國立交通大學

網路工程研究所

碩 士 論 文

MalCatcher:以存取以及網路洩漏隱私資料行為為基礎的 Android 惡意程式行為偵測

MalCatcher: Private and Network Data Leakage Behavior-Based Malware Detection on Android

研 究 生：謝維揚

指導教授：曾文貴 教授

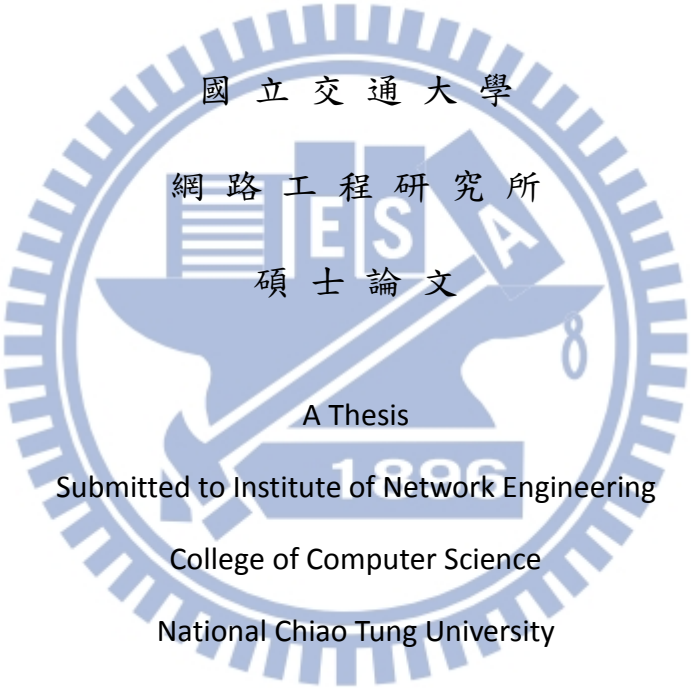
中 華 民 國 102 年 6 月

MalCatcher:以存取以及網路洩漏隱私資料行為為基礎的Android惡意程式行為偵測

MalCatcher: Private and Network Data Leakage Behavior-Based Malware Detection on Android

研 究 生：謝維揚　　Student：Wei-Yung Hsieh

指導教授：曾文貴　　Advisor：Wen-Guey Tzeng



國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science

June 2012

Hsinchu, Taiwan, Republic of China

中華民國 102 年 6 月

MalCatcher: 以存取以及網路洩漏隱私資料行為為基礎的 Android 上惡意程式行為偵測

學生：謝維揚 **指導教授：曾文貴**

國立交通大學網路工程研究所碩士班

摘　　要

由於智慧型手機的多功能性以及方便性，使得越來越多人都擁有智慧型手機，人們漸漸地將他們自身的隱私資料存入手機中，因此智慧型手機的系統安全防護就顯得非常重要，其中由於 Android 系統是屬於開放式原始碼，任何人皆可研究其系統詳細的架構以及運作流程，使得在其系統上開發惡意程式的門檻降低。在最近幾年，Android 上的惡意程式數量長大速度非常地快速，在現今防毒軟體所使用的 signature-based static analysis 方法已經無法跟上惡意程式的演化速度，因此我們希望能透過 dynamic analysis 方法來偵測惡意程式，並且盡可能地降低誤判的機率，提升偵測惡意程式的效果。因此，我們提出了一個新的有效動態分析(dynamic analysis)Android 惡意程式行為的方法，希望能有效地偵測出惡意程式行為，我們也實作出了一個名為 MalCatcher 的系統套用了上述的方法，實作的方法主要是修改了 Android 系統原始碼並重新編譯，然後運行在模擬器上建立一獨立且受控制的環境供 APP 導入執行，並且我們也將一網路封包監控軟體 snort 重新交叉編譯為 Android 系統可執行之版本運行在模擬器上來監控網路封包是否有洩漏使用者隱私資料。我們也取得了大量的實際惡意程式以及正常 APP 進行實驗來驗測試並驗證我們的方法的效果，實驗結果也顯示了我們的方法能夠非常有效地偵測出惡意程式。

MalCatcher: Private and Network Data Leakage Behavior-Based Malware Detection on Android

