國立交通大學

電機學院電信工程研究所碩士班

碩士論文

基於特徵向量之快速病毒資料庫搜尋演算法及其實現

A Fast Malware Database Searching Algorithm and Implementation based on the Feature Vectors Method

研究生：邱嗣儒

指導教授：李程輝 教授

中華民國 一 佰 零 一年 七 月

基於特徵向量之快速病毒資料庫搜尋演算法及其實現

A Fast Malware Database Searching Algorithm and Implementation based on the Feature Vectors Method

研 究 生：邱嗣儒　　Student：Si-Ru Chiu

指導教授：李程輝　　Advisor：Tsern-Huei Lee

國 立 交 通 大 學

電機學院電信工程研究所碩士班

碩 士 論 文

A Thesis

Submitted to College of Electrical and Computer Engineering

National Chiao Tung University

for the Degree of

Master

in

Institute of Communication Engineering

July 2012

Hsinchu, Taiwan, Republic of China

中華民國 一佰零一 年七月

基於特徵向量之快速病毒資料庫搜尋演算法及其實現

學生：邱嗣儒　　指導教授：李程輝

國立交通大學電機學院電信工程研究所碩士班

摘　　要

隨著行動裝置與網路快速的成長，越來越多的使用者會使用行動裝置來上網或儲存重要資訊，卻也成為攻擊者所想要攻擊的目標，因此對於保護使用者個人隱私，如何有效地偵測惡意軟體是個很重要的問題。本論文引用 **Anthony** 團隊所開發的 **Androguard** 程式去建立應用程式的控制流字串。且引用 **Silvio Cesare** 及 **Yang Xiang** 所提出的特徵向量法將控制流字串轉成特徵向量來代表這個應用程式的特性。本論文之目的，為發展一套有效率的偵測方法，我們設計了一套快速病毒資料庫搜尋演算法。可以依據被偵測程式的特徵向量算出其曼哈頓距離，並判斷其落於我們預先制定的某個曼哈頓距離區間中，進而給予預先制定的二進位樹去搜尋，而不用對整個病毒資料庫去做搜尋，可減少搜尋時間，並且利用特徵向量的相似度去偵測已知病毒的變種。本論文使用兩種計算相似度的方法，一種是用來判斷是否為已知惡意軟體的變種，另一種是用來計算曼哈頓距離區間交界的值。

關鍵字：惡意軟體、控制流字串、特徵向量法、病毒資料庫搜尋演算法、曼哈頓距離、二進位樹、變種。

A Fast Malware Database Searching Algorithm and Implementation based on the Feature Vectors Method

student：Si-Ru Chiu　　Advisors：Prof. Tsern-Huei Lee

Institute of Communication Engineering

Electrical and Computer Engineering College

National Chiao Tung University

ABSTRACT

With the rapid growth of mobile devices and Internet, more and more users will use mobile devices to access the Internet or store important information. However, these users become the attackers' targets. Therefore, it is important to detect malwares effectively for protecting users' privacies. In this thesis, we quote "**Androguard**" (developed by **Anthony's team**) to build the application's control flow strings. In addition, we quote the "feature vectors method" (proposed by **Silvio Cesare** and **Yang Xiang**) for transforming the control flow strings to feature vectors that can represent the characteristic of the application. In this thesis, we design a fast malware database searching algorithm for detecting malwares effectively. First, we can calculate the query's Manhattan distance from its feature vector, and determine which section it locates at. Then we do not have to search the total malware database, but just search the specific tree for decreasing the searching time. In addition, we use the similarities of feature vectors to detect the malwares' variants. We use two methods to calculate similarity between applications. One is used to determine the application is malware or not. Another is used to calculate the Manhattan distance sections' boundaries.

Keywords: malwares、control flow strings、feature vectors method、malware database searching algorithm、Manhattan distances、binary trees、variants。

誌　　謝

能夠完成我的研究以及這篇文章，要先感謝我的指導教授—李程輝教授。教授指導我正確的研究態度和方法，訓練我邏輯思考以及解決問題的能力，讓我知道如何做好研究。

再來要感謝我實驗室的夥伴—姜家安，我們一起研究 Android 所存在的問題，找解決的方法，討論怎麼做可以改善這些缺點。還要感謝實驗室的學長姊，以及學弟們，還有其他的同學給我的建議與鼓勵。

感謝我的父親—邱天化先生以及母親—李燕枝女士，他們在我焦慮以及失去動力的時候，給我鼓勵，陪我走過難關。並且供應我食衣住行，使我能夠全心全力在研究上面。

感謝陪伴我六年的桌球校隊，讓我在研究之餘，能夠拓展人際關係，增進球技，為校爭光。

最後謹將此論文獻給身邊所有愛我的人及我愛的人。

2012/07　邱嗣儒

目　　錄

圖　目　錄

表　目　錄

第一章

簡介

1.1 研究動機

隨著網路快速的發展，以及智慧型手機的普及性，已經有很多種惡意軟體散佈在Android Market跟其他的網站。由於現在大多數使用者都會將重要的資訊存在手機裡面，且通常不會安裝防毒軟體來保護手機，這也是攻擊者想要攻擊的主因。使用者在下載完應用程式並安裝之後，由於使用時並沒有感覺到異狀，所以往往在不知情的情況下，資訊被惡意的應用程式利用網路或是簡訊傳送到攻擊者指定的位置；或者是自動撥打電話到一個特定的號碼，攻擊者可以從中獲利，造成使用者電話費帳單驟增。

圖1.1 趨勢科技專家預測惡意軟體成長趨勢

圖1.1是趨勢科技的專家研究統計，他們預估在2012年底散佈在各個地方的惡意軟

體總數量會超過120,000。因此我們可以知道惡意軟體的發展是相當快速，而且會散佈在很多地方，所以要如何解決這快速發展的惡意軟體是很重要的問題。不僅是惡意軟體發展快速，一般的應用程式也是發展快速。我們可以由下面的圖1.2發現應用程式的成長率，在2011年底的時候，發佈在Google Play上面的應用程式大約是600,000，而同時在所有的網站大約有1,000,000個不同的應用程式

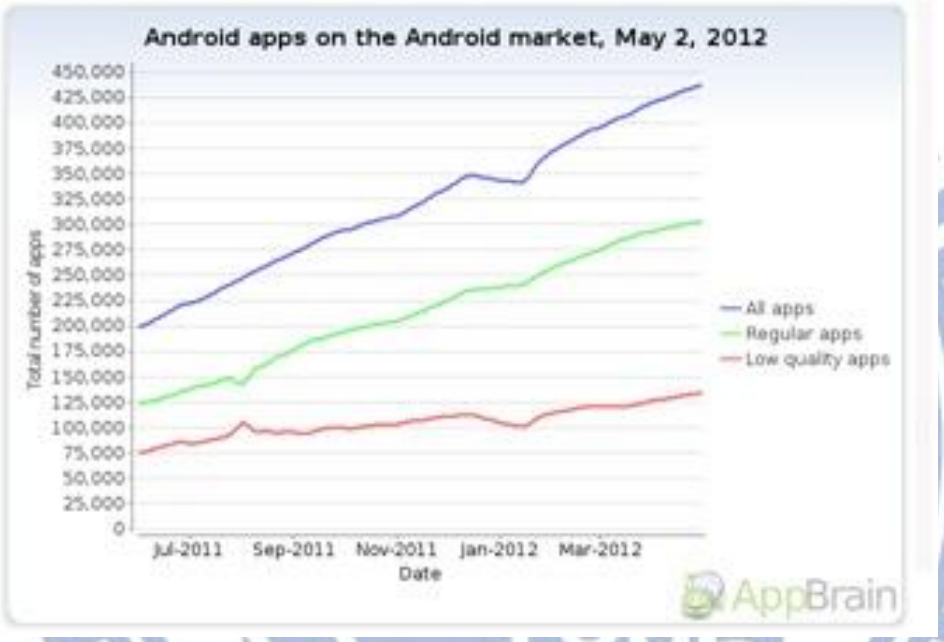


圖1.2 趨勢科技統計在Android Market的應用程式數量

智慧型手機在偵測惡意軟體的時候會有兩個主要的限制：

第一個限制就是運算的時間。由於手機的體積較小，手機的CPU無法做到像電腦一樣好，因此在運算上會比較慢。因此我們需要在偵測的時候，盡量減少運算，以達到能夠即時的偵測惡意軟體的效果。第二個限制是儲存的容量。由於手機的儲存容量較小，因此我們不能夠像在電腦上面做分析時用到大量的儲存空間，我們必須盡可能的節省儲存空間，並同時達到偵測的效果。因此我們希望能夠利用儲存少量的惡意軟體在資料庫中，能夠偵測到大量的未知的惡意軟體，以達到節省儲存空間的目的。

1.2 研究背景

偵測惡意軟體的方法大致可以分成兩類，第一類是靜態分析，第二類是動態分析。靜態分析跟動態分析最主要的差別在於，靜態分析並不需要去執行應用程式就能進行分析，而動態分析需要在執行應用程式的同時進行分析。我們的研究著重在靜態的分析上面，因為我們想要在使用者尚未安裝應用程式時去做偵測，並即時發現問題。靜態分析最普遍的做法是靜態特徵碼字串法，將每個應用程式建立一組特徵碼字串，然後將需要偵測的應用程式的特徵碼與資料庫中的病毒特徵碼字串去做比對。但是這種方法有一個很大的缺點，就是這種方法是將應用程式中的byte code擷取一段下來，當作這個應用程式的特徵碼字串。這麼做會使得即使這個應用程式的行為跟功能跟資料庫中的病毒是很接近的也無法被偵測出來，因為byte code會不一樣，所以無法比對出來。這種方法一定要資料庫中有跟被偵測的應用程式的特徵碼字串完全相同才會被偵測出來，但是隨著現在惡意軟體快速的發展，不可能每一個惡意軟體的特徵碼都儲存在資料庫中，所以為了解決這個缺點，近年來學者的研究，著重在應用程式的結構上，而不是應用程式的byte code。

近年來的研究有一種方法，稱為control flow graph method。一個control flow graph代表著應用程式中一個method所執行的路徑，一個應用程式中由許多method所組成，所以我們可以用許多control flow graph來代表這個應用程式。這個方法的最大優點，在於他是利用應用程式的結構去建立特性，所以即使在byte level或是instruction level有不同，在control flow graph的結果並不會有改變。這種方法已經被證實能夠有效地偵測出惡意軟體的變種，因為它能夠建立惡意軟體中較不變的特性，即使惡意軟體自動突變產生變種，也可以有效地偵測出來。要能夠偵測出變種，通常都會用一些數學式子去計算字串之間的距離，進而計算出相似度。所謂相似度指的是字串間越相像，距離就越接近，相似度的值就越大。兩個字串完全相同，距離為0，相似度為1。兩個字串不同，距離大

於0，相似度隨著距離增大而變小。在惡意軟體偵測中需要在正常應用程式與惡意軟體之間的相似度中先選出一個臨界值，用來當作判別是否為惡意軟體的依據。這個臨界值的大小會隨著被拿來訓練的應用程式的不同以及所要求的誤判率不同而改變，是一個隨著環境改變的變數。

1.3 論文結構

這篇論文的結構如下所敘述：我們會先在第二章中介紹一篇論文，主要是要解決惡意軟體變種的問題。這篇論文使用了數學的方法去建立原本字串的向量，完成原本無法在多項式時間內的距離計算，並且能夠有效率地偵測出惡意軟體的變種。我們也會提出我們的觀點，舉出覺得這篇論文可以改進的地方。在第三章時我們會詳細地介紹我們的方法，一開始會先描述問題，以及我們的動機，最原始的模型。再來會介紹我們的系統架構圖，以及每一個部分所需要做的事情。我們的系統主要分成兩大部分，一部分是偵測前所需要做的準備工作，另一部分是偵測。第一部分是主要的部分，因為這一部分的好壞會影響到第二部分的準確度。包含臨界值的挑選，以及儲存在資料庫中惡意軟體的特徵碼，都會影響到下一部分的偵測，因此我們會主要注重在這一部分的介紹。我們將會把模擬的結果展示在第四章中，包含誤判率的比較、以及資料庫搜尋時間的比較、偵測變種能力的比較、以及臨界值大小選擇的環境、q-gram擷取資訊前後的不同、基底刪減前後的差別、基底的比重如何分配。最後在第五章結論的地方，會總結我們所解決的問題，我們的做法，以及我們的貢獻，以及將來可以改進的地方。

第二章

相關文獻

2.1 逆向工程

在分析應用程式之前，需要做逆向工程。逆向工程的目的在於對那些有壓縮過或有混淆過的程式進行反向工作，為了要得到最初的原始碼以進行分析。Android的程式執行檔是".dex"的檔案，由於我們沒有辦法直接得到原始碼去做分析，所以我們需要透過逆向工程將二進位碼轉換得到原始碼。在Android的逆向工程中，已經有大量的工具可以被拿來使用。

2.2 Basic Block

簡單的說，一個basic block是一個區塊中連續的code，進入一個basic block只能在這個basic block一開始的地方，而要離開這個basic block只能在這個basic block最後結束的地方。而在計算機領域中，一個basic block是在一個程式中某一個區塊的程式碼，他擁有一些我們期望的特性使得其易於用來分析這個應用程式。編譯器通常會在分析的第一步將程式先分解成basic blocks，然後用來當作control flow graph的節點

2.3 Androguard

Androguard主要是一個以python編寫的工具集合，可以處理Dex、APK、Android's binary xml等類型的檔案，並且可以執行在Linux、Windows、MacOSX等作業系統。我們為了更接近實際Android手機的環境，使用Ubuntu(Linux kernel)的作業系統來安裝

