定的 Java 虛擬機器上執行時才能保持原有程式特性,如果任意更換虛擬機器將會使虛擬機器丟出例外或是當機，就某方面而言這是一種缺點，侷限了程式的流通及 Java 跨平台執行的能力，然而就保護軟體層面而言，此一取捨將會大幅增加軟體保護能力—即使攻擊者透過了不法管道取得了軟體，缺少了相關的虛擬機器亦將無法執行。
2. 讓攻擊者難以設計出自動化的工具：
   轉換規格的選擇全賴個人喜愛或是經驗，亦可輕易新增功能、額外屬性或是定期/不定期更換轉換規格，沒有什麼規則可利用來建立反編譯器。
3. 盡可能延遲反向工程所需的時間：
   只要轉換規格有經過一定的設計、考量，此法所提供的混淆程度將大幅增加反編譯的難度，欲執行反向工程非得要投入大量的人力、時間不可。
4. 保留或改善原程式的效率且以不增加過多的程式大小為限：
   當轉換規格之中沒有插入額外的指令之時，程式的大小即為原有的程式大小；效率方面雖須多經過一道轉換手續，然而因為少了執行類別檔驗證(bytecode verification)所需的時間，所以效能方面亦維持在與原程式相當之程度。

3.5 本章小結

本章描述 DFA-Based Obfuscation 的設計原理及相關事項。下一章將描述系統的實作及測試。

四、系統實作暨測試

經過了第三章設計原理的講解之後，本章將介紹系統實作及測試相關議題。

4.1 系統實作

為了實作上的方便，本論文採用的虛擬機器是「j2me_cldc-1_1-fcs-src-winunix」[31]，建構的目標平台是 Windows XP SP2。

4.1.1 系統建構—以類別載入為執行基礎的 DFA-Based Obfuscation 為例

整個系統的架構如同第三章圖 3.4 所示，下載 j2me_cldc-1_1-fcs-src-winunix 的原始碼之後，接著就可以進行新增元件的建構(參閱 3.3.1 節)。

首先，依 3.3.4 節所述的演算法建構一個轉換器(Ts)，其功用是讀入類別檔並依照使用者所選擇的轉換規格進行轉換。

其次，建構一個 RTs，功用是在 Java 虛擬機器的內部執行反轉換的工作，將混淆過的類別檔還原成原來的類別檔，以利後續執行。在此階段之中，並須在虛擬機器的原始碼之中找個適當的地方(於 4.1.3 小節解說)置入新增的程式碼，然後重新編譯。

4.1.2 反轉換實作上的選擇

在 3.3.3 節之中，曾說過有兩種方式可以執行轉換過的混淆程式，分別是：

1. 在虛擬機器的類別載入(Class loading)過程之中將利用 RTs 將 P_T 轉換回 P_0，以供直譯器執行程式。
2. 建構 DFA-Based 的直譯器，其功能是直接讀取已轉換過的指令執行程式，在執行過程中以 S 所定義的語意對應關係做為實際執行依據。

雖然在概念上這兩個實作方式相差無幾，不過仍有幾點相異之處值得細加考量，以下詳細解說之。

1. 以類別載入為執行基礎的 DFA-Based Obfuscation：

眾所皆知，Java 在執行時採用的是動態載入(dynamic loading)機制，除了一些基礎類別之外，其它類別只有在有需要的時候才會動態載入，並在載入過程之中執行類別檔驗證。當順利完成了上述的載入之後，虛擬機器就會讀取適當的指令以繼續執行該程式。在實作的時候，可以利用載入的過程執行反轉換的工作，如圖 4.1 所示：