Student: Wei-Yung Hsieh Advisor: Wen-Guey Tzeng

Institute of Network Engineering
College of Computer Science
National Chiao Tung University

ABSTRACT

More and more people use smartphones in the world. People put more and more their own personal private information into smartphones, so it is important to secure the mobile system, especially Android. Due to the fact that Android is an open-source system, it is easier to develop malwares on Android. In recent years, the number of malwares is dramatically increasing and evolving on Android. We need a effective approach to keep up the speed of malwares' changes. In this paper, we propose a new dynamic analysis scheme for malware detection on Android, we monitor the app's behaviors during its execution time and use these behavior information to judge the app whether a malware or not. We also develop a system called MalCatcher to implement our scheme. We add logging function into the Android system source code and compile the modified source code to a system image to build an isolated and monitored environment for Android apps' execution. Moreover, we have gained a large number of truly malware and normal app to do our experiment and testify our scheme's effectiveness. Our result show that our scheme can detect malware efficiently.

誌　　謝

首先我要感謝我的指導教授曾文貴老師這兩年來的認真教導，讓我對於密碼學有了更深一層的了解，並且也學習到了台上報告的技巧，如何讓台下聽眾能夠清楚地了解自己報告的內容，也很感謝老師在我研究上細心的指導，讓我能夠完成這一篇論文。同樣也非常感謝蔡錫鈞老師以及孫宏民老師在口試時給予建議，讓我的研究能更加完整。感謝毅睿學長以及宣佐學長平日在實驗室指導我完成實驗室的計畫以及研讀國外學者文獻的技巧與重點，讓我在不管程式撰寫能力或是閱讀國外文獻的能力都有很大的進步。最後感謝所有幫助過我，讓我能夠順利完成這次碩士論文研究的人，有了大家的幫忙我才能完成這篇論文。

目　　錄

圖片目錄

表格目錄

一、 前　　言

1.1 背景介紹

近年來，智慧型手機越來越熱門，根據 Gartner[1]的市場調查，如下圖 1，詳細的列出了各大行動平台手機全球市佔率以及數量的數據比較，搭載 Android 系統的智慧型手機在 2012 第四季全球市占達到 69.7%成為市占率第一，Apple 的智慧型手機系統 iOS 市占率為 20.9%居市占率第二，由此可看出 Android 系統已經是智慧型手機市場的龍頭，智慧型手機市場已漸漸飽和，幾乎每個人都持有智慧型手機。手機系統供應商(如 Google、Apple)架設了應用程式商店集中許多應用程式(或稱為 APP)以方便使用者下載，因為有了多樣化的 APP 可供使用者使用，才使得智慧型手機已非常驚人的速度普及，結至 2012 年 9 月為止，在 Google Play[2]上已有 675,000 個 APP，Apple App Store[3]上已有 700,000 個 APP 可供下載。

Worldwide Smartphone Sales to End Users by Operating System in 4Q12 (Thousands of Units)

Operating System	4Q12 Units	4Q12 Market Share (%)	4Q11 Units	4Q11 Market Share (%)
Android	144,720.3	69.7	77,054.2	51.3
iOS	43,457.4	20.9	35,456.0	23.6
Research In Motion	7,333.0	3.5	13,184.5	8.8
Microsoft	6,185.5	3.0	2,759.0	1.8
Bada	2,684.0	1.3	3,111.3	2.1
Symbian	2,569.1	1.2	17,458.4	11.6
Others	713.1	0.3	1,166.5	0.8
Total	**207,662.4**	**100.0**	**150,189.9**	**100.0**

圖 1 智慧型手機全球市占率表(來源: Gartner[1])

因為智慧型手機的強大的功能，使用者在日常生活上越來越依賴智慧型手機，不知不覺中手機裡漸漸地存有許多使用者的隱私資料，如通訊錄、簡訊內容、照片、對話紀錄、帳號密碼、目前位置等等。基於上述的原因，越來越多的惡意程式作者開始將他們的矛頭指向了 Android 系統，根據 Juniper Networks 的 Global Threat Center[4]的報告指

出，在 2010 年至 2011 年這段期間，Android 上的惡意程式數量成長了四倍，在 F-Secure 的 Mobile Threat Report Q4 2012[5]中指出，Android 上惡意程式在各平台的比例由 2010 年的 11.25%成長至 2012 年的 79%，我們可以從圖 2 中看到從 2010 年至 2012 年各個行動平台上的惡意程式數量百分比的變化，成長的速度非常驚人，因此 Android 系統的安全防護是一個非常重要的課題。