Androguard。Androguard中有包含逆向工程的工具，可以將APK檔案反向得到原始碼。 在得到原始碼以後，可以利用Androguard依照basic block的定義，將原始碼分解成許多basic blocks，使得程式碼為一連串的basic blocks所組成。得到這一連串的basic blocks，即得到這個程式的control flow graph組合。（圖2.1）

```
        0(0) const/4 v0 , [#+ 0] , {0} [ testMultipleLoops-BB@0x2 ]

testMultipleLoops-BB@0x2 :
        1(2) const/16 v1 , [#+ 50] , {50}
        2(6) if-lt v0 , v1 , [+ 15] [ testMultipleLoops-BB@0xa  testMultipleLoops-BB@0x24 ]

testMultipleLoops-BB@0xa :
        3(a) rem-int/lit8 v1 , v0 , [#+ 3]
        4(e) if-eqz v1 , [+ 14] [ testMultipleLoops-BB@0x12  testMultipleLoops-BB@0x2a ]

testMultipleLoops-BB@0x12 :
        5(12) const/16 v1 , [#+ 789] , {789}
        6(16) if-ge v0 , v1 , [+ 6] [ testMultipleLoops-BB@0x1a  testMultipleLoops-BB@0x22 ]

testMultipleLoops-BB@0x1a :
        7(1a) const/16 v1 , [#+ 901] , {901}
        8(1e) if-gt v0 , v1 , [+ 9] [ testMultipleLoops-BB@0x22  testMultipleLoops-BB@0x30 ]

testMultipleLoops-BB@0x22 :
        9(22) return-void

testMultipleLoops-BB@0x24 :
        10(24) add-int/lit8 v0 , v0 , [#+ 2]
        11(28) goto [+ -19] [ testMultipleLoops-BB@0x2 ]

testMultipleLoops-BB@0x2a :
        12(2a) mul-int/lit8 v0 , v0 , [#+ 5]
        13(2e) goto [+ -18] [ testMultipleLoops-BB@0xa ]

testMultipleLoops-BB@0x30 :
        14(30) sget-object v1 , [field@ 0 Ljava/lang/System; Ljava/io/PrintStream; out]
        15(34) const-string v2 , [string@ 335 'woo']
        16(38) invoke-virtual v1 , v2 , [meth@ 7 Ljava/io/PrintStream; (Ljava/lang/String;) V pr
intln]
        17(3e) goto [+ -22] [ testMultipleLoops-BB@0x12 ]
```

```
Ltests/androguard/TestLoops; testMultipleLoops ()V
-> :    B[]B[I]B[I]B[I]B[I]B[R]B[G]B[G]B[F0SP1G]
```

圖2.1 將basic blocks轉換成字串的範例

在這個例子中，我們可以明顯的發現，在basic block最後一行必定會有要到別的basic block的指令，表示符合basic block的定義，basic block中離開的地方必在其結束的地方。我們也可以發現在BB@0x24這個basic block中，由於其最後一行的指令為goto BB@0x2，即離開BB@0x24這個basic block並前往BB@0x2這個basic block，所以basic

block BB@0x2上面自然就會再被切分一塊basic block出來。

而我們想要對其做進一步的分析，所以需要將一連串的basic blocks轉換成字串，而Androguard將一連串的basic blocks轉換成字串的文法轉換表，我們列在圖2.2。

```
Procedure ::= StatementList
StatementList ::= Statement | Statement StatementList
Statement ::= BasicBlock | Return | Goto | If | Field | Package | String | Exception
Return ::= 'R'
Goto ::= 'G'
If ::= 'I'
BasicBlock ::= 'B'
Field ::= 'F'0 | 'F'1
Package ::= 'P' PackageNew | 'P' PackageCall
PackageNew ::= 'C'
PackageCall ::= 'M'
PackageName ::= Epsilon | Id
String ::= 'S' Number | 'S' Id
Exception ::= Id
Number ::= \d+
Id ::= [a-zA-Z]\w+
```

圖2.2 control flow graph轉換成字串的文法轉換表

這個文法轉換表的意思是，如果遇到左邊的情況，則將其轉換為”::=”右邊的參數。我們可以用比較簡單的方式來表達以上的文法表(圖2.3)。

```
將一個Procedure轉換成一個StatementList。
一個StatementList即為一個Statement，或是Statement和StatementList組成。
一個Statement即為一個BasicBlock，或是一個Return，或是一個Goto，或是一個If，或是一個Field，或是一個Package，或是一個String，或是一個Exception。
將Return轉換成一個R。
將Goto轉換成一個G。
將If轉換成一個I。
將BasicBlock轉換成一個B。
將Field轉換成一個F0或是一個F1。
將Package轉換成一個P和一個PackageNew，或是轉換成一個P和一個PackageCall。
將PackageNew轉換成一個C。
將PackageCall轉換成一個M。
將PackageName轉換成一個空集合，或是一個Id。
將String轉換成一個S和一個Number，或是一個S和一個Id。
將Exception轉換成一個Id。
將Number轉換成十進位數字。
將Id轉換成大寫小寫的英文字母。
```

圖2.3 簡易的文法轉換表

BasicBlcok則是在每個basic block一開始的地方轉換成B，然後接兩個中括號[]，

並且將basic block裡面的程式碼轉換後的參數寫在括號裡面。例如Return、Goto、If這些是指令,所以在basic block中的程式碼遇到這些指令要轉換成相對應的參數。Field是java語言中，用來描述屬性的一個變數，在java中使用屬性跟方法來描述一個物件。Package，java語言中將許多classes組合在一起的架構。PackageCall為呼叫一個已經存在的package。PackageNew為創立一個新的Package。String為java中一種資料型態，由一群characters所組成的一個序列。

我們在利用文法轉換表將control flow graph轉換以後可以得到一組字串來代表這個應用程式(圖2.1)。但是就如同第一章所說明的，若是利用字串的edit distance去計算字串之間的距離來算應用程式之間的相似度，會無法在多項式時間下完成，所以我們參考了一篇論文,他有一個很大的貢獻,就是第一個使用向量來表示control flow graph的字串，可以克服無法在多項式時間下完成相似度的計算這個問題。並且另一個優點是他可以進而利用control flow graph的較不變的特性，建立資料庫中的惡意軟體的特徵碼，來偵測跟資料庫中相似的惡意軟體變種。

2.4 特徵向量法(Feature Vectors Method)

Silvio Cesare 及Yang Xiang首先提出使用向量的方法來表示control flow graph的字串，他們發現利用向量不但可以解決運算時間無法在多項式時間內完成這個問題，還能夠利用control flow graph本身的較不變特性建立惡意軟體的特性，並將其特徵碼儲存在資料庫中，能夠在比對特徵碼的時候，偵測到跟資料庫中特徵碼相似但不完全相同的變種。圖2.4為其簡易的系統架構圖，我們接下來會說明每一個方塊中的功用。其中1~4為偵測前準備工作，5~11為偵測時的工作。

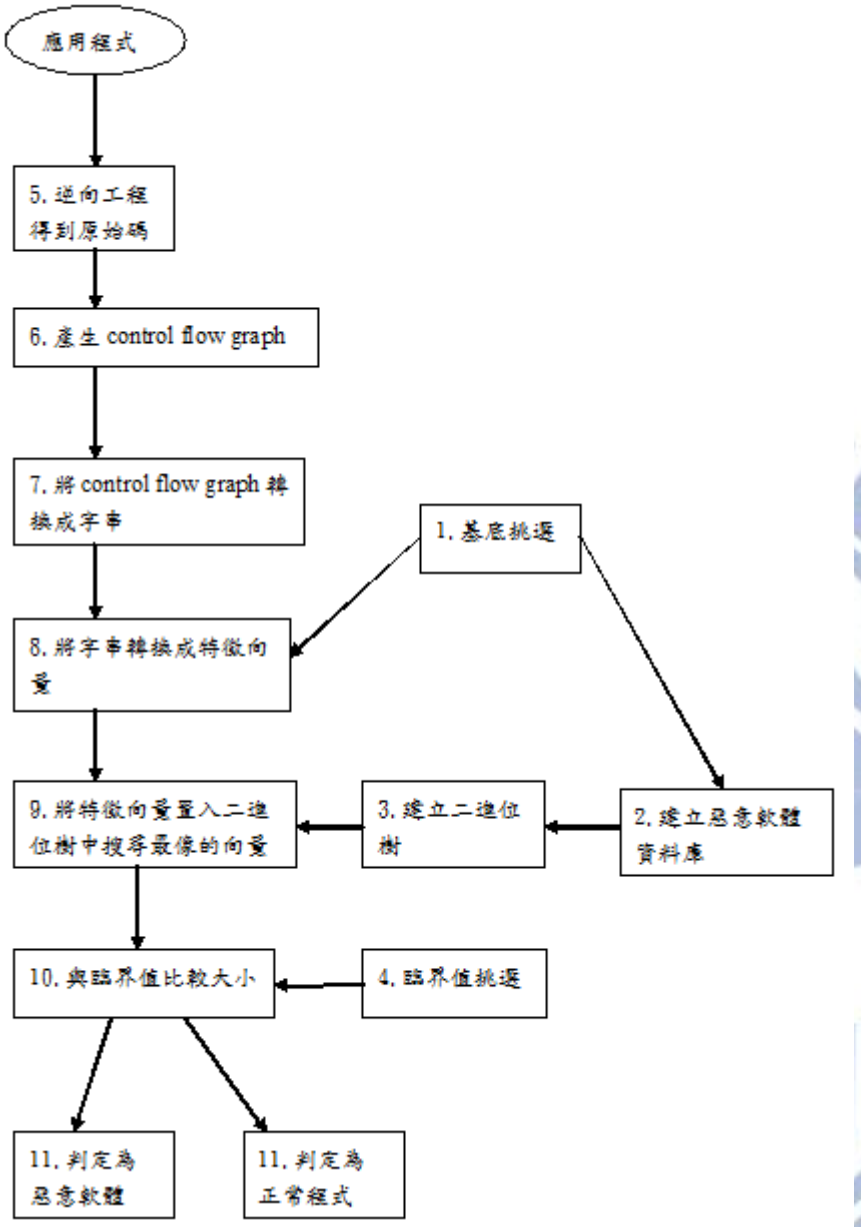


圖2.4 特徵向量法系統架構圖

2.4.1 偵測前準備工作

(1)基底挑選

在這個部分，從一群惡意軟體以及一群正常的程式，先將每一個程式都經過逆向工程得到這個應用程式的原始碼，再經由basic block的定義將原始碼切分為許多basic blocks所組成，再藉由這些basic blocks產生control flow graph，然後經由文法轉換

表轉換成字串，這時每一個程式都擁有一組字串，然後將連續長度為4的子字串視為一組特性，舉個簡單的例子說明：假如這個字串為”abcdef”，則其特性有”abcd”、”bcde”、”cdef”。因此每一個程式都會有一些特性，先統計這群惡意軟體跟這群正常的程式全部的特性出現的次數，並將其由多排到少。作者認為挑500個出現最多次的特性當作特徵向量的基底，除了這500個以外的特性都被忽略，因為出現的次數不多，無法足夠的代表這個程式的特性。我們也可以將這個部分畫成一個簡易的流程圖(圖2.5)。

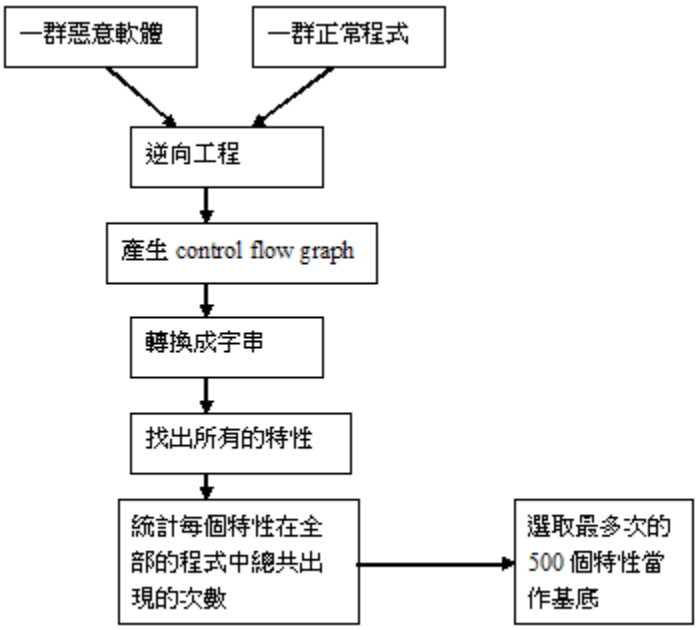


圖2.5 挑選基底流程圖

(2)建立惡意軟體資料庫

在挑選完基底以後，我們就可以建立每個惡意軟體的特徵向量，由於剛剛在挑選基底的時候已經有得到它們的字串，以及每個惡意軟體所擁有的不同特性以及次數，所以我們就不用在重複做一次逆向工程、產生control flow graph、產生字串等工作。我們只需要將字串依照500個基底(特性)的順序，填上相對應的特性所出現的次數即可轉換

成特徵向量。特徵向量的維度指的是不同種特性，而大小指的是這種特性在這個程式中所出現的次數。在這個例子，特徵向量為500維，每個分量的值是一個大於或等於0的整數，但是不會每個分量都是0。建立完特徵向量之後，分配給每個惡意軟體一個Id，Id的值是連續的正整數，然後將Id跟特徵向量存起來，之後在比對，要算相似度時再去查詢特徵向量。

(3)建立二進位樹

在建立完惡意軟體的資料庫以後，由於惡意軟體的數目很多，所以在比對的時候可能跟每一個資料庫中的惡意軟體都去算互相的相似度，這樣會浪費很多時間。因此作者使用Vantage Point Tree的方法建立一個二進位樹，使得其搜尋的複雜度為O(log(n))，其中n為惡意軟體的數目。下圖 2.6為vp tree分解的一個範例，其中弧線是代表將一個區域中分成內部跟外部，而點是代表vantage point。