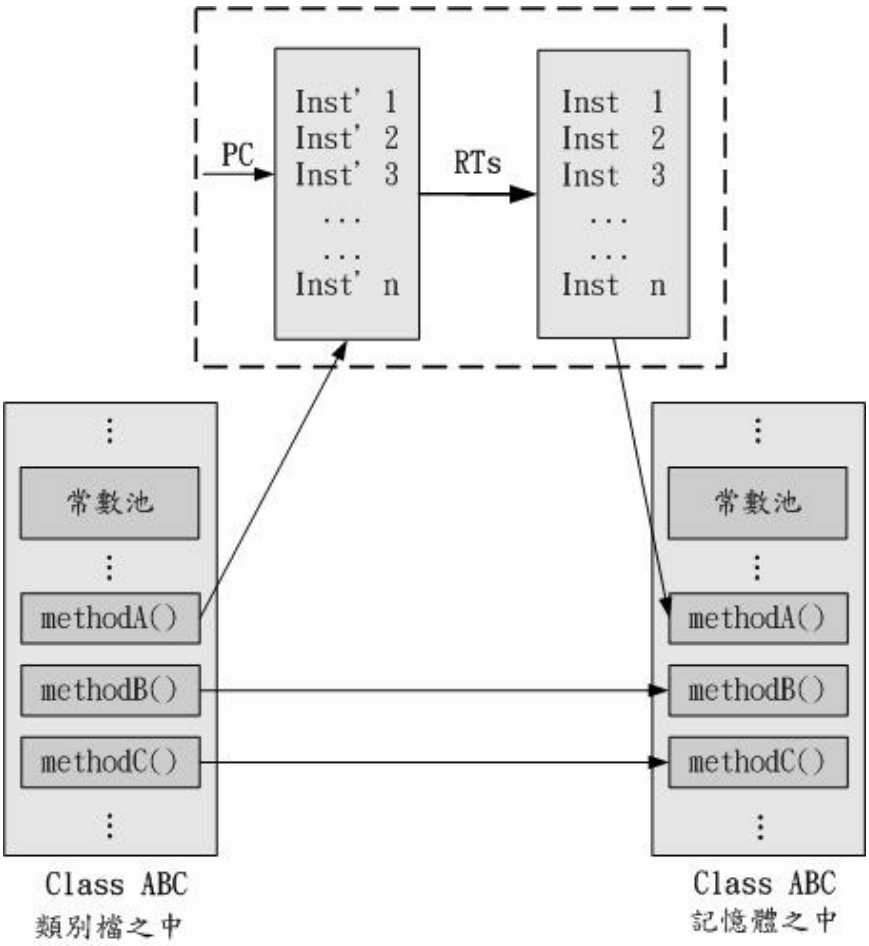


圖 4.1 在類別載入過程中進行反轉換

當載入一個未曾載入的類別時，在載入的過程之中，會將此類別所包含的所有方法一併做作反轉換(不論其載入之後是否會使用到)。

2. 以 DFA-Based 直譯器為執行基礎的 DFA-Based Obfuscation：

除了上述的方法之外，另一種的實作的方式是在直譯器擷取指令之後再進行反轉換，過程如圖 4.2 所示：

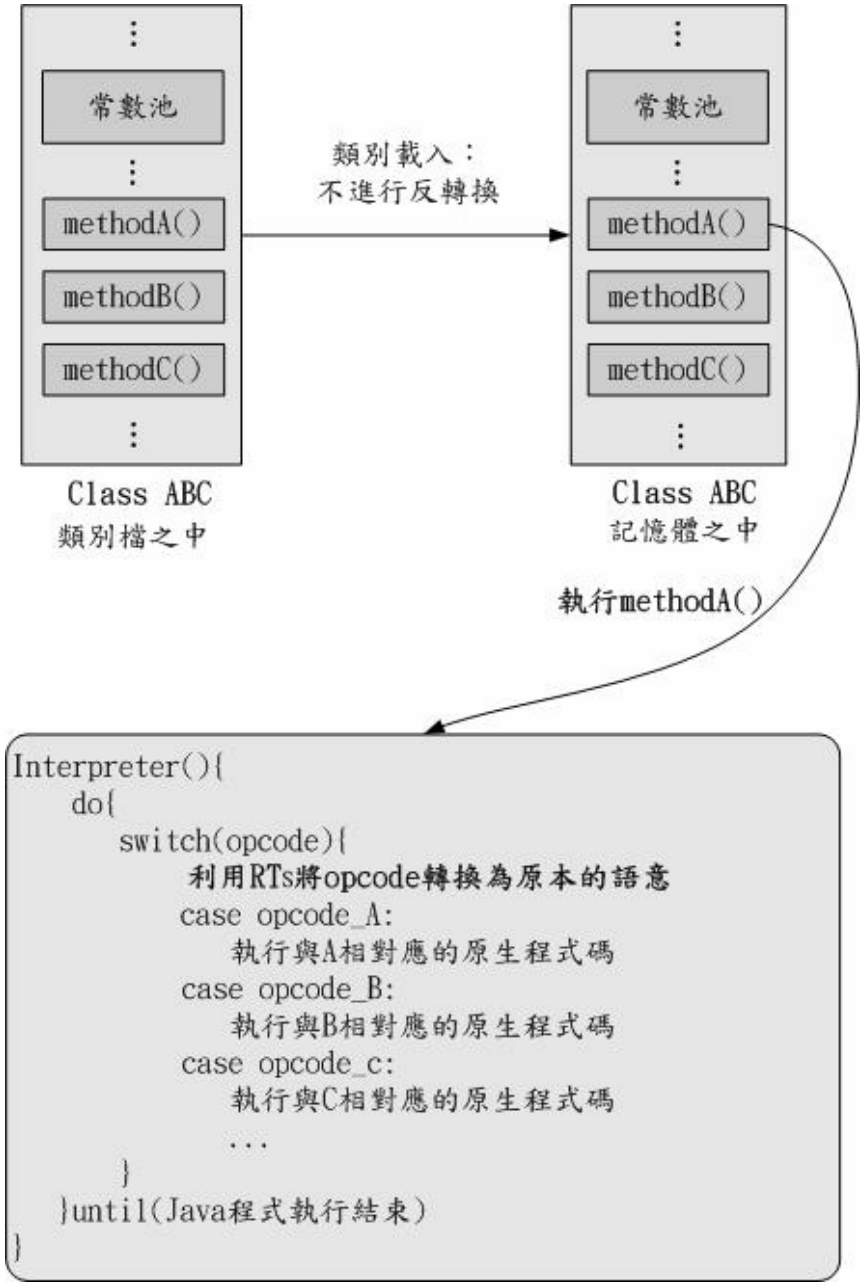


圖 4.2 DFA-Based 直譯器的示意圖

相較於 1.的方式，此法是等到真正需要執行的最後階段才進行反轉換。

比較兩者，第一種方式一次將一個類別之中所包含所有的方法全部一次進行反轉換，而第二種方式則是等到真正執行某一個方法時，一道指令一道指令依執行順序進行反轉換。

表面上看來這兩種方式並無多大差異，不過真正在實作時第二種方式會產生兩個重大問題：

1. 當遇到迴圈之時，第二種方式會產生一種惱人的問題，如圖 4.3 所示：

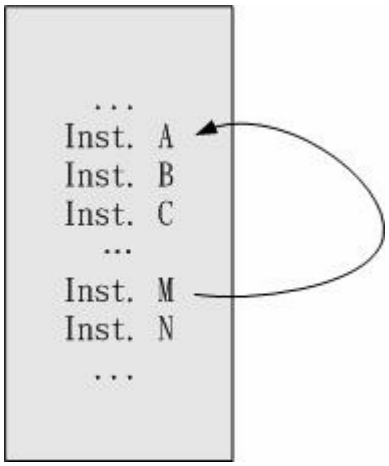


圖 4.3 以 DFA-Based 直譯器為執行基礎可能產生的問題

假設有個迴圈，其迴圈本體由 A 至 M，當第一次執行到此迴圈的時候一切安好，但是當第二次執行時，直譯器並沒有能力判斷這道指令是否已經轉換過？如果皆當作未轉換過的處理，則迴圈之中的指令會重複地進行反轉換；若新增一些資料結構作為判斷的輔助，則每次擷取一道指令後皆要判斷一次是否此道指令已轉換過，因此，可以很明顯的了解到，無論是上述何者，皆有其重大的缺點。

2. JVM 內原有的加速技巧難以使用：

眾所皆知，現在的 JVM 之中都有採用一些加速技術，例如 Just-In-Time Compiler，當使用第二種方式時，指令轉換的問題會令相關加速技巧難以發揮原有功效。

兩相比較，其實各有優缺，第二種方式雖有上述問題，然而正因為它是一道指令一指令的轉換，所以攻擊者若想進行反向工程將遭遇比第一種方式更難的環境；不過，相較之下第一種方式簡單、直覺、容易模組化，因此本論文採用第一種實作方式。