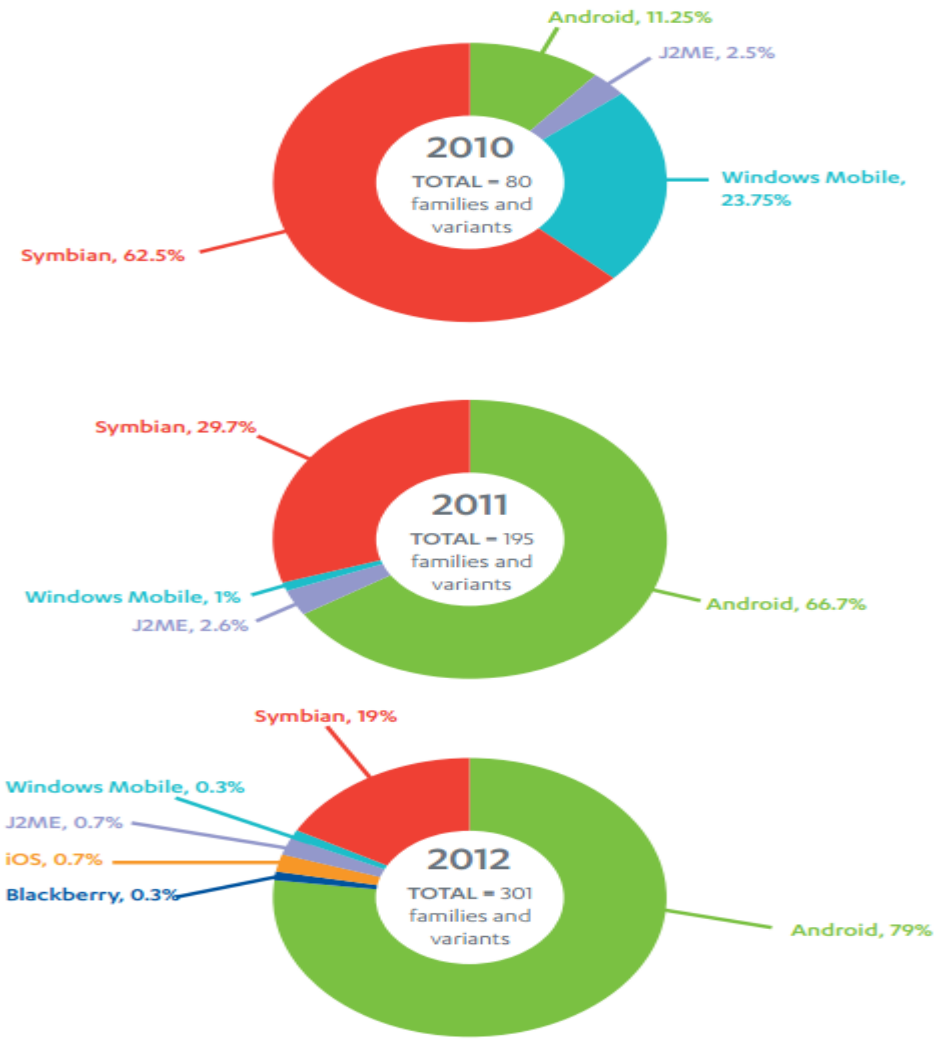


圖 2 各平台惡意程式比例圖(來源: F-Secure[5])

1.2 問題描述

Fake Player、Gemini、DroidKungFu 等等這些都是非常著名的惡意程式，有許多的 APP 經過惡意程式作者的修改、植入惡意程式碼後再重新包裝偽裝成知名熱門的正常 APP 提供給使用者下載，藉由這樣的方式來使得使用者在不知不覺中遭受感染。雖然目前已有許多人開始研究如何偵測出惡意程式，但是惡意程式的演進速度實在是太快，一種惡意程式可能會隨著偵測方法的改變進而改良成為更進一步規避偵測的變形體，這對於 Android 系統安全研究者來說實在是一大挫折。因此如何即時的跟上甚至超越惡意程式的演化腳步是很重要的，若能即時偵測出先前尚未被發現的惡意程式的話，我們就能即時地確保系統的安全。