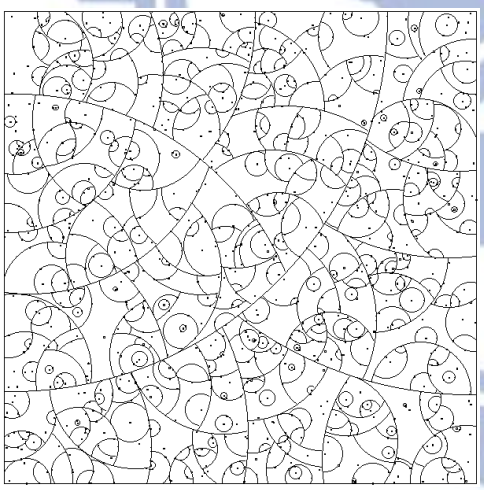


圖2.6 vantage point切分範圍圖

Vantage point tree主要的目的是希望能夠將不同的點平均分散，意思就是希望在交界(弧線)的部分點能夠越少越好，vp tree中的每一個節點都是vantage point，意思是透過這個二進位樹搜尋，進來的時候先跟root比較，並算出一個值，然後用這個值跟這一個區域裡面的交界值(弧線的值)比較，若是小於交界值就去左邊的child裡面繼續做一樣的事情，若是大於交界值就去右邊的child裡面繼續做一樣的事情。至於vantage

point的點從root一值往下的child分別是那些點，要如何挑選呢?我們可以先看圖2.7。

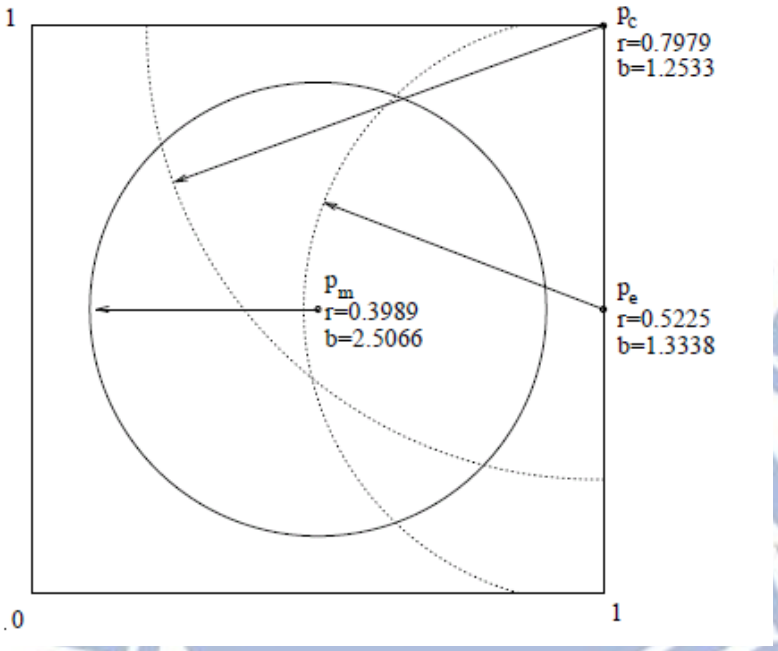


圖2.7 vantage point 的挑選圖

在這個方形的區域中，橫軸跟縱軸的值為0~1，P_e、P_c、P_m分別為3個可能的vantage point。r為點到交界線的距離(弧線的半徑)，b為交界線的長度。假設這個區域上的點分布是uniform分布的話，意思是每個地方出現的機率都相同，那如果有一條線的長度較另外一條長，我們可以知道這條比較長的線，有比較大的機率會出現比較多點在線上。Vantage point主要的概念就是希望在交界的地方，能夠出現越少點越好，由以上的推論即可知道要交界的那條線的長度越短越好。所以在這個圖2.7的例子中，我們普遍的做法是畫一個圓分成內外面積一樣的兩部分，交界線的長度是最長的，但是在vantage point的挑選上卻是這三個點中最差的。在這個例子中最好的vantage point是右上角的P_c。所以我們建立二進位樹的方法，就是一次找一個惡意軟體當vantage point，用剛剛的原則去找，找到以後可以把空間切分成弧線內部跟外部，然後分別在去做一樣的事情，就是找弧線內部的vantage point跟弧線外部的vantage point。這樣一直遵循一樣的原則去找vantage point以及切分區域，最後就能建立出一個二進位樹，

我們把左邊的child定義成弧線內部的區域，右邊的child定義成弧線外部的區域，而點跟點之間的距離，我們是用(1-相似度)去表示，因為相似度為1的時候表示向量最像(近)。

兩個特徵向量的相似度，跟向量之間的距離有關，因為運算複雜度的考量，使用了1-norm而不使用2-norm的方法來計算向量之間的距離。假設有一個向量為 $f_1=[f_{11},f_{12},...,f_{1n}]$，一個向量為 $f_2=[f_{21},f_{22},...,f_{2n}]$，則這兩個向量的相似度為 $s=1-\frac{d(f_1,f_2)}{|f_2|}$。在這個例子中，$f_2$為要偵測的程式的特徵向量，$f_1$為資料庫中的某個惡意軟體的特徵向量。其中 $d(f_1,f_2)=\sum_{i=1}^{n}|f_{1i}-f_{2i}|$，$|f_2|=\sum_{i=1}^{n}|f_{2i}|$。

(4)臨界值挑選

臨界值的挑選，會直接影響到誤判率，臨界值定的越低，能夠偵測到同一個家族中較多的變種，但是相對的誤判率(正常的被判斷成惡意)也會提高，所以在偵測前就要謹慎選擇臨界值。誤判會讓使用者一堆正常的程式都無法使用，但是偵測變種能力如果不好，則資料庫中就必須儲存比較多的惡意軟體，所以臨界值選取的太高或是太低都不是很好。其中臨界值跟誤判率的關係可以由下表2.1來說明。

相似度(s)	在各個範圍的個數	累積個數
$0.0\le s<0.1$	X_0	W_0
$0.1\le s<0.2$	X_1	W_1
$0.2\le s<0.3$	X_2	W_2
$0.3\le s<0.4$	X_3	W_3
$0.4\le s<0.5$	X_4	W_4
$0.5\le s<0.6$	X_5	W_5

$0.6 \le s < 0.7$	X_6	W_6
$0.7 \le s < 0.8$	X_7	W_7
$0.8 \le s < 0.9$	X_8	W_8
$0.9 \le s < 1.0$	X_9	W_9
$s = 1.0$	X_{10}	W_{10}

表2.1 惡意軟體跟正常程式的相似度個數統計圖(一)

假設我們拿Y_1個惡意軟體跟Y_2個正常的程式去做訓練，則在惡意軟體跟正常的程式間總共會有$Y_1 * Y_2$個相似度，若相似度為$0.0 \sim 1.0$，$X_0 \sim X_{10}$為各個相似度範圍所出現的個數，則$Y_1 * Y_2 = X_0 + X_1 + ... + X_{10}$。另外在定義累積個數的參數$W_0 \sim W_{10}$，其中$W_i = \sum_{k=0}^{i} X_k$，$i = 0 \sim 10$。若$Y_1 * Y_2 = W$，且臨界值=$0.1 * i$，則誤判率$\le (1 - \frac{W_i}{W}) * 100\%$。

或者我們可以舉一個更普遍簡單的例子，如表2.2。

相似度s(臨界值為T)	各個範圍的個數
$s < T$	X_1
$s \ge T$	X_2

表2.2 惡意軟體跟正常的程式的相似度個數統計圖(二)

在這個例子中，臨界值=T，則誤判率$\le (\frac{X_2}{X_1 + X_2}) * 100\%$。

誤判率會小於或等於(不一定等於)上述所計算的值的原因，是由於在那些統計被誤判的次數裡，有可能一個正常的程式被誤判了兩次以上，即跟兩個以上的惡意軟體相似度大於或等於臨界值。但是在誤判率中實際上只算這一個正常的程式被誤判，而不是算兩次，誤判率$= \frac{\text{正常程式被判斷成惡意軟體的個數}}{\text{正常程式個數}} * 100\%$。

2.4.2 偵測時的工作

逆向工程得到原始碼、產生control flow graph、將control flow graph轉換成字串、將字串轉換成特徵向量。這幾個動作不只在偵測時要做，在挑選基底之前要先做完前三個動作。而前四個動作不只有在偵測時需要，在建立惡意軟體資料庫之前要先得到惡意軟體的特徵向量，以及挑選臨界值時要計算相似度之前也要先產生正常程式跟惡意軟體的特徵向量，所以我認為這四個動作是整個系統最基本也是最重要的地方。有了待偵測程式的特徵向量之後，將其置入二進位樹中，並從root開始往下搜尋，最後找到跟其相似度最高的向量，並且算出其相似度的值，再跟所挑選的臨界值去比較大小，若是小於臨界值則判定為正常程式，反之若是大於或等於臨界值則判定為惡意軟體。

第三章

曼哈頓距離預先濾波器

3.1 動機及問題描述

在逆向工程之後，我們要將程式的原始碼先切分成許多basic blocks，再經過Androguard的文法轉換表轉換成一組字串來代表這個程式的所有特性。我們在參考的Androguard工具中發現其文法轉換表，有幾個參數是沒有用到的，也有些參數我們覺得可以用更好的方式去表示。在擷取特性的時候，我們希望能夠在固定的長度字串(4-gram)中，包含更多的訊息，因此我們會將一些多餘的訊息濾除，並且將原本長度為2的參數改為長度為1的參數(因為其實只有一個參數)。

產生字串之後就是要建立向量的基底，然後再將字串轉換成向量。參考的論文中所提到的，將惡意軟體跟正常程式中出現次數前500多的這些子字串，當作程式的特性來建立基底，我們覺得應該有更好的做法，而不是直觀的取出現次數最多的當作特性。我們覺得應該要觀察正常程式跟惡意軟體之間的子字串出現次數的差距大小，而不是只統計哪個特性出現比較多次就比較重要。我們覺得在正常程式跟惡意軟體之間的子字串出現次數差距越大，表示這種特性，比較能夠區分出是正常程式，或是惡意軟體。

有了基底之後，我們可以進一步建立程式的特徵向量。向量在數學上有很多優點，可以計算距離，可以計算角度，也可以計算平均值、變異數等數學統計特性。我們覺得應該可以用一些數學特性來表示程式，並不完全要使用相似度的方法來跟資料庫之中程式的向量做比較。我們覺得若是能夠找到一種模型來描述這個向量，那我們能夠更正確地去代表這個程式的特性，可以減少比對的複雜跟時間。

3.2 系統設計

我們發現若是使用向量的曼哈頓距離(Manhattan distance)，可以將許多相似(相似度大於或等於臨界值)的程式分群，但也有可能分到不相似(相似度小於臨界值)的程式而曼哈頓距離卻很相似(圖 3.1)。但我們可以確定的是，如果這兩個程式很相似，那它們的曼哈頓距離會很相似(圖 3.1)，即曼哈頓距離的值會收斂在一個範圍之中。而這個範圍的最大值以及最小值會跟這一群程式彼此之間的2種相似度有關，我們會定義一種新的相似度，曼哈頓距離相似度，主要是要計算程式之間曼哈頓距離的相似程度，並且會跟原本參考的論文中定義的相似度做連結。利用兩種相似度之間的關係找到每一群曼哈頓距離的範圍最大值與最小值，然後可以做成預先濾波器，使得偵測時在建立完向量之後，先經過曼哈頓距離預先濾波器，這時會判斷向量落在哪一群曼哈頓距離範圍內，然後去那一群之中搜尋。因此我們也需要先建立每一群曼哈頓距離範圍之中的二進位樹，在判斷完是哪一群之後就可以去那一個特定的樹做搜尋了。因此我們在實現曼哈頓距離預先濾波器的時候，必須要確定每一群曼哈頓距離的範圍最大值與最小值，以及每一群之中所包含的成員有哪些惡意軟體，然後將這些成員依照vantage point tree的方法建立成二進位樹。

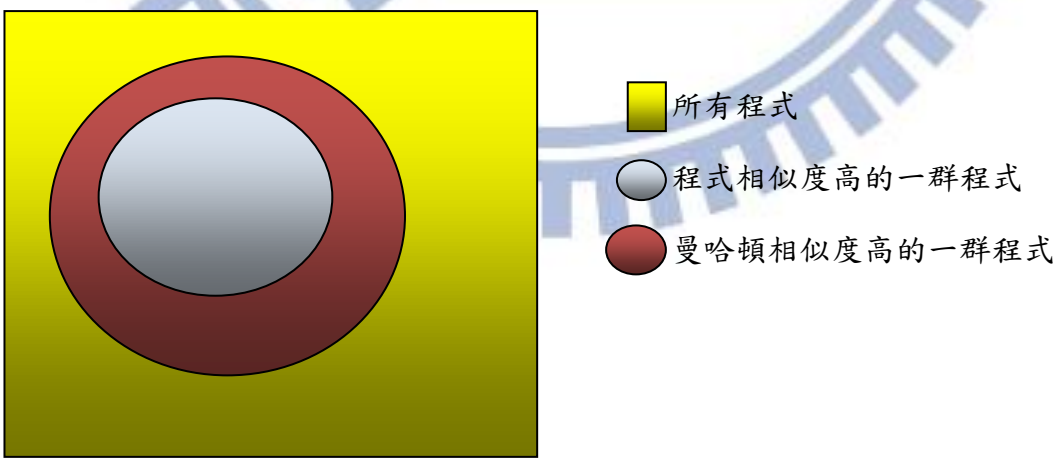


圖3.1 曼哈頓距離相似度跟程式相似度的關係

所以我們主要的設計是透過偵測的程式特徵向量的曼哈頓距離，先濾除掉與偵測的程式曼哈頓相似度低的程式，但是不濾除程式相似度高(大於或等於臨界值)的那些程式。在整個偵測系統中，大部分跟第二章所提的架構相同，有幾個不同的地方分別為(1)q-gram資訊擷取、(2)刪減基底的方法、(3)曼哈頓距離預先濾波器、(4)依照曼哈頓距離區分的二進位樹。因此我們在接下來的小節，會注重在這幾個不同的地方，其他相同的地方就不再詳加說明，圖 3.2 為系統架構圖。