4.1.3 KVM 中反轉換器(RTs)的建構

如我們所知，Java 會於需要時動態載入類別，由於整個載入過程牽涉眾多步

驟，在此，僅就本研究相關者加以解說，整個過程約略如圖 4.4 所示。

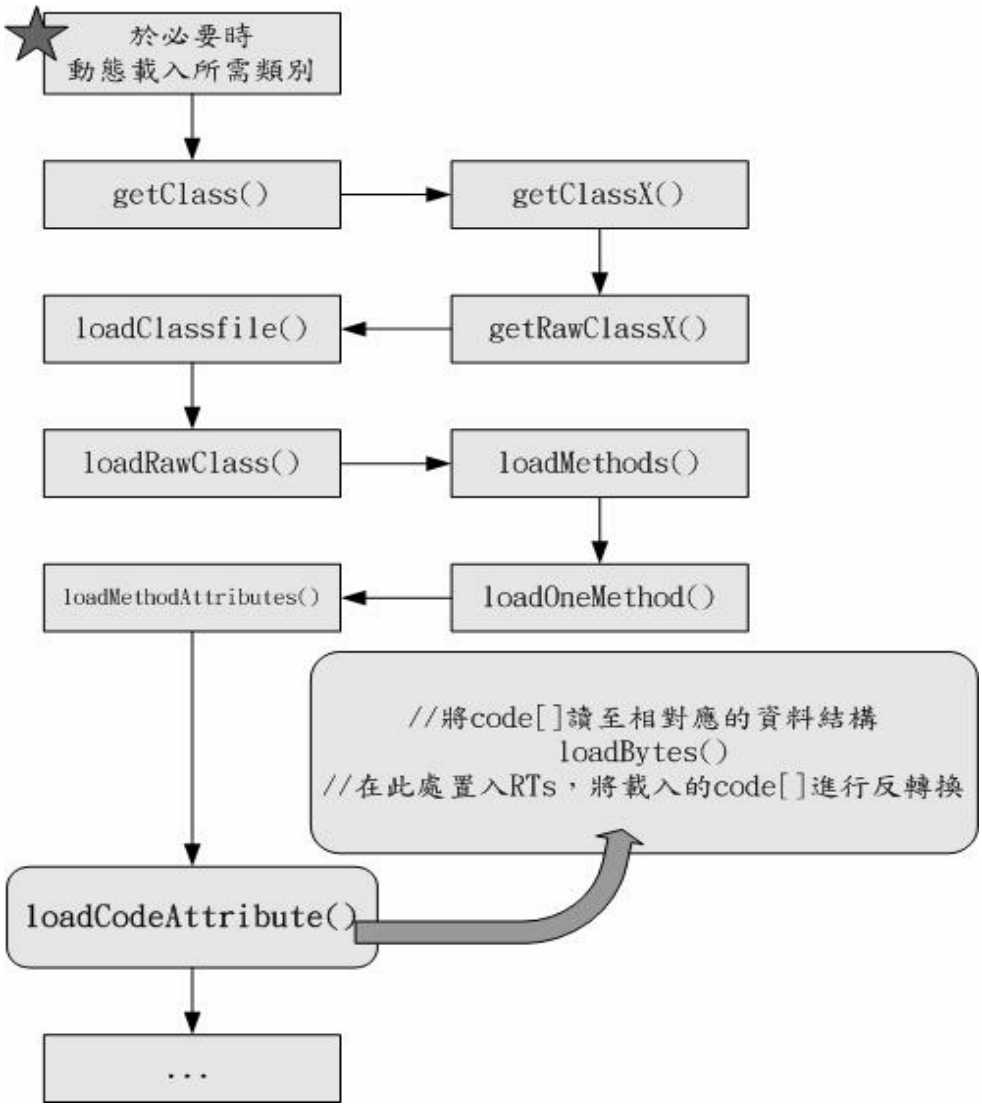


圖 4.4 類別檔的載入過程

當有需要的時候,KVM 會以 VmCommon/src/class.c 之中的 getClass()函式來將一個類別檔載入至記憶體，該函式內容如表 4.1 所示，其中參數 name 即為要載入的類別名稱，可以看出此函式最主要的作用就是呼叫 getClassX()。

表 4.1 getClass()的原始碼

```
CLASS
getClass(const char *name)
{
    if (INCLUDEDEBUGCODE && inCurrentHeap(name)) {
```

```
            fatalError(KVM_MSG_BAD_CALL_TO_GETCLASS);
        }
        return getClassX(&name, 0, strlen(name));
}
```

接著 getClassX() 在其函式中依序呼叫 getRawClassX() 及 loadClassfile()。前者的主要作用為在記憶體之中配置一空間用來存放 classStruc 結構，再將之加入一個名為 ClassTable 的 hash table 之中以利後續使用該類別時的查詢。而 loadClassfile()的主要作用在讀取該類別檔，並於過程之中進行相關解析，依需求遞迴或迭代載入父類別或父介面。其中讀取二元碼的工作指派給 loadRawClass()進行，該函式重要內容如表 4.2 所示。其中 loadMethods()載入定義於一個類別中所有的方法，當然，此部分是以迴圈一次載入一個方法進行，也就是 loadOneMethod()的功用。

表 4.2 loadRawClass ()的原始碼

```
static void loadRawClass(INSTANCE_CLASS CurrentClass, bool_t
fatalErrorIfFail)
{
   /* Load class identification information */
   loadClassInfo(&ClassFile, CurrentClass);

   /* Load interface pointers */
   loadInterfaces(&ClassFile, CurrentClass);

  /* Load field information */
   loadFields(&ClassFile, CurrentClass, &StringPool);

   /* Load method information */
   loadMethods(&ClassFile, CurrentClass, &StringPool);
}
```

其次 loadOneMethod()呼叫 loadMethodAttributes()，主要的作用在於載入該方法的 Code 及 Exceptions 這兩個屬性(參閱第三章，表 3.2-3.4)；而載入 Code 這個屬性的工作就由 loadCodeAttribute()此函式負責，其重要內容如表 4.3 所示。

表 4.3 loadCodeAttribute()的原始碼

```
Static unsigned int
loadCodeAttribute(FILEPOINTER_HANDLE ClassFileH,
                  METHOD_HANDLE thisMethodH,
                  POINTERLIST_HANDLE StringPoolH)
{
    unsigned int actualAttrLength;
    unsigned int codeLength;
    int nCodeAttrs;
    int codeAttrIndex;
    BYTE* code;
    bool_t needStackMap = TRUE;

    METHOD thisMethod = unhand(thisMethodH);

    /* Create a code object and store it in the method */
    thisMethod->u.java.maxStack = loadShort(ClassFileH); /* max stack */
    thisMethod->frameSize       = loadShort(ClassFileH); /* frame size */
    codeLength                  = loadCell(ClassFileH);  /* code length */
    /* KVM verifier cannot handle bytecode longer than 32 KB */
    if (codeLength >= 0x7FFF) {
        raiseExceptionWithMessage(OutOfMemoryError,
            KVM_MSG_METHOD_LONGER_THAN_32KB);
    }

    /* KVM frames cannot contain more than 512 locals */
    if (thisMethod->u.java.maxStack + thisMethod->frameSize >
            MAXIMUM_STACK_AND_LOCALS) {
        raiseExceptionWithMessage(OutOfMemoryError,
            KVM_MSG_TOO_MANY_LOCALS_AND_STACK);
    }

    /* Allocate memory for storing the bytecode array */
    if (USESTATIC && !ENABLEFASTBYTECODES) {
```

```
        code = (BYTE *)mallocBytes(codeLength);
    } else {
        //ENABLEFASTBYTECODES

    code=(BYTE)callocPermanentObject(ByteSizeToCellSize(codeLength));
    }

    thisMethod = unhand(thisMethodH);
    thisMethod->u.java.code = code;
    thisMethod->u.java.codeLength = codeLength;

    //由類別檔中，將 Code[]讀至相對應的資料結構
    loadBytes(ClassFileH, (char *)thisMethod->u.java.code, codeLength);

//置入反轉換器，以 Code[]中的指令串流進行反轉換

    actualAttrLength = 2 + 2 + 4 + codeLength;
...
}
```