1.3 貢　　獻

在本篇論文中，提出了一個新的動態分析方法來分析 Android 惡意程式行為，藉由隱私資料的存取和網路封包監控隱私資料洩漏的行為監控來動態分析並判斷 APP 是否為一惡意程式，並且透過大量真正的惡意程式以及正常的 APP 來測試本系統架構的效果，我們實作出了一個系統 **MalCatcher** 實現了上述的方法並且經過實驗測試。

實驗結果顯示，本篇的系統架構能夠有效地偵測判斷 APP 是否為惡意程式，在 412 個真正的惡意程式以及 412 個正常的 APP 樣本測試下，能得出高達九成五的判斷準確度，也就是在這 824 個樣本中有 95%的樣本能被正確的分類。

1.4 組織架構

本篇論文的組織架構如下，第二章節描述了近幾年來對於 Android 系統上提出過的惡意程式行為偵測方法的介紹。第三章節詳細描述了系統架構和解釋惡意程式行為偵測的方法以及原理。第四章節中我們詳細的解釋如何實作出我的系統。第五章節中提出了

詳細的實驗方法以及結果，並且對於實驗結果作了詳細的分析以及討論。第六章節我們探討目前系統尚未完善並且待改善的部分。第七章節中為本篇論文做個總結。

二、相關研究

在目前研究中，分析並偵測惡意程式行為的方法主要有兩大類：靜態分析(static analysis)和動態分析(dynamic analysis)，目前各大知名防毒軟體公司幾乎主要都是使用靜態分析的方法來偵測惡意程式。

靜態分析的方法主要是透過解析惡意程式的程式碼，找出惡意行為的程式碼片段之後做成特徵碼，並將這些特徵碼收集起來建立成資料庫，最後防毒軟體利用這些特徵碼資料庫來偵測裝置上是否有惡意程式。雖然靜態分析方法能很精準地找到惡意程式，但是隨著惡意程式開發者的改進，阻礙分析者對於惡意程式的程式碼分析的方法越來越多，使得靜態分析越來越難找到惡意行為的程式碼片段，再加上由於對程式碼作分析需要時間，因此在特徵碼分析出來之前，防毒軟體常常對於尚未發現或是剛發現不久的惡意程式是無法偵測出來的，這一空窗期很有可能會對系統造成很大的威脅。

另一方面，動態分析的方法主要是透過在一獨立以及受控制的作業環境執行惡意程式並且記錄程式執行期間的相關資訊、行為，之後利用這些資訊來去判斷程式是否為惡意程式。由於動態分析是藉由記錄的資訊、行為來對程式作預測是否為惡意程式，因此在判斷是有一定的機率會造成誤判，但是由於是利用程式執行期間的記錄資訊來做判斷、預測，因此動態分析能及時得抓出未知的惡意程式以保護系統的安全。

Asaf Shabtai 等人[6]主要透過了多種不同的行為監控來偵測惡意程式，如 CPU 使用時間、送出的封包數量、使用者互動事件(如觸碰或滑動螢幕、按下電源鍵等)、執行中的程序數量、電量消耗等。Min Zhao 等人[7]主要以 APP 執行期間所產生的 intents(也可以說是系統事件訊號)以及一些系統資源的使用狀況(如存取目前位置、使用媒體撥放器、存取或寄送簡訊等)為依據來偵測惡意程式。Dong-Jie Wu 等人[8]主要透過解析 APP 的程式碼來收集 APP 使用的 permission、activity、receiver、API call 等資訊，利用這些資訊為依據並且搭配 K-means 分群演算法來偵測惡意程式。Takamasa Isohara 等人[9]收集了 APP 執行期間的特定 system call 執行狀況，如 read、write、recv、open、bind 等而非所有的 system call 種類為依據來偵測惡意程式。Gianluca Dini 等人[10]同時收集了 APP 執行期間的 kernel-level 和 application-level 的資訊，kernel-level 的有如 CPU 使

用率、system call 執行狀況、執行中的程序、RAM 剩餘空間等資訊，user-level 的有如 APP 閒置/啟動狀態、使用者輸入事件、接收或寄送簡訊事件、藍芽/Wifi 狀態等資訊為依據來偵測惡意程式行為。Iker Burguera 等人[11]收集了 APP 執行期間所有的 system call 執行狀況並以此為依據來偵測惡意程式行為。