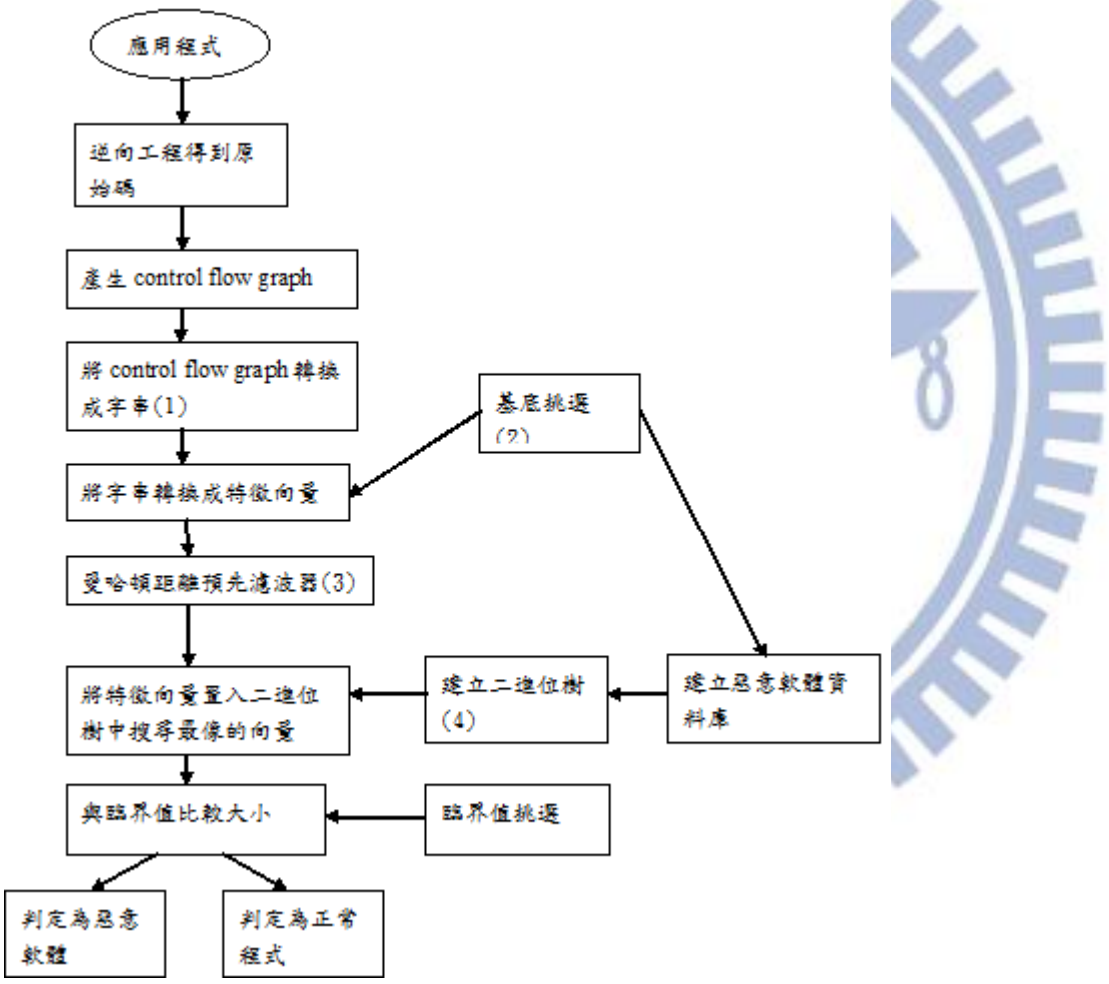


圖3.2 系統架構圖

(1)q-gram資訊擷取

在一個程式經過逆向工程得到原始碼，經過basic block的定義切分成basic blocks，再轉換成字串。我們使用q-gram的方法，從這組字串中擷取所有連續長度為q的子字串當作這個程式的所有特性。q-gram是一個計算統計與機率領域的一種方法，舉著名的例子，例如蛋白質序列(protein sequence)、以及DNA序列(DNA sequence)都是使用q-gram的方法擷取特性。q-gram model是一個機率類型的語言模型，預測在序列中的下一個項，屬於(q-1) order 的馬可夫模型。我們可以把q-gram的方法，解釋成已知在q-gram序列中的前q-1個項，那麼預測下一項(第q項)會出現什麼，即為這個序列的意思。我們透過這些序列出現的次數多寡，可知道其機率的高低。這些序列分別代表程式中不同的特性，最後可以將一個程式用這些不同的特性組合而成。在這個部分我們提出一些我們的想法，我們覺得既然是在固定的長度下去擷取一段資訊當作某種特性，那這些資訊應該是要越多有用的資訊越好，如果有某些項是可以透過前一項或是其他項就可確切得知，那麼這些項相對地就不那麼有用，我們的目的在於找出這些較無用的項，並將其刪除，使得q-gram取得的特性能夠更有用。

首先，我們舉個例子，在文法轉換表(圖2.2)中，當遇到basic block的開頭時會將其轉換成一個” B[“ ，然後後面在接basic block裡面所出現的指定參數(例如”String”->”S”)，在basic block結束的地方，再指派一個”] ”，所以我們可以得到” B[S] ”這個字串。乍看之下似乎沒什麼問題，但是仔細一看發現光是要描述一個basic block，” B[ ] ”就佔了長度3，因此我們覺得這是其中一個不太有用的一串資訊。在這個長度為4的” B[S] ”字串中，我們只能得到兩個資訊，第一個就是這是在”一個basic block(B[])中”，第二個就是”這一個basic block中有一個String(S)”。在字串的部分沒有問題，問題是在於B[]中，長度雖然為3，但是其實只有一個資訊要表達，因此我們覺得” B[] ”應該要改成” [“就可以明確地表達”在一個basic block中”這個訊息，這是在一個basic block

中，當遇到下一個”[”的時候，就表示要進入”下一個basic block”了。在我們修改完之後，原本的”B[S]”變成”[S”，長度由4變為長度為2的序列，所以可以再多擷取兩個額外的資訊進去序列，如此有達到我們的目的，即在固定的長度下，擷取越多有用的資訊越好。

我們再舉第二個例子，在文法轉換表(圖 2.2)中，當遇到Field的時候，我們會將其轉換成”F0”或是”F1”，但是其實不論是”F0”或是”F1”，實際上都只有一個資訊，就是” 有一個Field ”，所以我們覺得若是將其長度為2的參數，改成只用長度為1的參數去表示，例如F0改成用F表示，F1改成用f表示，那麼同樣是表達一個資訊，我們可以減少長度1，可以使得在序列中又多1個資訊可以擷取。其他一樣若是只有表達一個資訊的參數，但是其長度為2，我們也會將其改為新的參數(Package PackageCall(PM->C); Package PackageNew(PC->N))，且長度改完以後為1，目的跟先前說明的一樣，要在固定的長度內擷取到更多的資訊，不要擷取那些多餘的項放入序列中。另外我們發現在androguard中並沒有使用到Exception、Number、Id，所以我們也將其從文法表中濾除。我們將文法轉換表改成下面圖 3.3，目的就是要刪除那些多餘的項，使得在固定長度q-gram可以擷取到更多資訊來當作特性。

```
Procedure : := StatementList
StatementList : := Statement | Statement StatementList
Statement : := BasicBlock | Return | Goto | If | Field | Package | String
Return : := 'R'
Goto : := 'G'
If : := 'I'
BasicBlock : := '['
Field : := 'F' | 'f'
Package PackageCall: := 'C'
Package PackageNew: := 'N'
String : := 'S'
```

圖3.3 修改後的文法轉換表

(2)刪減基底的方法

我們在擷取特性完以後，要先找出向量的基底，才能夠將字串轉換成向量，這一小節除了說明如何找到基底以外，也會說明我們所提出的減少基底的方法，使得偵測更有效率。首先說明如何找到所有基底，我們先拿400個惡意軟體，以及368個正常程式，先對每一個程式都建立control flow graph，然後都轉換成字串，在用上一個小節所提的找出所有q-gram的序列當作特性。我們找到在這768個程式裡，出現過不同的特性(q-gram序列)，每一個不同的特性就是向量中的一個基底。但是基底的數量太多會造成運算速度緩慢，所以我們覺得要怎麼減少基底但卻不會影響到偵測的結果，是一個重要的問題。在參考的論文裡，作者使用出現次數最多的500個基底當作刪減後剩下的基底。雖然這很直觀，出現次數越多表示這種特性在所有程式都出現機率越大，表示每個程式都有可能會有這種特性，所以作者認為普遍出現在每一個程式中的那些特性，被拿來當作基底比較適合。

我們的想法是既然我們要區別正常程式跟惡意軟體，那我們何不統計在每一種特性中的差異大小，我們覺得差異越大的那些特性，拿來當作基底會比較能夠區分惡意軟體跟正常程式。因此我們去統計了每種特性在惡意軟體及在正常程式之間的差異大小，然後將差異較小的基底刪去。在刪減完基底以後，我們另外去做了一個測試(圖 3.4)，測試如果只用惡意軟體(malwares)建立基底(basis 2)，或是只用正常程式(benign apps)建立基底，會不會跟用惡意軟體跟正常程式(malwares and benign apps)建出來的基底(basis 3)不同？結果發現建出來的基底相同。因此我們用正常程式跟惡意軟體去建立基底，然後再依照差異大小去刪減差異較小的基底。但是有一個很重要的問題，就是要刪減多少個基底，但是卻不會影響到偵測結果。我們接下來會介紹如何刪減最多基底，但卻不影響偵測結果的方法。

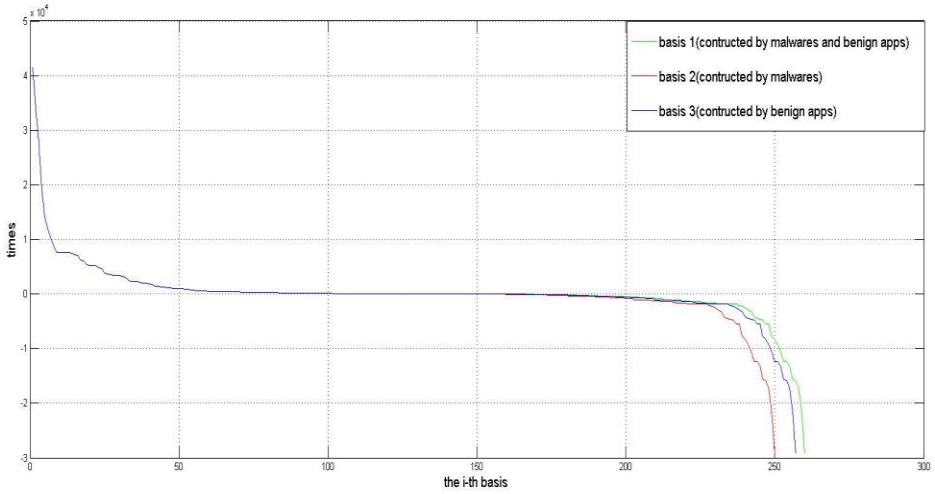


圖3.4 用三種方式建立基底(刪減後)

先用上述的方法產生400個惡意軟體以及368個正常程式的特徵向量(n個基底)。假設第i個惡意軟體的特徵向量為 $m_i = [m_{1i}, m_{2i}, ..., m_{ni}], \forall i = 1,2,...,400$，第j個正常程式的特徵向量為 $b_j = [b_{1j}, b_{2j}, ..., b_{nj}], \forall i = 1,2,...,400$，則我們使用下列幾個步驟來刪減基底。

在 $\sum_{j=1}^{368}\sum_{i=1}^{400}\left|m_{1i} - b_{1j}\right|$ 這個式子中，代表第一個基底出現次數在所有惡意軟體跟正常程式間的差異大小。因此我們計算 $v_k = \sum_{j=1}^{368}\sum_{i=1}^{400}\left|m_{ki} - b_{kj}\right|, \forall k = 1,2,...,n$，將n個基底的出現次數在惡意軟體及正常程式間的差異大小計算出來。算出 v_k 值以後，將其依照值的大小排列出來，由大排到小，假設我們得到 $w_1, w_2, ..., w_n$，其中 $w_1 \geq w_2 \geq ... \geq w_n$。因此我們現在都可以用這組依照惡意軟體跟正常程式的出現次數差異大小的順序建立的基底來建立特徵向量。我們將基底在特徵向量中的順序依照其 w 值排序，然後再將那400個惡意軟體跟368個正常程式的特徵向量依照這個新的順序建立。接下來要介紹如何刪減差異性較小的基底，即比較不會影響判斷惡意軟體跟正常程式差異的基底。我們在第二章的時候有介紹過如何挑選臨界值使得滿足誤判率在一個定值以下，因此在這裡我們要用到這個觀念，我們的要求是想要刪減基底之後仍能滿足誤判率維持一樣的定值。因此我們要先做一些初始化的動作，我們在刪減基底之前，先將臨界值跟誤判率的關係去統計出來，

然後我們先假設我們想要的誤判率是在一個值以下，那我們就取誤判率等於那個定值時的臨界值是多少，我們將這個臨界值記錄下來。由於不知道一次要刪減多少個基底仍能滿足要求，所以我們一次刪除一半那些差異較小的基底。在刪除完一半之後，我們會去利用剛剛記錄的臨界值去統計誤判率，若是誤判率仍然跟剛剛的定值相同，則表示還可以再刪除基底，若是較剛剛的定值為低，則表示刪除太多基底，需要在加一些基底回去。因此在第二次的時候，會變成增加一半的一半或是減少一半的一半，即增減四分之一(若是非整數，則無條件捨去小數點後的數字，因基底越少越好)，以此類推，直到最後一次只有增減1個基底以後，且滿足我們的誤判率要求才停下來。因此我們需要做 $\log_2 n$ 次可以找到最佳的基底值，使得刪減基底個數為最多，但是卻不影響我們的誤判率要求。

(3)曼哈頓距離預先濾波器

我們在前兩個小節已經有介紹過如何讓q-gram的效率更好，如何讓基底的個數減少但卻不影響誤判率，在這一小節中，我們要介紹如何利用向量的曼哈頓距離，來將資料庫中的惡意軟體分群，並且使得在偵測時可以透過我們的曼哈頓距離預先濾波器，找到特定的一群惡意軟體，而不用對全部的惡意軟體做搜尋比對。我們在這一個小節中，會先介紹曼哈頓距離相似度跟參考論文的相似度之間的關係，然後介紹如何找到每一群惡意軟體所屬曼哈頓距離的範圍，並且說明每一個範圍內如何找到有哪些惡意軟體成員。假設有一個向量為 $f_1=[f_{11}, f_{12},..., f_{1n}]$，一個向量為 $f_2=[f_{21}, f_{22},..., f_{2n}]$，則這兩個向量的相似度為 $s=1-\frac{d(f_1,f_2)}{|f_2|}$。在這個例子中，$f_2$ 為要偵測的程式的特徵向量，f_1 為資料庫中的某個惡意軟體的特徵向量。其中 $d(f_1,f_2)=\sum_{i=1}^{n}|f_{1i}-f_{2i}|$，$|f_2|=\sum_{i=1}^{n}|f_{2i}|$。上述是參考論文所定義的程式之間相似度的式子，而我們所提出的曼哈頓距離相似度，是將 $d(f_1,f_2)$ 的值修改成 $d_M(f_1,f_2)=\left|\sum_{i=1}^{n}f_{1i}-\sum_{i=1}^{n}f_{2i}\right|$，其中 $\sum_{i=1}^{n}f_{1i}=Md_{f_1}, \sum_{i=1}^{n}f_{2i}=Md_{f_2}$，$Md$ 為曼哈頓距離，因此我們可以將原式改成如下等式：

$$s_M = 1 - \frac{d_M(f_1, f_2)}{|f_2|} = 1 - \frac{\left|\sum_{i=1}^{n} f_{1i} - \sum_{i=1}^{n} f_{2i}\right|}{\sum_{i=1}^{n} |f_{2i}|} = 1 - \frac{|Md_{f_1} - Md_{f_2}|}{Md_{f_2}}$$

s 為參考論文中定義的相似度，而 s_m 為我們定義的曼哈頓距離相似度，s_M 在某種情況下，會剛好等於 s 的值，就是在 f_1 所有的分量都小於或等於 f_2，或是 f_1 所有的分量都大於或等於 f_2。因為滿足這種情況我們可以從 $d(f_1, f_2) = \sum_{i=1}^{n} |f_{1i} - f_{2i}|$ 推出 $d(f_1, f_2) = \sum_{i=1}^{n} |f_{1i} - f_{2i}| = \left|\sum_{i=1}^{n} f_{1i} - \sum_{i=1}^{n} f_{2i}\right| = d_M(f_1, f_2)$。但是並不是所有的 s_m 都會跟 s 相同，只有上述那種情況例外，因此我們想要利用這兩個相似度之間的關係來建立每一個惡意軟體的曼哈頓距離範圍，若是有程式的曼哈頓距離落在在這個範圍之內，則我們會說這個程式可能是這個惡意軟體的變種，但不一定是變種，因為曼哈頓距離相似度高，程式相似度有可能低。但我們可以確定的是程式相似度高，則曼哈頓距離相似度高。即曼哈頓距離相似度高為程式相似度高的必要條件，但不為其充分條件。