最後，基於執行速度及本方法之特性，取消類別檔驗證，如表 4.4 所示。

表 4.4 取消類別檔驗證

```
void initializeClass(INSTANCE_CLASS thisClass){
if (thisClass->status == CLASS_ERROR) {
        raiseException(NoClassDefFoundError);
} else if (thisClass->status < CLASS_READY) {
        if (thisClass->status < CLASS_VERIFIED) {
            //disable the bytecode verification
//verifyClass(thisClass);
        }
…
}
```

4.1.4 系統套件的考量

如同在 3.5 小節所述，Java 擁有豐富的類別庫支援，可以說實用的程式一定會使用到系統的類別庫。在此需考量的問題是這些預設/延伸套件須要加以混淆嗎？還是保留原有的類別檔？整體概念如圖 4.5 所示。

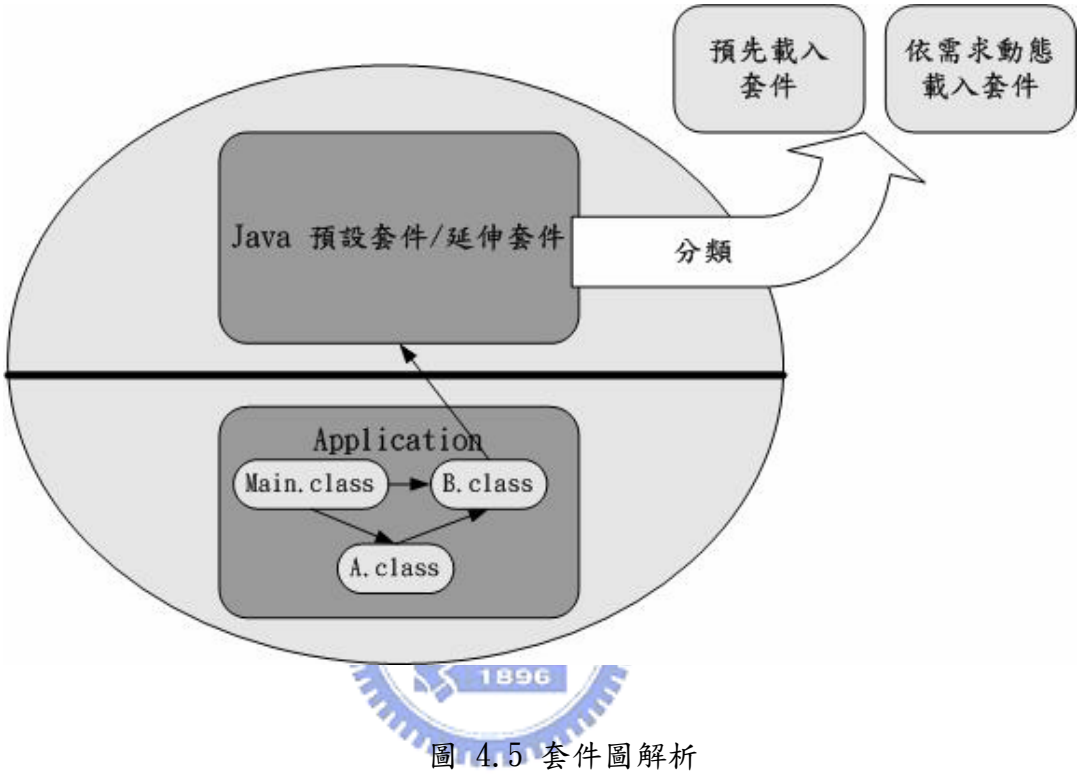


圖 4.5 套件圖解析

關於這個議題，亦有兩種實作方式：

1. 系統套件亦予以混淆：

   如同要保護的程式，亦將系統預設/延伸套件予以混淆，然而如此做之後將具有如下缺點：

   (1) 系統及延伸套件的功能皆是已知、容易了解的，對於這些套件攻擊者皆能輕易取得原有的類別檔，只要兩相比較，等於是提供了眾多的分析樣本予以攻擊者，以利其找出轉換規格。
   (2) 降低效能：一般系統及延伸類別之中的方法所佔空間皆有一定之大小，執行時予以轉換無疑會降低效能。

2. 系統套件不予以混淆：

因為 1.具有上述缺點，因此，建議採用第二種方式。然而，在實作的時候必須要能正確區分出正在執行的指令是來自於那個套件中的類別，所幸 Sun 的套件命名規則：以 java 以及 javax 開頭的套件名稱保留給 Sun 使用，他人不得使用，因此，可以藉由套件名稱加以判斷。

相關的資料結構及判斷方式如表 4.5 所示。

表 4.5 系統及延伸套件的判斷方式

if(thisMethod->ofClass->clazz.packageName ==NULL){
//anonymous package
進行反轉換並載入至記憶體
}else{
if(strstr(thisMethod->ofClass->clazz.packageName->string, JAVA) != NULL || strstr(thisMethod->ofClass->clazz.packageName->string, JAVAX) != NULL){
//預設/延伸套件，不進行反轉換，直接載入至記憶體
}else{
//非預設/延伸套件，進行反轉換並載入至記憶體
}
}	

說明	thisMethod：指向目前正在處理(載入)的方法 ofClass：指到定義有這個方法的類別結構 clazz：代表該類別的實體 packageName：指向存有 packageName 的結構的指標 string：真正儲存的字串內容

4.1.5 變換運算元(Change operand)

在 3.4 節圖 3.9 的轉換器演算法之中有一個「Change Operand」的項目，此項目的功能很簡單，判斷此道指令是否需要變換 operand，如果是，則依轉換規格進行轉換，如果不需要則不作任何處理。

設計此功能具有幾項優點：

1. 可以改變分支指令的 offset，讓反編譯器、攻擊者弄不清那裡是分支指令的跳躍目的地，例如：
   goto 2 → goto 18 將會使反編譯器、攻擊者弄錯真正的跳躍目的地。
2. 對一些 operand 代表常數池索引(index)的指令而言，此方式可以讓反編譯器、攻擊者誤用常數池之中的項目，例如：
   getstatic 10 → getstatic 20 將會使人誤以為是使用常數池中第 20 個項目。