由上述的眾多方法我們可以瞭解到，不同動態分析方法之間最大的不同點在於監控的資訊、行為的種類，根據挑選監控的事件資訊的不同，偵測惡意程式的效果也會有很大的不同，雖然上述的方法所展現出來的實驗結果都還不錯，但是其中並沒有任何一篇實際運用了大量的真正惡意程式為測試樣本來測試其系統是否真的能有效的偵測出惡意程式，因此在本篇論文中，運用了由 Yajin Zhou 和 Xuxian Jiang[12]所提供的 1260 個真正的惡意程式以及由陶嘉仁[13]提供的 824 個正常的 APP 來測試本系統以得到最實際的效率。

三、 系統設計

3.1 系統原理

在 **MalCatcher** 中我們採用了監控了隱私資料存取的行為，稱為 **private data leakage operation**，以及網路封包洩漏隱私資料的行為，稱為 **network leakage operation** 為主要依據，希望透過這些行為能讓 **Malcatcher** 有效地偵測出惡意程式行為，系統架構圖如圖 3，我們修改了 Android 4.0.3 CyanogenMod 系統原始碼，加入了必要的監控記錄函式，並且運行在 Android 模擬器上，這些監控記錄的函式主要是在記錄 **private data leakage operation**，並且我們在系統中安裝了 snort，網路封包監控軟體，用來監控網路封包，主要是監控記錄 **network leakage operation**，同時我們在 Linux 環境下利用 shell script 實現自動化導入 APP 進入模擬器執行，執行完成會從模擬器中取出 **private data leakage operation** 以及 **network leakage operation** 的記錄檔，得到這些記錄檔後透過另一個由 shell script 寫成的程式將記錄檔轉換為 Weka 此一統計分析軟體可以接受的格式檔案。

在接下來的部分為分別對 **private data leakage operation** 以及 **network leakage operation** 兩部分行為作定義以及說明選擇監控這些行為的原由為何。