下面將介紹我們如何在每一個惡意軟體中挑選一個曼哈頓距離相似度最小值出來，並且使用這個曼哈頓距離相似度最小值來計算出這一個惡意軟體的曼哈頓距離範圍的最大值跟最小值。我們首先會算出在惡意軟體中，任兩個互相的程式相似度以及曼哈頓距離相似度(如圖 3.5)。我們在這裡可以看成是假設固定一個惡意軟體(q)，則其他惡意軟體(p)是否為其變種，若是程式相似度有大於或等於我們先前算出的臨界值(在這假設臨界值為0.8)，則我們說這些惡意軟體為這個固定的惡意軟體的變種，我們有興趣的是這些變種之中，曼哈頓距離相似度會是多少。我們想要利用這兩種相似度之間的關聯性，來建立一個曼哈頓距離預先濾波器，使得被偵測的軟體雖然不知道跟所有惡意軟體的程式相似度是多少，但可以從曼哈頓距離的值，找到那些跟其曼哈頓距離相似度高於一定值的惡意軟體。所以我們要找到對每一個固定的惡意軟體而言，會有一個曼哈頓距離相似度的最小值，在其他惡意軟體跟其之間。

(q,p)	程式相似度	曼哈頓距離相似度
(1,1)	1.000	1.000
(1,2)	0.747	0.828
(1,3)	0.823	0.808
(1,400)	0.843	0.865
(2,1)	0.656	0.721

圖3.5 惡意軟體之間的兩種相似度

我們的想法是在固定一個惡意軟體下，統計那些跟其程式相似度大於或等於臨界值的成員，且找到在他們之中跟固定的惡意軟體的曼哈頓距離相似度的最小值(如圖3.6)，所以我們在每一個固定的惡意軟體之中各會找到一個曼哈頓距離相似度最小值，這個最小值的意義是在曼哈頓距離預先濾波器，若是被偵測的程式跟某惡意軟體的曼哈頓距離相似度小於此惡意軟體所找到的曼哈頓距離相似度最小值，則我們會說此惡意軟體會被濾除掉，因為曼哈頓距離相似度太低了，程式相似度更不可能會大於或等於臨界值，所以我們會把此惡意軟體濾除掉。在圖3.6中，要先找到紅色框框的這些成員，這些成員代表跟這個固定的惡意軟體的程式相似度大於或等於臨界值(在此假設為0.8)，並且在這些成員中，在他們相對應的曼哈頓距離相似度中找出最小值。對每一個固定的惡意軟體(q)而言，都可以找到一個最小值，而藍色框框裡的代表的是，雖然曼哈頓距離相似度大於或等於其最小值，但是其程式相似度小於臨界值的成員，表示我們沒有辦法直接經由曼哈頓距離判定為變種，但是那些程式相似度高於臨界值的也不會被濾除掉，我們只會濾除掉那些曼哈頓距離相似度低的(小於最小值)。

圖3.6 找出曼哈頓距離相似度最小值

但是我們並不想要在偵測的時候才去計算跟每個惡意軟體的曼哈頓距離相似度，我們想要先將惡意軟體分群，然後偵測時可以經由被偵測程式的曼哈頓距離，先判斷為其屬於哪一群，再去預先建立好的那一群成員所建立的二進位樹做搜尋。我們會對每一個惡意軟體，依照剛剛所說的各自的曼哈頓距離相似度最小值來對每一個惡意軟體都計算出一個曼哈頓距離範圍，若是有程式的曼哈頓距離落在這個範圍內，表示為其變種。因此我們可以透過曼哈頓距離相似度最小值來找到該惡意軟體的曼哈頓距離範圍，這些範圍的最大值跟最小值可以從剛剛得到的曼哈頓距離相似度最小值所計算出來。我們假設剛剛得到的曼哈頓距離相似度最小值為 $s_{M,\min}$ ，且範圍的最大值為 R_{Max} ，最小值為 $R_{\min}$ ，則我們可以經由簡單的數學推導得到 R_{Max} 跟 $s_{M,\min}$ 的關係、 $R_{\min}$ 跟 $s_{M,\min}$ 的關係。

首先我們先推導 R_{Max} 跟 $s_{M,\min}$ 的關係：

$$s_{M,\min} = 1 - \frac{\left| Md_{f_s} - Md_{f_t} \right|}{Md_{f_t}}$$

$$= 1 - \frac{\left| R_{Max} - Md_{f_t} \right|}{Md_{f_t}} \qquad (Md_{f_s} = R_{Max})$$

$$= 1 - \frac{R_{Max} - Md_{f_t}}{Md_{f_t}} \qquad (\because R_{Max} \geq Md_{f_t})$$

$$\Rightarrow R_{Max} = (1 - s_{M,\min}) * Md_{f_t} + Md_{f_t}$$

$$= (2 - s_{M,\min}) * Md_{f_t}$$

同理我們再推導 $R_{\min}$ 跟 $s_{M,\min}$ 的關係：

$$s_{M,\min} = 1 - \frac{\left| Md_{f_s} - Md_{f_t} \right|}{Md_{f_t}}$$

$$= 1 - \frac{\left| R_{\min} - Md_{f_t} \right|}{Md_{f_t}} \qquad (Md_{f_s} = R_{\min})$$

$$= 1 - (-\frac{R_{\min} - Md_{f_t}}{Md_{f_t}}) \qquad (\because R_{\min} \leq Md_{f_t})$$

$$= 1 + \frac{R_{\min} - Md_{f_t}}{Md_{f_t}}$$

$$\Rightarrow R_{\min} = (s_{M,\min} - 1) * Md_{f_t} + Md_{f_t}$$

$$= s_{M,\min} * Md_{f_t}$$

因此我們知道只需要計算出每一個惡意軟體的曼哈頓距離（Md_{f_q}），以及惡意軟體任兩個之間所找到的曼哈頓距離相似度最小值（$s_{M,\min}$），我們就能夠找到對此惡意軟體而言，有可能為其變種的曼哈頓距離範圍（$R_{\min}$，R_{Max}），其中 $R_{\min} = s_{M,\min} * Md_{f_q}$，$R_{Max} = (2 - s_{M,\min}) * Md_{f_q}$，我們可以對每一個惡意軟體都找到一個存在這樣關係的曼哈頓距離範圍(如圖3.7)。

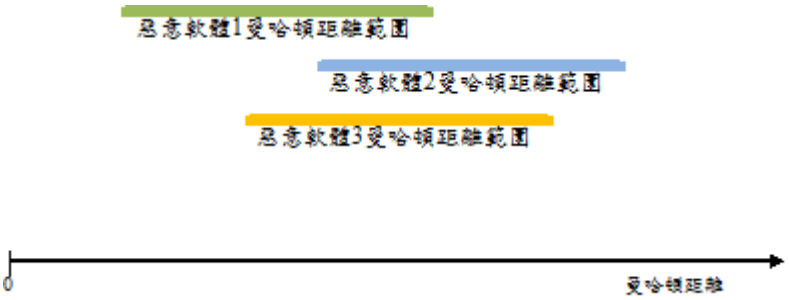


圖3.7 曼哈頓距離範圍

但是只有這些範圍還不能達到我們的目的，我們的目的是只利用曼哈頓距離的值就能決定有哪些惡意軟體跟偵測程式曼哈頓距離相似度高(大於或等於最小值)，而不是還要去跟每一個惡意軟體計算曼哈頓距離相似度，看是否有落在此範圍內。因此我們必須從這些範圍，去統整成另一個有用的資訊，即使用曼哈頓距離區分許多群，而同一群的曼哈頓距離會在一個範圍之內，不同群各自會有一個範圍，這些範圍的值可以從剛剛算出每個惡意軟體的曼哈頓距離範圍所交集得到。並且我們將所有惡意軟體的範圍都列在一個數線上，我們想要得到由這些曼哈頓距離範圍取交集以後，所得到的新的分群的結果。在這個分群中，我們可以知道在每個曼哈頓距離範圍之中，有哪些惡意軟體的成員，即在偵測的時候，通過曼哈頓距離預先濾波器還沒有被濾除的惡意軟體。我們想要在偵測之前就先建立好每一群有哪些惡意軟體成員，還有每一群的曼哈頓距離範圍的最小值跟最大值，另外值得一提的是有些曼哈頓距離的範圍，裡面會沒有惡意軟體成員，意思若是偵測時，被偵測的程式的曼哈頓距離落在這些範圍內，可以直接知道其為正常程式，就不需要再麻煩地去做搜尋了。我們可以從圖 3.8 得知，我們將每個惡意軟體的範圍依照大小排列在一數線上，可以得到彼此交集的部分，舉例來說，若是有一區域中(第四群)含有惡意軟體1、2、3的範圍，則此區域的成員即為惡意軟體1、2、3。而在第一群跟第七群中，因為沒有跟任何惡意軟體的範圍交集，所以假如有曼哈頓距離落在這兩個區域，則我們判定其為正常程式。

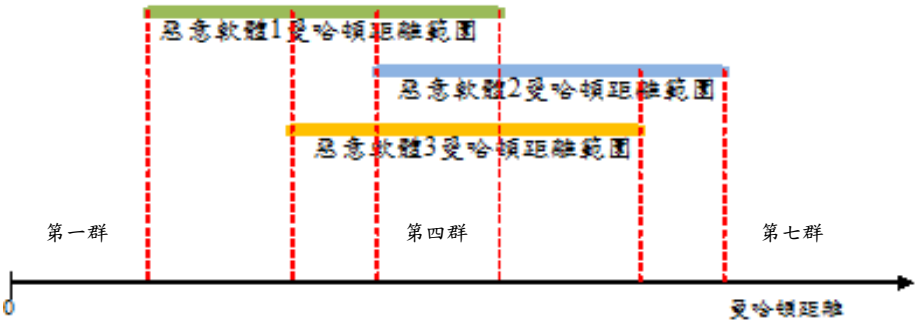


圖3.8 曼哈頓距離分群

總結上面所介紹的方法，從惡意軟體之間的兩種相似度關係，建立出以曼哈頓距離所區分的不同群惡意軟體，這些群可以在偵測的時候，由被偵測的程式的曼哈頓距離得知其屬於哪一群，若是那一群並沒有惡意軟體成員(如圖3.8的第一群跟第七群)，則直接判定為正常程式；反之，若是那一群有惡意軟體的成員，則我們會做進一步的搜尋，搜尋的方法以及搜尋的二進位樹將在下一小節中介紹。

(4)Vantage Point Tree

Vantage Point Tree(vp tree)是一種二進位空間區分的樹(Binary Space Partitioning)，會依照空間中的位置先選出許多參考點(Vantage Point)，然後將資料分成兩區：會有一個臨界值存在於一區域內，假如資料是離參考點比較近(小於臨界值)，則此資料會被分到比較近的那一區；反之，若是離較遠(大於臨界值)，則將其分到比較遠的那一區。簡言之，在一個區域內會有一個臨界值以及一個參考點來判斷資料跟參考點是該被分到近的那一區還是遠的那一區，依此類推，可以將所有的資料建立成一個二進位樹。而要決定空間中是比較遠還是比較近，要先考慮兩個資料間的距離怎麼計算，因此我們定義兩個資料的距離=兩個資料的特徵向量相似度，相似度越小的代表距離越遠，相似度越大的代表距離越近。而為了計算上的方便，以及利用一些數學的特性來表達彼此之間的遠近

關係，會將特徵向量改成在度量空間中，定義依照參考點的不同其他點跟其遠近關係，下面先介紹度量空間，然後再介紹參考點跟其他點的距離關係。

度量空間的定義在數學中，度量空間是一個集合，在其中可以定義在這個集合的元素之間的距離（叫做度量）的概念。若用數學特性來表示(圖 3.9)，

度量空間是元組 (*M*,*d*)，這裡的 *M* 是集合而 *d* 是在 *M* 上的度量(metric)，就是函數

$$d : M \times M \to \mathbb{R}$$

使得

1. *d*(*x*, *y*) ≥ 0 (非負性)
2. *d*(*x*, *y*) = 0 若且唯若 *x* = *y* (不可區分者的同一性)
3. *d*(*x*, *y*) = *d*(*y*, *x*) (對稱性)
4. *d*(*x*, *z*) ≤ *d*(*x*, *y*) + *d*(*y*, *z*) (三角不等式)。

圖3.9 維基百科定義度量空間的數學特性

因此我們要用度量空間的特性來表達特徵向量彼此之間在空間中的距離，以及其數學的特性。假設在向量空間中，定義(S,d)為一[0,1]固定區間的度量空間(metric space)，且假設p為參考點，p∈S，a、b為S中的任意兩點，則定義如下：

1. $\prod_p : S \to [0,1]$，其中 $\prod_p(a) = d(a,p)$

2. $d_p : S \times S \to [0,1]$，其中 $d_p(a,b) = \left|\prod_p(a) - \prod_p(b)\right| = \left|d(a,p) - d(b,p)\right|$

函數$\prod_p$可以被視為以p當作參考點，將S投影到[0,1]上。 換句話說，S被p視為到其距離d，而d的值在0到1。因此在上一小節所算出的程式相似度，要依照p的不同，以及依照其相似度大小給予0到1的值，其中0表示最近，所以給相似度最高的(相似度=1)，1表示最遠，給相似度最低的，其餘按照均勻(uniform)分佈給予[0,1]的值。函數d_p在特定情況下不是一個度量空間，若a,b為不同的兩點，但到p的距離相同，即$d_p(a,b) = 0$。但是其函數有對稱性以及滿足三角不等式的特性，所以稱其為偽度量(pseudo metric)。但因為d為一度量空間，所以可依其對稱性及三角不等式得到下面式子：

$d(a,b) \geq |d(a,p) - d(b,p)| = d_p(a,b)$。因此可以知道$d_p(a,b) \geq \tau \Rightarrow d(a,b) \geq \tau$。