整體的設計概念可以圖 4.6 表示：

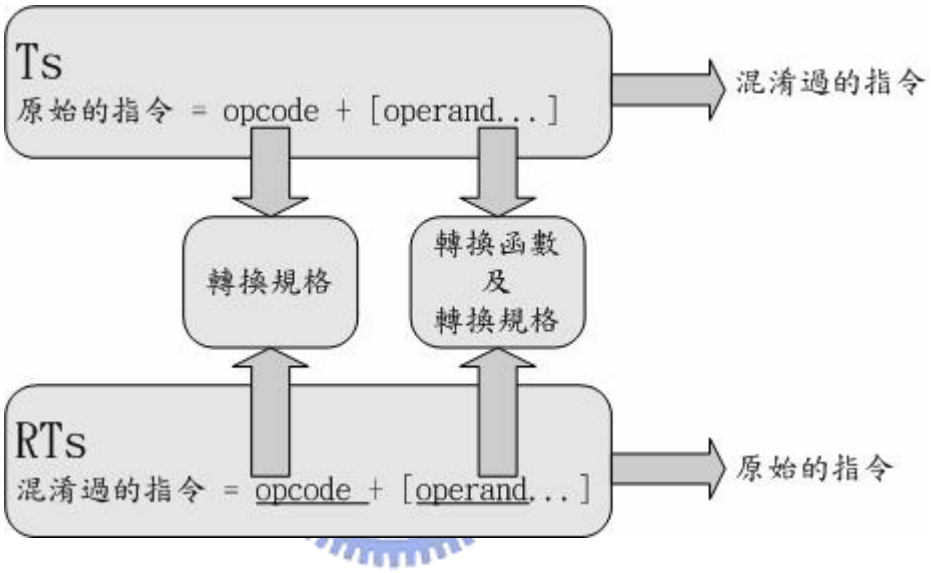


圖 4.6 Change operand 示意圖

其中轉換函數代表使用者可自由選定的數學函數 $f(x)$及$f^{-1}(x)$，分別安插在 Ts 及 RTs 之中，而 x 則置於轉換規格之中。當 Ts 有需要轉換 operand 之時依轉換規格取出 x 再計算 $f(x)$，而後即以 $f(x)$之值做為新的 operand，RTs 的建構原理亦同，不再贅述。

4.2 系統測試

測試環境如下：

1. 作業系統：Windows XP Professional SP2
2. 虛擬機器：j2me_cldc-1_1-fcs-src-winunix
3. 處理器：Celeron 2.0 GHz

4. 記憶體：256MB

5. 編譯器：Sun j2sdk1.4.2_03 版、Visual C++ 6.0

為了測試本論文所提「以定性的有限狀態機為基礎的 Java 程式混淆」之效果，於網路上取得各家發表的反編譯器進行測試，由於可取得的反編器過多(超過 20 套)，因此僅挑選其中具有代表性、使用不同反編譯引擎者加以測試，挑選後結果如表 4.2 所示。

表 4.6 測試用反編譯器列表

名稱 \ 項目	開發程式語言	版本	說明
Mocha[4]	Java	Beta 1	第一個 Java 反編譯器
SourceTec Decompiler[33]	Java	1.10	Jasmin Java Decompiler A patch of Mocha
Jad[14]	Win32	1.5.8e	有數個反編譯器使其引擎作為建構基礎
JReversePro[34]	Java	1.4.1	
JODE[35]	Java	1.1.2-pre1	有數個反編譯器使用其引擎作為建構基礎
Dava Decompiler[25]	Java	2.1.0	Soot framework 2.1.0
ClassSpy[36]	Java	2.0	
JAscii[37]	Java	1.0.20	Evaluation version

4.2.1 測試範例一

測試程式：3.3.5 節中的表 3.6

轉換規格：3.3.5 節中的表 3.9

表格符號：

X：無法產生.java 檔，或是產生了程式碼卻不完整或是無法重新編譯

◯：表示混淆後，反編譯成功，重新編譯成功，執行不正確

☆：表示混淆後，反編譯成功，重新編譯成功，執行正確

表 4.7 範例一測試結果

反編譯器名稱	使用原始碼	使用本法混淆後	說明
Mocha	X	X	版本老舊，無法辨識新版類別檔
SourceTec Decompiler	X	X	版本老舊，無法辨識新版類別檔
Jad	☆	X	
JReversePro	☆	X	
JODE	☆	X	
Dava Decompiler	☆	X	
ClassSpy	☆	X	
JAscii	☆	X	

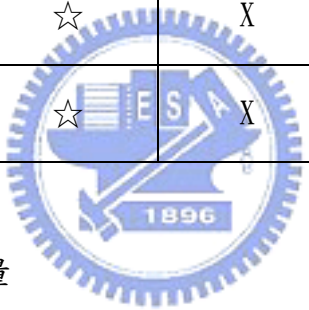
4.2.2 測試範例二

測試程式：附錄二

轉換規格：附錄二

表 4.8 範例二測試結果

反編譯器名稱	使用原始碼	使用本法混淆後	說明
Mocha	X	X	版本老舊，無法辨識新版類別檔
SourceTec Decompiler	X	X	版本老舊，無法辨識新版類別檔
Jad	☆	X	
JReversePro	☆	X	
JODE	☆	X	
Dava Decompiler	☆	X	
ClassSpy	☆	X	
JAscii	☆	X	



4.3 轉換規格之選擇考量

了解了相關元件建構上的考量後，接著探討一下在選擇一個轉換規格的時候，有什麼是值得我們加以注意的，以下分述之。

1. 在選擇「需混淆的指令」時，應考慮這道指令出現的頻率，在 202 個 opcode 之中，其實有些 opcode 出現的頻率非常低，建議可以適當地選擇幾個，不用多選。
2. 在選擇轉換規格之時，可以先分析一下原始程式編譯後的類別檔，針對最重要的部分予以最高度的混淆。
3. 此外，將分支指令變成非分支指令則流程會改變，反之亦然。
4. 改變分支指令的 operand，則其 offset 改變。
5. 對於參考到常數池的指令而言，改變其 operand 將會造成常數池的誤用。

當然，上述五點只是一個簡單的摘要，有興趣的讀者可以自行研究、探討。

4.4 應用方向

本論文所提出的「DFA-Based Obfuscation」在應用上適合使用於以下情況：當應用程式內含高價值的演算法或是不欲人知的機密，卻又擔心現有技術保護能力不足，則可搭配各種技術促成「多重防禦」之保護方式，讓反編譯的層級涵蓋原生語言，大幅增加反編譯難度並限制使用對象，沒有獲得相對應虛擬機器的人即使獲得了該程式也無法執行。

此外，當有需要時，亦可以建構一個能同時執行混淆過及未曾混淆過程式的系統，如下圖所示。舉例而言，數位電視之中亦使用了 JVM，在此，系統廠所開發的應用/系統程式可以保護模式執行,而使用者後續加入的應用程式可以一般模式執行。