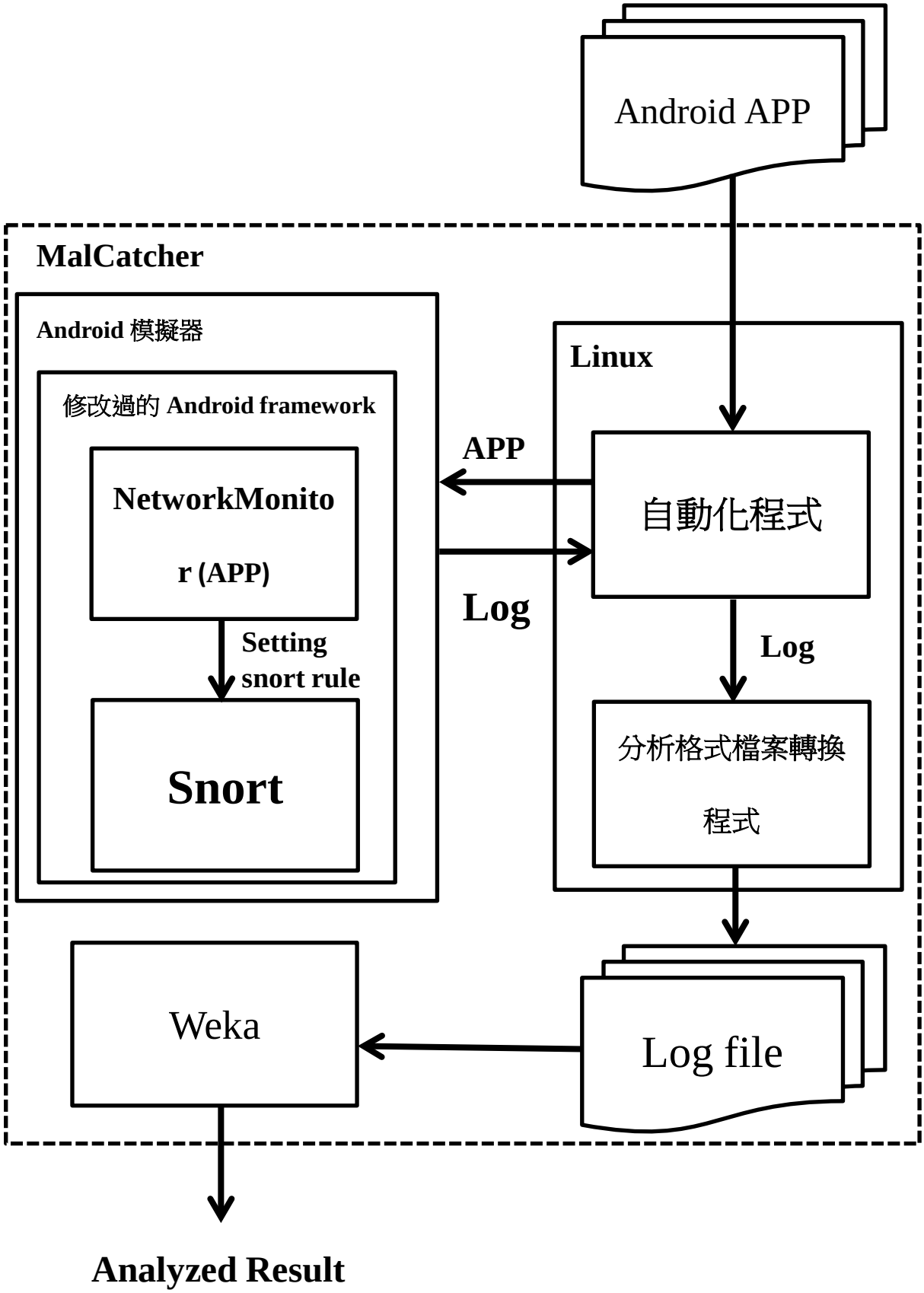


圖 3 MalCatcher 系統架構圖

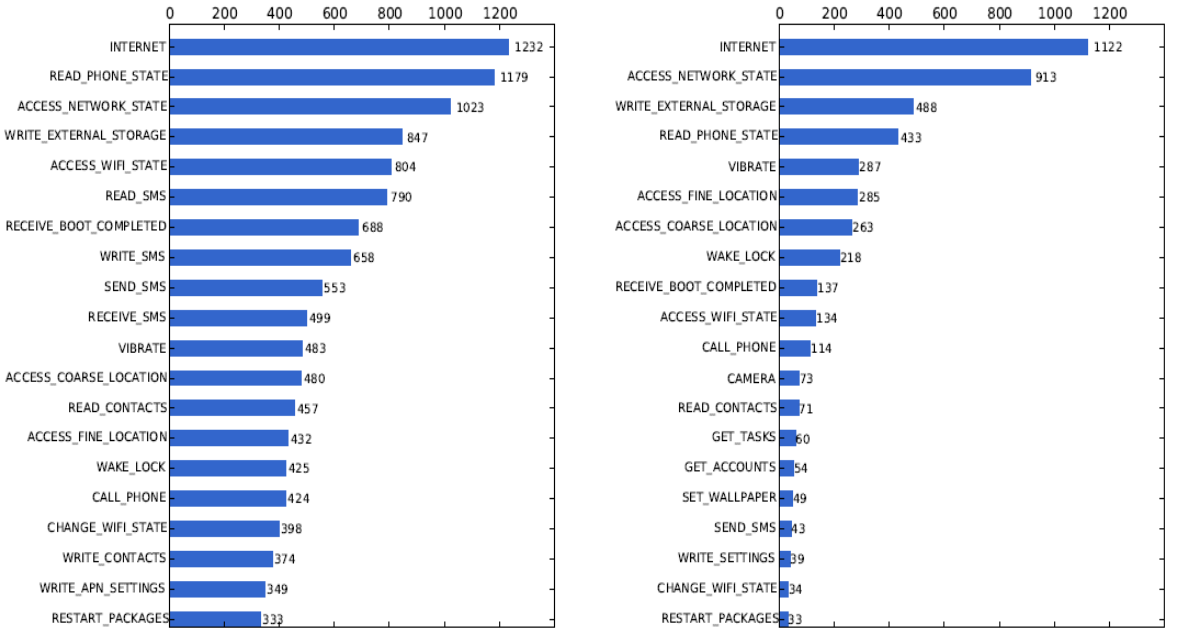
3.1.1 Private Data Leakage Operation

根據 Yajin Zhou, 以及 Xuxian Jiang[12]研究指出，惡意程式的目的大致上分為兩大類，分別是 financial charge、personal information stealing。