可以利用以上特性，假使有一資料庫中的某個成員x，跟一個要被檢測的q，在搜尋的過程中，我們將不會對那些$d_p(q,x) \geq \tau$的x有興趣。

p為在資料庫（S_D）中的某些成員，現在考慮S_D以p為參考點在[0, 1]上的投影$\prod_P(S_D)$，且以中位數μ來將其劃分成兩個區域$[0, \mu)$、$[\mu, 1]$，其中第一區可以用在球面$S(p,\mu)$內來表示，另一區則是在此球面上或是球面外來表示。我們可以用S_{pL}及S_{pR}分別表示這兩個區域的度量空間，N_L及N_R分別表示在這兩個區域中的成員數目(即多少個點)。在一般普遍的情況下，可以將S_D均勻的分佈在S_{pL}及S_{pR}中，但是如果是有很多個點x到p的距離都剛好等於中位數（$d_p(x) = \mu$），那麼被分到S_{pR}的數目會遠多於S_{pL}（N_R遠大於N_L）。因此可以知道若是在球面上的點越少，就越有可能達到$N_R = N_L$，即$N_R \approx N_L$。假設$NN|_\tau(q)$為在搜尋時最靠近(nearest neighbor)q的那個點，其中τ為一變數，我們的目標是若$\prod_p(q) \geq \mu + \tau$，則可以在$S_{pR}$中找到$NN|_\tau(q)$，其中$d_p(a,b) \geq \tau \Rightarrow d(a,b) \geq \tau$所以可以不考慮$S_{pL}$中的點。或者如果$\prod_p(q) \leq \mu - \tau$，則可以在$S_{pL}$中找到$NN|_\tau(q)$，同理可以不考慮$S_{pR}$中的點。在最理想的情況下就是可以設定一個夠小的$\tau$，使得不可能發生$\prod_p(q) \in (\mu - \tau, \mu + \tau)$（圖3.10)的情況。

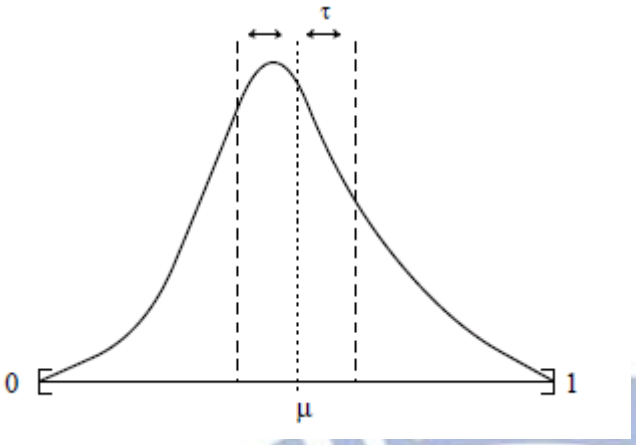


圖3.10 $\prod_p(S_D)$在[0,1]的連續密度分布

意思是我們要避免出現像圖3.10的情形，要設定好τ的值讓離q最近的點不會出現在$(\mu-\tau,\mu+\tau)$的範圍之中。因此我們可以用上述介紹的方法，使用中位數μ連續地劃分區域，並且每一個參考點(vantage point)可以劃分出兩個區域，而這個點的找法就是要讓其他點能夠越均勻分佈到被劃分的兩個區域中越好，其中在區域交界的地方要越少點越好(即$(\mu-\tau,\mu+\tau)$範圍中的點)，然後就可以將所有資料庫中的點都分佈到各自區域中，並且建構成一棵二進位的樹。而在其中樹的左邊內的點代表這區域內的點是在以參考點畫出的球面內，在樹的右邊內的點則是代表這區域內的點是在以參考點畫出的球面上或是球面外。我們可以依上述觀念建造出最基本的vp tree，而找參考點以及劃分球面內外區域的演算法將在下面幾頁中介紹。我們可以依照圖3.11的步驟來找到Vantage Point。其中Make_vp_tree()這個函數是用來劃分左區域(球面內)以及右區域(球面上及球面外)，而Select_vps()這個函數是要在每個區域中選擇最佳的vantage point。其中p表示參考點，mu為所有其他點跟p的距離之中的中位數，left為被劃分到左區域的點，right為被劃分到右區域的點、L代表左區域(小於mu)，R代表右區域(大於或等於mu)。

Algorithm 1 *Given set* S *of metric space elements; returns pointer to the root of an optimized vp-tree.*

```
function Make_vp_tree(S)
    if S = ∅ then return ∅
    new(node);
    node↑.p := Select_vp(S);
    node↑.mu := Median_{s∈S} d(p, s);

    L := {s ∈ S−{p} | d(p, s) < mu};
    R := {s ∈ S−{p} | d(p, s) ≥ mu};
    node↑.left := Make_vp_tree(L);
    node↑.right := Make_vp_tree(R);
    return node;
function Select_vp(S)
    P := Random sample of S;
    best_spread := 0;
    for p ∈ P
        D := Random sample of S;
        mu := Median_{d∈D} d(p, d);
        spread := 2nd-Moment_{d∈D} (d(p, d) − mu);
        if spread > best_spread
            best_spread := spread; best_p := p;
    return best_p;
```

圖3.11 找Vantage Point以及劃分區域的演算法

我們可以先看函數Make_vp_tree()，一開始先判斷S是否為空集合，因為若為空集合，則表示不用劃分區域也不用找參考點，因此回傳空集合。接下來若是不為空集合，則會先新建(new)一個節點(node)，這個節點即為待會找到的所有vantage point所建立的二進位樹的所有點，其中node會儲存p、mu、left、right等參數。宣告完node之後會去呼叫Select_vps()這個函數，P為隨機從S中選出來(但在我們的系統是從惡意軟體資料庫中選出)的參考點可能，D為隨機從S中選出來的點，用來算跟P的距離的中位數mu，並且在此有一個重要的參數spread，spread的值代表所有D中的點跟P中挑出的任一參考點的距離有沒有接近mu，若是接近則spread的值會較小，若是較遠則spread的值會較大。我們的目標是要找到p，滿足其spread最大，因此在找p的時候會一直更新spread的值，直到所有P中的p都找過之後，選取一個spread最大的p當作最佳的參考點。因此我們可以找到第一個參考點，在這個函數結束後回傳p的值(即p是哪一個點)，並且計算其他在S中的點跟p距離，找到距離的中位數mu且回傳給node，然後用mu來劃分左區域跟右區域，將S中的其他點依照跟p的距離，若是小於mu則劃分到左區域中，若是大於或等於mu則劃分到右區域中，如此將S中的點都分到L以及R中，且將左區域的點儲存在left中，將右區域的點儲存在right中，最後再回傳給node。因此在找完一個參考點之後，node

之中會儲存一個p(參考點)、區分區域的mu(其他點跟p的距離的中位數)、一群left(被分到左區域的點)跟一群right(被分到右區域的點)。

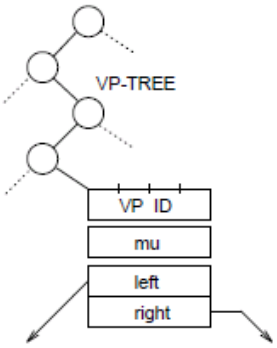


圖3.12 vp tree 簡單示意圖

我們可以從圖 3.12 中看到每一個參考點(vantage point)會儲存那個點的ID，以及其他點跟他的距離的中位數mu，而left裡面含有被分到左邊區域的那些點，right裡面含有被分到右邊區域的那些點，並且重複演算法做一樣的事情，找到在left裡面的參考點，然後一樣儲存在左邊區域中其他點到此參考點的距離的中位數mu，找到在right裡面的參考點，然後儲存在右邊區域中其他點到此參考點的距離的中位數mu。依此類推可以將所有的點建成一棵樹，而每個點都是一個參考點，會儲存上述提到的ID以及中位數mu。

但是這種方法只有儲存每一個參考點跟其區域的mu，我們沒辦法知道任意S中的點跟其參考點之前的參考點(追溯到root)的關係。因此有第二種演算法，參考點不儲存mu而會儲存其左區域跟右區域的lower bound跟upper bound(共4個值)，以及其之前的參考點的左區域跟右區域的lower bound跟upper bound。除了儲存每個區域的lower bound跟upper bound之外，還會將每個S中的點到其參考點的距離以及點到其參考點之前的參考點(追溯到root)的距離都儲存在hist中，即每個S中的點會各自儲存一個hist。這第

二種演算法在每個參考點會儲存將左區域的lower bound跟upper bound和右區域的lower bound跟upper bound合併過的總lower bound及upper bound，如圖3.13。

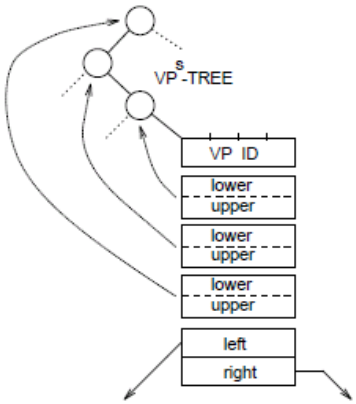


圖3.13 第二種vp tree示意圖

在圖3.13中可以看到參考點會儲存其之前的參考點的lower bound跟upper bound，並且一樣儲存left跟right，因此不僅可以知道區域中的點跟參考點的關係，也能知道跟參考點之前的參考點的關係。這種演算法主要的目的是能夠紀錄S中的點與其參考點以及之前的參考點的距離關係，所以hist是個很重要的參數，每個item(S中的點)會儲存一個hist，而hist中儲存點跟其之前每個參考點(追溯到root)的距離，item又會儲存另一個參數id，表示這個item是S中的哪一個點。在挑選p(參考點)時，使用第一種演算法中的函數Select_vp()，而其中傳入的參數為list，list初始化為所有S中的點，即從S中的點去尋找p。整個流程包含一個主要函數Make_vps_tree()以及一個子函數Recurse_vps_tree()，在這個子函數中又有兩個子函數，分別為第一種演算法中的函數Select_vp()，以及第二種演算法才有的合併左區域和右區域的bounds的函數Merge()。

Algorithm 2 *Given set* S *of metric space elements; returns pointer to the root of an optimized vp*^{8}*-tree.*

```
function Make_vps_tree(S)
    list = ∅;
    for s ∈ S
        new(item); item↑.id := ↑s; item↑.hist := ∅;
        add item to list;
    return Recurse_vps_tree(list);
function Recurse_vps_tree(list)
    if list = ∅ then return ∅
    new(node); node↑.p := Select_vp(list);

    delete node↑.p from list;
    for item ∈ list
        append d(p,item↑.id) to item↑.hist;
    mu := Median_{item∈list} tail(item↑.hist);
    L := ∅; R := ∅;
    for item ∈ list
        if tail(item↑.hist) < mu then
            add item to L, delete from list;
        else
            add item to R, delete from list;
    node↑.left := Recurse_vps_tree(L);
    node↑.right := Recurse_vps_tree(R);
    node↑.bnds := Merge(node↑.left↑.bnds,
                        node↑.right↑.bnds,node↑.p↑.hist);
    return node;
function Merge(range_list1,range_list2,value_list)
    "The two range lists are combined with
    the value list to yield a single range list
    which is returned."[5]
```

圖3.14 vp tree第二種演算法

我們可以看到在主函數中，先初始化item，包含item要儲存的id跟hist，並且將所有item都存入list中，因此一開始list中有S中所有的點的id，且每個點的hist都為空集合，且在主函數的最後一行呼叫子函數Recurse_vps_tree()。這個子函數主要在重複地做挑選參考點、算出mu切分區域、更新list跟hist的內容、以及合併bounds等工作。一開始先建立一個node，node中儲存由第一種演算法所找到的參考點，每找到一個參考點就會將其從list中移除以避免重複，所以每次抓完參考點list中剩下的點即為除了參考點之外在S中的點。然後會將這些剩下的點去計算跟其參考點之間的距離以及和參考點之前的參考點間的距離，然後儲存在每個item的hist中。在這裡mu的算法跟之前一樣，只是因為hist之中還有儲存之前的參考點，所以這邊會取tail，即取最近的那個參

考點，然後一樣算出list(除了參考點剩下的點)中的所有點跟這個參考點之間的距離的中位數當作mu。然後一樣用mu來劃分區域，若是點到參考點的距離小於mu，則分到左區域，反之則分到右區域。而分到左區域中時，會將左區域中的點儲存到其左區域的list中，將分配到右區域中的點儲存到其右區域的list中，然後可以重複地去劃分左區域跟右區域直到list中沒有點了為止。這邊跟第一種演算法不同的地方是會去額外計算每個區域的bounds，每個參考點的左區域的bounds跟右區域的bounds合併會變成這個參考點儲存的bounds。

在此總結這第二種演算法，比第一種演算法多了每個點跟其參考點以及之前的參考點間的距離"hist"，以及在每個參考點儲存回溯到root間每一個參考點的bounds(由其左區域跟右區域的bounds合併而來)。跟第一種演算法一樣用mu去劃分區域，但是參考點沒有儲存mu的值而是儲存bounds的值"bnds"，即對參考點而言，其左區域跟右區域的lower bounds跟upper bounds。這種方法多儲存了bounds的值，但是執行時間仍然可以維持O(nlog(n))。