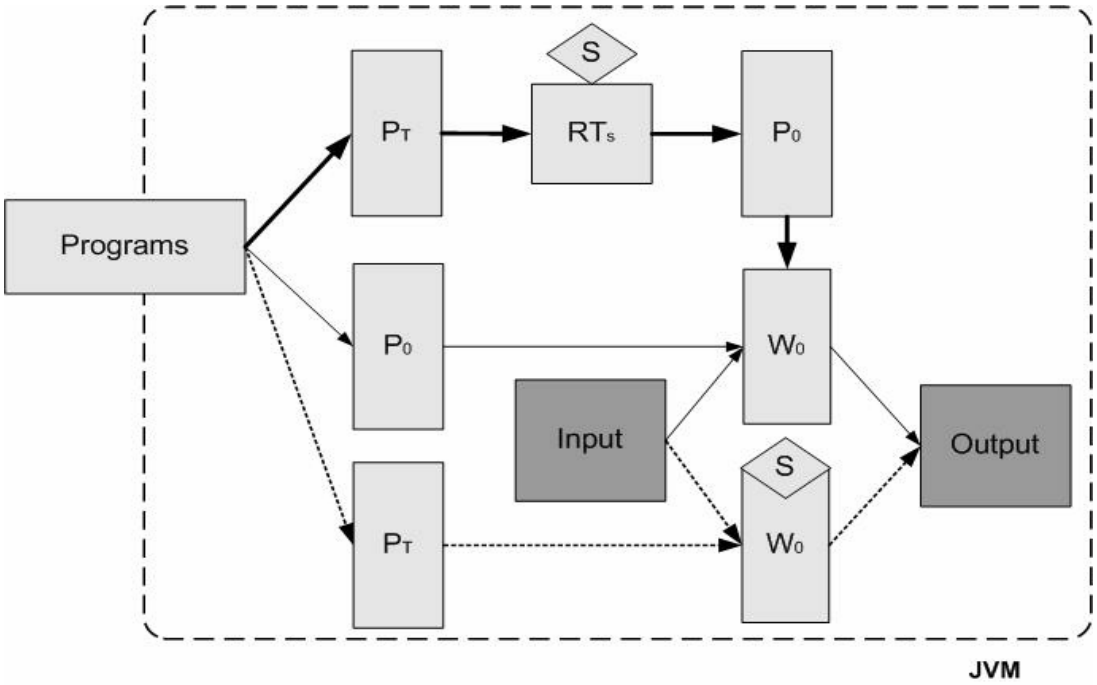


圖 4.7 應用方式—多執行模式系統

4.5 本章小結

本章一開始實作了一個 DFA-Based Obfuscation 系統並進行相關測試，測試結果顯示，只要轉換規格的選擇有一定之考量，則沒有一個反編譯器可以成功完成反編譯。在後面幾個小節討論了相關的優、缺點及補強方式，應用之時候須得多加留意。

五、結論與未來展望

5.1 結論

軟體保護與反編譯/反組譯就像早年美蘇雙方的軍備競賽一般，這場較競會持續到何時沒人知曉，亦無法判斷誰將獲得最後的勝利。處於現今這個年代，或許我們所能做的只有防護再防護，避免他人惡意侵害智慧財產權。

本研究提出的「以定性的有限狀態機為基礎的 Java 程式混淆」，是從既有保護技術的「互補」層面來看待整個保護系統，最主要的優點在於結合既有的保護技術提昇整體保護效果。

5.2 未來展望

當反編譯的工具、相關知識越來越容易取得、學習之際，混淆技術所擔負的責任就相形加重。未來，如何增加軟體反編譯的難度仍有賴於相關人員的研究。此外，由於智慧財產相關法規規定權利人需自行舉證以證明他人確有侵害本身權利，故軟體浮水印(Software Watermarking)的重要性將與日俱增，亦是值得加以研究的領域。最後，法律、道德上的防範更是值得加強的重點，如此才能全面、有效地防止他人惡意反編譯、竊取機密進而保護自身的智慧財產權。

參考文獻

[1] Tim Lindholm, Frank Yellin, "The Java™ Virtual Machine Specification" Second Edition, Addison Wesley, 1999

[2] Meyer, Downing, "Java Virtual Machine", O'REILLY, 1997

[3] Bill Venners. "Inside the Java 2 virtual machine", McGraw-Hill, 1999.

[4] Mocha, the Java Decompiler, http://www.brouhaha.com/~eric/computers/mocha.html

[5] Rachel Greenstadt, "Virtual Machine Technology Alone Cannot Stop Software Piracy".

[6] Collberg, Christian, Clark Thomborson and Douglas Low. "A taxonomy of obfuscating transformations", Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July, 1997. http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html

[7] Jien-Tsai Chan and W. Yang, " Advanced obfuscation techniques for Java bytecode", Journal of Systems and Software, July 2002. (NSC 89-2213-E-009-146 and NSC 90-2213-E-009-142).

[8] Bill Joy, Guy Steele, Jar Gosling, Gilad Bracha, "The Java Language Specification", Second Edition, Addison Wesley, 2000

[9] Mikhail Sosonkin, Gleb Naumovich and Nasir Menmon, "Obfuscation of Design Intent in Object-Oriented Applications", DRM' 03 October 27, 2003, Washington, DC, USA.

[10](美)Gamma Johnson, Helm Vlissides 著，物件導向設計模式，葉秉哲譯，培生，台北，民國八十九年。

[11]Douglas Low, "Java Control Flow Obfuscation", Master's Thesis, Department of Computer Science, University of Auckland, New Zealand, June 1998

[12]蔡明修，「使用反反編譯混淆法設計與實作 Java 抗加入式浮水印產生器」，國立成功大學，碩士論文，民國九十二年。

[13]Java Obfuscator: Zelix Klass Master, http://www.zelix.com

[14]Jad - the fast Java Decompiler, http://kpdus.tripod.com/jad.html

[15]Christian Collberg, Clark Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection", IEEE Transactions on Software Engineering, vol.28, no.8, August 2002, pp. 735-746

[16]Christain Collberg, Clark Thomborson, " Software watermarking: models and dynamic embeddings", In Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Antonio, Texas, United States, 1999, pp. 311-324

[17]Christian Collberg, Clark Thomborson, Douglas Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Diego, California, United States, 1998, pp. 184-196

[18]Douglas Low, "Protection Java Via Code Obfuscation", ACM Crossroads Student Magazine, April 1998, http://www.acm.org/crossroads/xrds4-3/codeob.html

[19]Christian Collberg, Clark Thomborson, Douglas Low, "Breaking Abstractions and Unstructuring Data Structures", IEEE International Conference on Computer Languages (ICCL'98).