3.3 貢獻

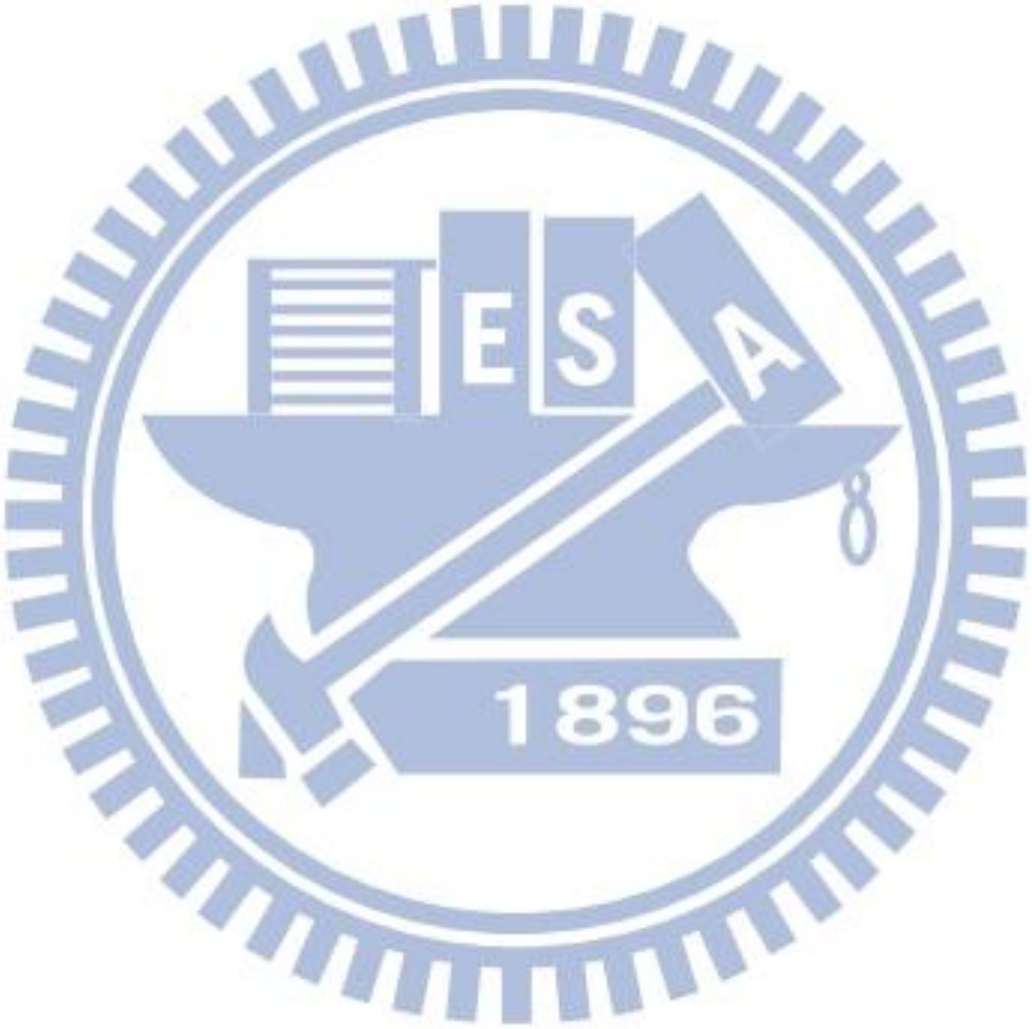
我們主要參考那篇論文的作法是將所有惡意軟體全部做成一棵vp-tree，所以只要一開始建立好這棵樹，就可以在搜尋時維持0(log(n))的時間。但是他有一個壞處，就是無論要偵測什麼軟體，都會經過這棵樹來搜尋，雖然較簡單但是會較浪費時間。所以我們利用特徵向量的曼哈頓距離去將惡意軟體依照其曼哈頓距離分成好幾群，而這些群中，同一群的曼哈頓距離相似度會高於我們所訂的臨界值，而這個臨界值的制定又會跟這群中的軟體之間的相似度有關，我們不能確保在這一群中，彼此都是相似度高於我們要判斷為惡意軟體的臨界值，但是我們可以確定那些相似度高於我們要判斷為惡意軟體的臨界值的那些可能為變種的軟體，不會被我們所濾除掉。

因此經過我們的曼哈頓距離預先濾波器之後，我們可以透過被偵測軟體的曼哈頓距離，知道要去哪一群的惡意軟體所建立的樹做搜尋。因此我們設計的優點是可以減少搜尋的次數加快搜尋的速度，並且依照其被偵測軟體的曼哈頓距離不同，去給予特定的樹做搜尋，而這些樹是可以事先建立好的。我們在偵測時只需要多計算被偵測軟體的曼哈頓距離並且判斷其屬於我們制定的哪一群，就可以去我們預先建立好的樹去搜尋。其中最大的優點，就是有些群是完全不用作搜尋就可以知道其為正常軟體，即這些群中沒有惡意軟體的成員建立的樹，所以當我們遇到被偵測的軟體其曼哈頓距離落在這些群時，我們完全不用去做搜尋的動作就可以知道其為正常軟體了。

在所有的軟體中，正常軟體的比重還是比惡意軟體來得多，所以如果大部分的軟體都可以一經過曼哈頓預先濾波器得知落在那些沒有惡意軟體的群，就可以直接知道其為正常軟體了。我們的方法雖然多了要計算其曼哈頓距離，以及判斷其落在哪一群中，但是這會比在搜尋時去計算跟惡意軟體之間的相似度來的快，因此我們會多做這個步驟，來減少後面搜尋時跟惡意軟體算相似度的次數，才可以節省大部分的時間。

我們提出利用特徵向量的曼哈頓距離可以將惡意軟體分群，並且我們覺得如果再多一些參數能夠描述這個特徵向量，那麼在未來有可能對每一隻惡意軟體都去建立其模型，而這個模型可能包含其曼哈頓距離、一些數學統計的參數，來描述這個軟體。用模型來表示軟體的話，可以大幅減少搜尋時的次數，節省大部分的計算時間，未來可以研究還有那些原因會決定一個軟體的特性，並且可以透過分析其特徵向量來得到這些特性。在理想狀況下，可以找到所有特性，將這些特性建立成一個模型，並且透過這個模型可以知道被偵測的軟體是否為惡意軟體，而不用去做搜尋。我們提出統計惡意軟體跟正常軟體之間的特性出現次數的差異，當作挑選基底的條件，可以使得基底的個數減少，且不影響原本判斷的正確性。我們提出不同基底給予不同的比重，可以讓相似度的計算更能區分惡意程式跟正常程式的不同，並且也能偵測到更多的變種。

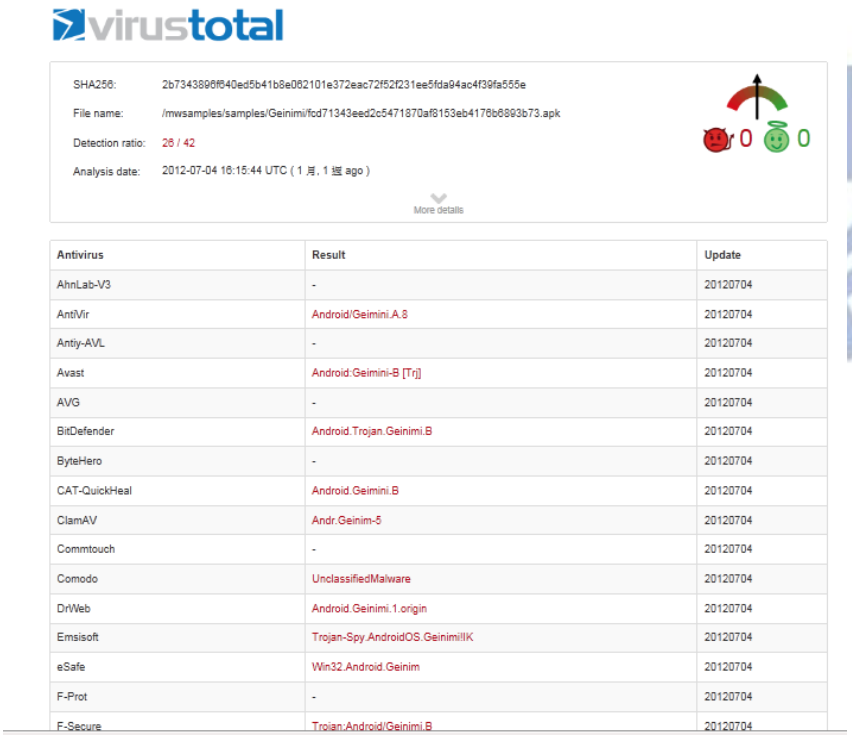
未來的研究可能會朝向建立特徵向量的方法，如何縮短其建立時間，以及如何找到更多數學特性來描述特徵向量最後建立成數學模型，還有如何調整特徵向量每個基底的比重，使得相似度的計算會更加準確，能夠讓同一個家族的惡意軟體相似度都很高，不同家族的惡意軟體相似度很低。



第四章

實驗結果與討論

我們在第三章的3.2(1)q-gram資訊擷取已經介紹過如何在一個有限的長度內去擷取有效的資訊，因此我們首先要來比較在使用我們的方法擷取資訊前後的差別。我們先在各個網站上面下載apk檔案，然後送至virustotal上面去確認其為正常程式或為惡意程式，其中virustotal為一個集合眾多知名防毒公司的線上掃毒，可以上傳檔案得到這些防毒軟體的掃描結果，我們將所有防毒軟體都認定是正常的程式當作我們的正常程式資料庫，而1/2以上的防毒軟體認定是惡意程式的程式當作我們的惡意程式資料庫。圖4.1為一檢測的例子，此apk在42個防毒軟體中，有被26個檢測到為惡意程式，因此我們將其放入惡意程式資料庫。



virustotal

SHA256: 2b7343896ff40ed5b41b8e062101e372eac72f52f231ee5fda94ac4f39fa555e

File name: /mwsamples/samples/Geinimi/fcd71343eed2c5471870af8153eb4176b6893b73.apk

Detection ratio: 26 / 42

Analysis date: 2012-07-04 16:15:44 UTC (1 月, 1 週 ago)

More details

Antivirus	Result	Update
AhnLab-V3	-	20120704
AntiVir	Android/Geimini.A.8	20120704
Antiy-AVL	-	20120704
Avast	Android:Geimini-B [Trj]	20120704
AVG	-	20120704
BitDefender	Android.Trojan.Geinimi.B	20120704
ByteHero	-	20120704
CAT-QuickHeal	Android.Geimini.B	20120704
ClamAV	Andr.Geinim-5	20120704
Commtouch	-	20120704
Comodo	UnclassifiedMalware	20120704
DrWeb	Android.Geinimi.1.origin	20120704
Emsisoft	Trojan-Spy.AndroidOS.GeinimiIIK	20120704
eSafe	Win32.Android.Geinim	20120704
F-Prot	-	20120704
F-Secure	Trojan:Android/Geinimi.B	20120704

圖 4.1 virustotal 檢測範例

我們總共挑選 368 個正常程式當作正常程式資料庫，400 個惡意程式當作惡意程式資料庫。並且我們挑選其中 146 個來測試跟正常程式之間的相似度關係，其中這 146 個彼此相似度皆不為 1，即它們為皆不相同的惡意程式。我們測試這些惡意程式跟正常程式之間的相似度分布，並且比較使用擷取資訊方法的前後差別。

擷取前後 相似度範圍	擷取前相似度分布	擷取後相似度分布
相似度≤0	12317	14692
0<相似度≤0.1	14152	15514
0.1<相似度≤0.2	7208	8570
0.2<相似度≤0.3	5744	7430
0.3<相似度≤0.4	5424	5152
0.4<相似度≤0.5	4670	1915
0.5<相似度≤0.6	3177	378
0.6<相似度≤0.7	900	61
0.7<相似度≤0.8	119	10
0.8<相似度≤0.9	13	6
0.9<相似度≤1.0	4	0

表 4.1 擷取資訊前後相似度分布

在正常的程式跟惡意的程式之間，理論上相似度要越低越好，亦即在相似度低的地方要分布多一點，在相似度高的地方要分布少一點，我們可以從表 4.1 中發現，在使用我們的方法擷取有用的資訊之後，已經沒有分布在相似度 0.9 以上，並且在相似度 0.3-1.0 之間皆變得較少，而 0.3 以下的變得較多，因此使用擷取資訊的方法有讓相似度的計算變得更為準確。我們也可以進一步得知當臨界值設為不同數值的時候，兩者之間的誤判率差異，如下表 4.2。

擷取前後 臨界值	擷取前誤判個數/全部個數(誤判率)	擷取後誤判個數/全部個數(誤判率)
0	366/368(99.46%)	363/368(98.64%)
0.1	365/368(99.18%)	353/368(95.92%)
0.2	362/368(98.37%)	341/368(92.66%)
0.3	355/368(96.47%)	297/368(80.71%)
0.4	332/368(90.22%)	228/368(61.96%)
0.5	285/368(77.45%)	136/368(36.96%)
0.6	190/368(51.63%)	39/368(10.60%)
0.7	58/368(15.76%)	5/368(1.36%)
0.8	5/368(1.36%)	2/368(0.54%)
0.9	2/368(0.54%)	0/368(0.00%)

表 4.2 臨界值設定跟誤判率差異

其中誤判個數為在 368 個正常程式中，被誤判成惡意程式的情況，亦即只要有一個惡意程式跟其相似度大於或等於臨界值，則此正常程式為誤判，誤判率即為在 368 個正常程式中被誤判成惡意程式的比率。在擷取完有效的資訊以後，由於基底的個數很多，但是其實有很多基底沒有很重要(不太影響相似度)，所以我們在第三章的 3.2(2)有提出刪減基底的方法，我們要將在惡意程式跟正常程式中基底出現次數差異較大的基底保留下來，而將基底出現次數差異較小的基底刪去。我們先在表 4.3 展示我們所統計的基底出現次數的差異大小。

	最小值	最大值
差異個數	0	40229076

表 4.3 基底出現次數的差異統計

在表 4.3 中，我們可以發現基底出現次數的差異變化很大，從最小的 0 到最大的 40229076，所以我們的目標是刪除掉那些差異較小的基底，可以使得利用剩下來的基底去建立特徵向量不會影響表 4.2 所列的誤判個數(誤判率)。我們將在下面表 4.4 中展示刪減基底後的誤判個數跟基底個數的關係。(在此先假定我們的臨界值設定在 0.7，亦即誤判個數為 5 個。) 我們每一次的試驗的基底個數，若是遇到非整數情形，則取整數部分。我們一直採用二分法並且觀察其是否有達到誤判率要求，並且每次試驗的基底個數會是最近一次達到誤判率要求的個數和最近一次沒有達到誤判率要求的個數的平均值；舉個例子，663=[442(X)+884(O)]/2。

試驗次數	基底個數	誤判個數	達到誤判率要求 (O/X)
第 1 次	1768	5	O
第 2 次	884	5	O
第 3 次	442	7	X
第 4 次	663	5	O
第 5 次	552	5	O
第 6 次	497	6	X
第 7 次	524	5	O
第 8 次	511	5	O
第 9 次	504	6	X
第 10 次	507	5	O
第 11 次	505	6	X
第 12 次	506	6	X

表 4.4 刪減基底測試誤判個數跟基底個數

在這 12 次的試驗中，其中第一次試驗為全部基底的個數，因此誤判個數會跟表 4.2 所列的相同，從第 2 次試驗開始，逐次將差異較小的基底刪去，若是沒有達

到誤判率要求，則下一次試驗將增加一些(其中差異較大的)被刪去的基底回來，若是有達到要求，則跟最近一次沒有達到要求的基底個數取平均值，進行下一次試驗，一直重複做直到找到基底個數最少且滿足誤判率要求才停止。在此試驗臨界值設定為 0.7，最終的基底個數為 507，亦即使用這些差異較大的前 507 個基底當作特徵向量的基底，所計算的誤判率結果會跟原始的 1768 個基底相同，且 507 為最少且符合誤判率要求的基底個數。當然基底個數 507 這個值並不是一個定值，會隨著臨界值制定的不同，或是誤判率所要求的不同而改變。我們另外去做了參考論文所說的使用基底出現次數最多的當作特徵向量的基底，並且一樣找到符合誤判率要求的最少基底個數為 503 個，這也許能夠用說明，那些在惡意程式跟正常程式中基底出現次數差異較大的基底，會大約等於那些出現次數比較多的基底，亦即出現次數越多的基底越有可能差異越大。

我們做了基底給予不同比重的測試，目標是希望能夠找一組分配不同基底的不同比重，使得在相同條件下的誤判率能夠比原本更低。我們的想法是既然每個基底在正常程式中及惡意程式中的差異有大有小，那如果我們將基底給予不同的比重，可能可以讓相似度的計算更能區分正常程式跟惡意程式，在上面的測試中，我們發現那些差異較大的基底也是恰為那些出現次數較多的基底，所以我們這邊會使用出現次數來做為比重給予大小的依據。我們覺得有兩個因素會影響基底的比重，第一個因素是基底在多少個程式中有出現過，即基底的普遍性高不高，我們覺得出現在越多個程式中的這種基底會比出現在少數程式中的基底來的重要；第二個因素是基底在所有程式中累積出現次數，代表說這種基底通常都會出現比較多次，因此在距離的計算中，應該要佔較小的比重。舉例說明，若是有兩種基底累積出現次數不同，且它們在距離另一個向量的這兩種基底下都同樣是 1(出現次數差 1)，則我們必須將累積出現次數較多的那個基底給予較小的比重(例如 0.5)，將累積次數較少的基底給予較大的比重(例如 0.6)。因此我們定義我們基底的比重是跟其出現在多少個

程式中成正比，而跟其累積出現次數成反比。因此我們將不同的基底給予不同的比重，原本的相似度公式將更改為如下說明：

$s(p,q)=1-\frac{\sum_{i=1}^{n} d(p_i,q_i)*weight_i}{\sum_{i=1}^{n} q_i*weight_i}$，其中 p 為惡意程式中的某一程式的特徵向量，q 為受檢測的程式的特徵向量。p_i 為 p 的第 i 個基底出現的次數，即 p 向量的第 i 個分量，同理 q_i 為 q 向量的第 i 個分量。$d(p_i,q_i)$ 為第 i 個分量的距離，即分量的差取絕對值（$|p_i-q_i|$）。其中 $weight_i=\frac{\text{第i個基底出現在多少個程式中}}{\text{第i個基底在程式中累積出現次數}}$ 為每個基底所分配的比重。我們在固定基底個數下，去測試我們所提出的基底的比重，對於誤判率跟臨界值的關係的影響，將結果列在下表 4.5 中：

分配比重前後 / 臨界值	分配比重前誤判個數/全部個數(誤判率)	分配比重後誤判個數/全部個數(誤判率)
0	363/368(98.64%)	359/368(97.55%)
0.1	353/368(95.92%)	298/368(80.98%)
0.2	341/368(92.66%)	249/368(67.66%)
0.3	297/368(80.71%)	139/368(37.77%)
0.4	228/368(61.96%)	54/368(14.67%)
0.5	136/368(36.96%)	236/368(6.25%)
0.6	39/368(10.60%)	7/368(1.90%)
0.7	5/368(1.36%)	2/368(0.54%)
0.8	2/368(0.54%)	2/368(0.54%)
0.9	0/368(0.00%)	0/368(0.00%)

表 4.5 分配完基底比重前後臨界值跟誤判率的關係

除了誤判率以外，我們還想知道變種偵測的能力。若是我們將分配基底比重前後的誤判率固定在定值，在此例中分配比重前臨界值 0.8，分配比重後臨界值 0.7，

兩者誤判率相同(0.54%)，我們想要觀察在惡意程式資料庫中彼此之間的相似度關係，我們發現在分配基底比重之後，在新的相似度算法下相同的惡意程式會有更多的程式跟其相似度高於 0.7，比原本尚未分配基底比重的相似度算法，能找到更多相似度高於臨界值的程式。我們在 146 個惡意程式所組成的資料庫中，其中有 17 個程式在使用分配基底比重後，可以找到更多相似度高於臨界值的程式，129 個程式找到個數相同，0 個程式找到個數更少。因此可以確定我們的分配基底比重的方法，可以在相同誤判率的條件下，偵測到更多的變種；或者可以確定在偵測到相同的變種的條件下，我們的方法誤判率會較低(較不易將正常程式判定為惡意程式)。我們在第三章有提到使用曼哈頓距離預先濾波器來濾除在資料庫中較不相似的程式，並事先建立好每個區域中的惡意程式成員，事先建立二進位樹在偵測時做搜尋用。因此在偵測時我們需要做計算曼哈頓的距離的值並且判斷在哪個區域中然後在去那個區域中事先建立好的二進位樹做搜尋。我們測試我們的時間包含計算曼哈頓距離跟判斷區域以及搜尋到最相似的程式所花費的時間，並跟原論文的方法中在整個二近位樹中搜尋最相似的程式去做比較，我們分別測試 500、1000、2000、5000 個受偵測程式(虛擬創造的特徵向量)，去比較我們方法跟原論文方法的時間，在建立特徵向量的時間先假定相同，因為有很多種方法可以從 apk 建立特徵向量，我們假設是使用相同的方法去建立特徵向量，去比較不同處所花費的時間。

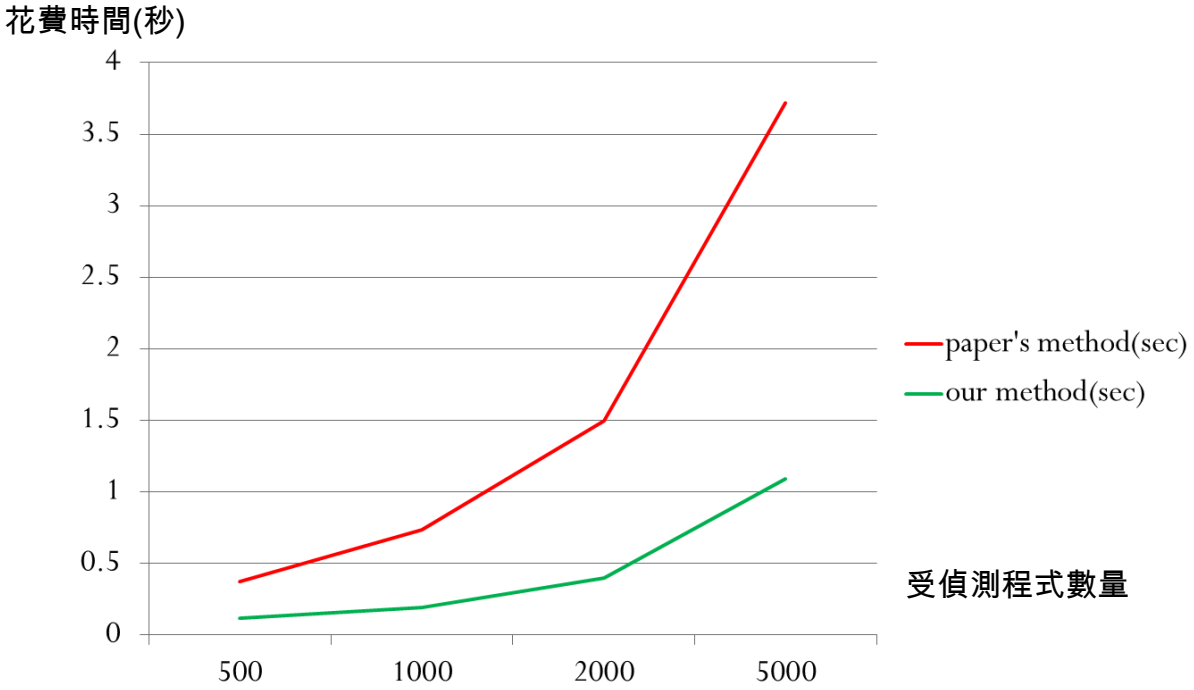


圖4.2 偵測時間比較(不含建立向量時間)

在圖4.2中，我們可以發現我們的方法較為省時，我們的測試環境是在只有使用146個惡意程式的資料庫，因為惡意程式更新速度很快，所以資料庫中的程式會越來越多，需要搜尋的程式也會跟著增加，若是不能根據其向量的特性去分配其資料庫中部份的程式去搜尋，每次都搜尋全部的程式，則搜尋的時間問題會隨著資料庫中程式增加而越來越嚴重。

第五章

結論

我們在利用 q-gram 擷取資訊時，提出有效率的擷取資訊方法，使得誤判率下降。我們也提出了刪減基底的方法，可以在誤判率不變的情況下使用最少的基底，去建立特徵向量。我們提出每種基底應分配不同的比重，如此計算出的相似度將可以更能區分惡意程式跟正常程式間的差別，並且在維持相同的誤判率下能夠偵測更多的變種，或是能夠在相同的偵測能力下擁有較小的誤判率。我們也提出使用曼哈頓距離預先濾波器的方法，能夠事先建立不同曼哈頓距離區域的二進位樹，由惡意程式資料庫中的程式所組成，能夠讓偵測時只需要去特定的區域中搜尋，不須搜尋整個資料庫中的程式。

我們也提出了使用數學模型來建立程式特性的想法，我們覺得若是未來能夠在向量中找到某些變數或是某種統計值會影響這些惡意程式跟正常程式的特性，能夠提升預先濾波器的效能，使得剩下來要去做搜尋的程式又更少了，若能做到大部分的正常程式都能夠經過預先濾波器初步的檢測就已知其為正常程式，那理想的數學模型也就很接近了。在未來研究，可以朝如何縮短建立特徵向量的時間，或是找到數學模型的其他參數描述特徵向量的特性，或是如何建立程式中某些特定程式的特性(例如一段惡意程式嵌在正常程式之中)，以及如何節省資料庫中所需儲存的惡意程式(提升偵測變種能力)。

參考文獻

[1] S.Cesare and Y.Xiang “ Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs”, Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference.

[2] Androguard ,http://code.google.com/p/androguard/

[3] Peter N.Yianilos” Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces”, SODA '93 Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, Pages 311 – 321.

[4] A.Shabtai et.al, “Google Android: A comprehensive Security Assessment”, 2010 IEEE.

[5] G.Delac et.al, “Emerging Security Threats for Mobile Platforms”, MIPRO, 2011 Proceedings of the 34th International Convention.

[6] S. Cesare and Y. Xiang, "A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost," in IEEE 24th International Conference on Advanced Information Networking and Application (AINA 2010), 2010.

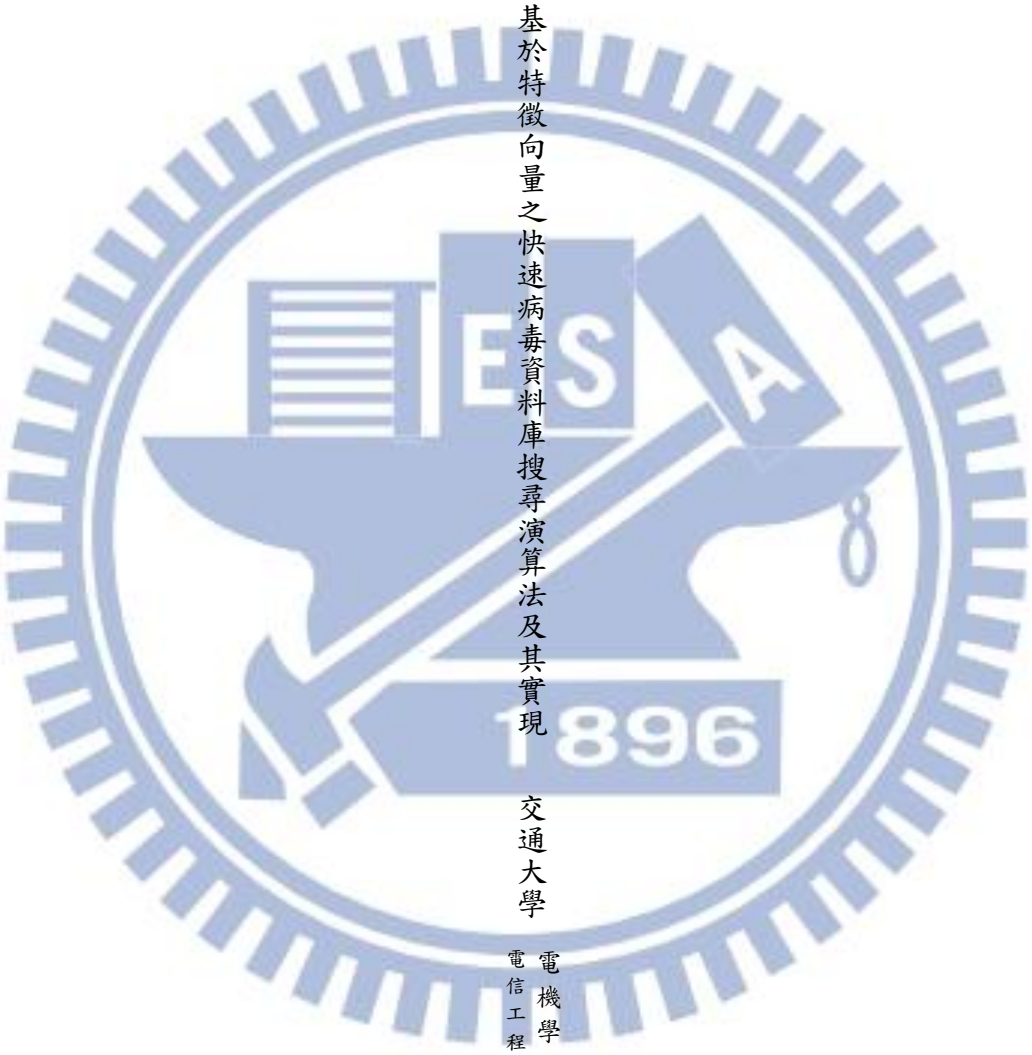
[7] S. Cesare and Y. Xiang, "Classification of Malware Using Structured Control Flow," in 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), 2010.

[8] K. Griffin, et al., "Automatic Generation of String Signatures for Malware Detection," in Recent Advances in Intrusion Detection: 12th International Symposium, RAID 2009, Saint-Malo, France, 2009.

100

碩士論文

基於特徵向量之快速病毒資料庫搜尋演算法及其實現



交通大學 電機學院
電信工程研究所碩士班

邱嗣儒


E S A
1896