[20]Chenghui Luo, Jian Zhao, "Obfuscating and Watermarking Java Software for Copyright protection", In the INI GraphicsNet research publications, Computer Graphik topics Issue 4, vol 11, 1999, pp. 31-32

[21]How to Write Unmaintainable Code Coding Obfuscation. http://mindprod.com/unmainobfuscation.html

[22]Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai A., Vadhan, S. and Yang, K. (2001), " On the (im)possibility of obfuscating

programs" Lecture Notes in Computer Science, 2139:1-18, Springer-Verlag.

[23]Patrick Lam., "Of Graphs and Grounds: Decompiling Java", Sable Technical Report No. 6, September 10, 1998.

[24]Raja Vallee-Rai, "Soot: A Java Bytecode Optimization Framework", Master Thesis, School of Computer Science, McGill University, Montreal, 2000.

[25]McGill's "Dava" Java Decompiler, http://www.program-transformation.org/twiki/bin/view/Transform/DecompilationDava#McGill_s_Dava_Java_Decompiler

[26]Miecznikowski, J.Hendren, "Decompiling Java Using Staged Encapsulation", In Reverse Engineering, 2001. Proceedings. Eighth Working Conference, Oct. 2001, pp.368-374

[27]Jclasslib, http://www.ej-technologies.com/products/jclasslib/overview.html

[28]Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S.,Stefanovic, D. and Zovi, D.D. (2003)," Randomized instruction set emulation to disrupt binary code injection Attacks", Proc. 10th ACM Conference on Computer and Communications Security (CCS2003), 281-289,Washington DC, USA.

[29]Jonh C. Martin. "Introduction to Languages and the Theory of Computation", Third Edition, McGRAW-Hill, 2003.

[30]Akito Monden, Antoine Monsifrot, Clark Thomborson, "A Framework for Obfuscated Interpretation".

[31]Java 2 Platform, Micro Edition (J2ME), http://java.sun.com/j2me/index.jsp

[32]探矽工作室著，深入嵌入式 Java 虛擬機器，初版，學貫出版社，民國九十一年。

[33]SourceTec Java Decompiler, http://www.srctec.com/decompiler/

[34]JReversePro,
http://sourceforge.net/project/showfiles.php?group_id=31100

[35]JODE, http://jode.sourceforge.net/

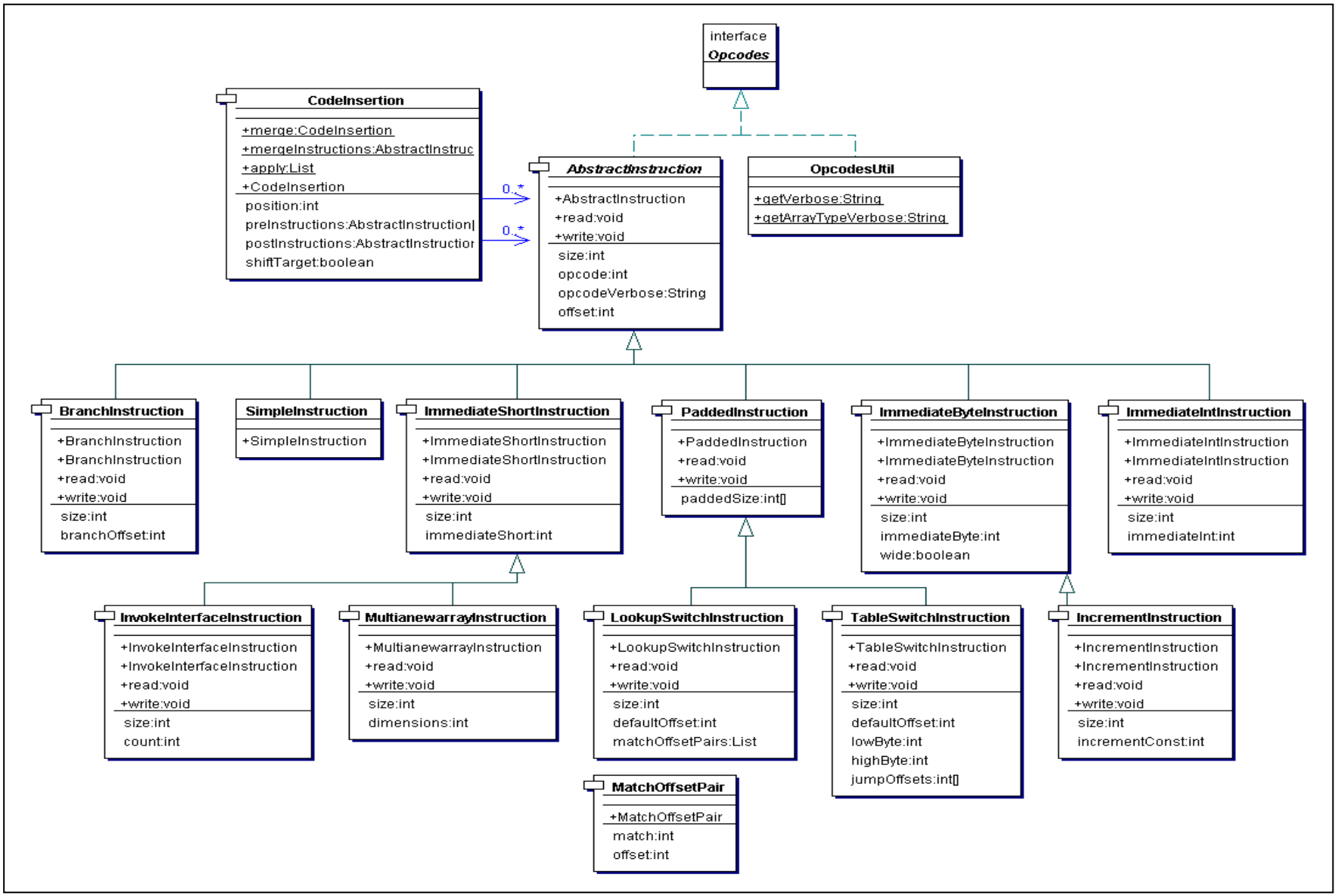
[36]ClassSpy,
http://www.freedownloadscenter.com/Programming/Java/ClassSpy.html

[37]JAscii, http://zdnet.com.com/3001-2417_2-10227092.html?idl=n

[38]Stunnix CXX-Obfus - the obfuscator for C and C++ source code.
http://www.stunnix.com/prod/cxxo/overview.shtml?g

[39]C++ Source Code Obfuscator.
http://www.semdesigns.com/Products/Obfuscators/CppObfuscator.html

附錄一



資料來源：jclasslib[26]

附錄一：指令分類

1.Simple Instruction <1>opcode 之後沒有 operand <2>指令長度為 1				
wide	nop	aconst_null	iconst_m1	iconst_0
iconst_1	iconst_2	iconst_3	iconst_4	iconst_5
lconst_0	lconst_1	fconst_0	fconst_1	fconst_2
dconst_0	dconst_1	iload_0	iload_1	iload_2
iload_3	lload_0	lload_1	lload_2	lload_3
fload_0	fload_1	fload_2	fload_3	dload_0
dload_1	dload_2	dload_3	aload_0	aload_1
aload_2	aload_3	iaload	laload	faload
daload	aaload	baload	caload	saload
istore_0	istore_1	istore_2	istore_3	lstore_0
lstore_1	lstore_2	lstore_3	fstore_0	fstore_1
fstore_2	fstore_3	dstore_0	dstore_1	dstore_2
dstore_3	astore_0	astore_1	astore_2	astore_3
iastore	lastore	fastore	dastore	aastore
bastore	castore	sastore	pop	pop2
dup	dup_x1	dup_x2	dup2	dup2_x1
dup2_x2	swap	iadd	ladd	fadd
dadd	isub	lsub	fsub	dsub
imul	lmul	fmul	dmul	idiv
ldiv	fdiv	ddiv	irem	lrem
frem	drem	ineg	lneg	fneg
dneg	ishl	lshl	ishr	lshr
iushr	lushr	iand	land	ior
lor	ixor	lxor	i2l	i2f
i2d	l2i	l2f	l2d	f2i
f2l	f2d	d2i	d2l	d2f
i2b	i2c	i2s	lcmp	fcmpl
fcmpg	dcmpl	dcmpg	ireturn	lreturn
freturn	dreturn	areturn	return	xxxunusedxxx
arraylength	athrow	monitorenter	monitorexit	breakpoint
impdep1	impdep2			

2. ImmediateByte Instruction <1>opcode 之後的第一個位元組為其 operand <2>指令長度為 2				
bipush	ldc	iload	lload	fload
dload	aload	istore	lstore	fstore
dstore	astore	ret	newarray	

3. ImmediateShort Instruction <1> opcode 之後的第一、二個位元組為其 operand <2>指令長度為 3				
ldc_w	ldc2_w	getstatic	putstatic	getfield
putfield	invokevirtual	invokespecial	invokestatic	new
anewarray	checkcast	instanceof	sipush	

4. Branch Instruction <1> opcode 之後的第一、二個位元組為其 operand <2>指令長度為 3 <3>灰影所示是 non-conditional branch instruction				
ifeq	ifne	iflt	ifge	ifgt
ifle	if_icmpeq	if_icmpne	if_icmplt	if_icmpge
if_icmpgt	if_icmple	if_acmpeq	if_acmpne	goto
jsr	ifnull	ifnonnull		

5. ImmediateInt Instruction <1> opcode 之後的第一到第四個位元組為其 operand <2>指令長度為 5				
goto_w	jsr_w			

6. Increment Instruction <1>opcode 之後的第一、第二個位元組分別代表兩個 operand <2>指令長度為 3
iinc

7. TableSwitch Instruction <1>operand 的全部數目依前幾個 operand 計算而得 <2>變動長度指令，指令長度需依其 operand 進行運算
tableswitch

8. LookupSwitchInstruction <1>operand 的全部數目依前幾個 operand 計算而得 <2>變動長度指令，指令長度需依其 operand 進行運算
lookupswitch

9. InvokeInterface Instruction <1>opcode 之後第一、二個位元組為 operand 1，之後兩個位元組為 operand 2 <2>指令長度為 5
invokeinterface

10. Multianewarray Instruction <1>opcode 之後第一、二個位元組為 operand 1，第三個位元組為 operand 2 <2>指令長度為 4
multianewarray

附錄二：測試程式二

PaperDemo2.java：

```
package a;

import java.util.Calendar;


public class PaperDemo2{

	public static void main(String[] args){

		Calendar rightNow = Calendar.getInstance();

		int[] added = { 1, 5, 10, 20};

		B b = new B();
		b.add(added);
		b.show();

		System.out.println("Date = " + rightNow.getTime());
	}
}
```

B. java：

```
package a;

import java.lang.Integer;
import java.util.Vector;

public class B{

	Vector v;

	public B(){
		v = new Vector(2);
	}



	public void show(){
		for(int i = 0; i < v.size(); i++){
			System.out.println("Element" + i + " = " + (Integer)v.elementAt(i));
		}
	}

	public void add(int[] added){
		Object ob = null;
		for(int i = 0; i < added.length; i++){
			ob = new Integer(added[i]);
			v.addElement(ob);
		}
	}
}
```

轉換規格(S)：

state	input/output	transition	x	state	input/output	transition	x
0	goto /if_icmplt	4	0	4	goto /if_icmplt	5	0
	if_icmplt /goto	7	2		if_icmplt/goto	4	1
	aload_2 /newarray	5	0		aload_2/bipush	2	0
	iconst_1 /aload_2	2	0		iconst_1/dup	1	0
	newarray/iconst_1	1	0		newarray/astore_2	3	2
	bipush/dup	6	3		bipush/aload_2	0	2
	astore_2/bipush	3	0		astore_2/iconst_1	7	0
	dup/astore_2	0	9		dup/newarray	6	0
1	goto/goto	1	0	5	goto/goto	0	3
	if_icmplt/if_icmplt	3	1		if_icmplt/if_icmplt	1	0
	aload_2/iconst_1	4	0		aload_2/dup	4	0
	iconst_1/bipush	2	0		iconst_1/astore_2	2	0
	newarray/aload_2	0	6		newarray/aload_2	3	5
	bipush/astore_2	6	0		bipush/newarray	6	0
	astore_2/dup	7	0		astore_2/bipush	5	0
	dup/newarray	5	0		dup /iconst_1	7	0
2	goto/if_icmplt	7	0	6	goto/if_icmplt	2	0
	if_icmplt/goto	0	4		if_icmplt/goto	5	1
	aload_2/astore_2	6	0		aload_2/aload_2	4	0
	iconst_1/iconst_1	5	0		iconst_1/iconst_1	1	0
	newarray/bipush	4	2		newarray/newarray	6	0
	bipush/newarray	2	0		bipush/bipush	7	0
	astore_2/aload_2	1	0		astore_2/dup	0	0
	dup/dup	2	0		dup/astore_2	3	0
3	goto/goto	3	0	7	goto/goto	6	1
	if_icmplt/if_icmplt	6	7		if_icmplt/if_icmplt	2	0
	aload_2/iconst_1	2	0		aload_2/iconst_1	5	0
	iconst_1/newarray	5	0		iconst_1/bipush	3	0
	newarray/dup	1	0		newarray/astore_2	4	2
	bipush/astore_2	7	2		bipush/dup	1	2
	astore_2/bipush	4	0		astore_2/aload_2	7	0
	dup/aload_2	0	0		dup/newarray	0	0