(1)**Financial Charge** 惡意程式攻擊的目的其中一個就是從使用者手機獲取利益、金錢，其中最常見的辦法就是透過使用者手機發送小額付費簡訊至特定的號碼。能使得 Android 手機達成這樣的攻擊目的的方法主要就是透過 Android framework 提供的 sendTextMessage API，它能允許任一的 APP 只要宣告使用了特定的權限(也就是 android. permission. SEND_SMS)，之後再透過呼叫 sendTextMessage 就可使得 APP 在使用者不知情的情況下寄送簡訊。

(2)**Personal Information Stealing** 由於使用者越來越依賴手機，漸漸地將個人許多的隱私資料存放在手機上，因此許多惡意程式會竊取手機上的隱私資料，如 IMEI、IMSI、SIM 卡序號、手機號碼、通訊錄名單、簡訊內容、目前位置等。

Android 系統在版本不斷的更新之下，先前版本的漏洞已經逐漸被修復，對於 Android 系統的一鍵 ROOT 程式的實現也越來越困難，因此惡意程式想透過一鍵 ROOT 程式來背景地將受感染的使用者手機完成 ROOT 來取得 super user 權限已經越來越困難了，如此一來惡意程式所能執行的動作大大地受限於 Android 系統的權限管理機制，因此惡意程式宣告使用哪些權限的趨勢對於我們未來的惡意程式偵測有很大的幫助。



(a) Top 20 Permissions Requested By 1260 Malware Samples

(b) Top 20 Permissions Requested by 1260 Top Free (Benign) Apps on the Offical Android Market

圖 4 惡意程式權限宣告趨勢對照(來源: [12])

圖 4 是惡意程式與正常 APP 各別有 1260 個樣本數統計出來前 20 名最常被宣告使用權限的趨勢對照圖，我們可以看出來相較於正常 APP 的權限宣告趨勢，有幾個存取隱私資料的權限是較常被惡意程式宣告使用，如 READ_PHONE_STATE、SEND_SMS、WRITE_SMS、READ_CONTACTS 等在相同樣本數的情況下，宣告使用的次數卻相差相當大，從這點我們可以推斷單一惡意程式平均宣告使用的權限數量可能高出單一正常 APP 很多，由此可見，大部分的惡意程式都會存取使用者手機中的隱私資料，並且很有可能同時存取多種不同類型的隱私資料。

宣告使用某一權限並不能代表 APP 真的去使用到相對應的功能，有可能是因為 APP 開發者對於某一特定功能使用不夠了解或是 Android 的文件不夠詳細導致開發者宣告了多餘的權限。在 Adrienne Porter Felt 等人[14]的研究中就有指出，通常一個 APP 宣告使用了某一權限而在程式碼中卻找不到相對應的功能使用，這種情況通常可能是因為開發者不夠了解或是文件說明不夠清楚所導致。

因此我們認為單純透過 APP 宣告的權限並不能真正地確實描述出其 APP 的行為，所以我們監控實際真正會存取隱私資料的 API，如此一來任何 APP 當需要存取

隱私資料的時候我們就能確實記錄到 APP 真正的行為，而我們將監控這些存取隱私資料的 API 的記錄，稱為 **private data leakage operation**。

3.1.2 Network Leakage Operation

考慮到一般正常 APP 也有可能存取使用者的隱私資料，而且也可能同時存取多種不同的隱私資料類型，但是可能只限於本地手機端作運用而沒有經由網路傳送出去或是可能只傳出少許的隱私資料，若只單純以 **private data leakage operation** 來偵測惡意程式的話，我們推測會有很大的機率對於上述這類型的正常 APP 判斷成為惡意程式。

然而惡意程式存取了許多的隱私資料，那攻擊者想要取得這些資料的唯一途徑就是經由網路傳出，因此為了避免可能會存取許多隱私資料但是不會經由網路傳出或是少許資料傳出的正常 APP 被誤判為惡意程式，MalCatcher 同時監控了網路封包內容是否含有使用者隱私資料，我們將 APP 經由網路傳出使用者隱私資料的行為稱為 **network leakage operation**。

透過 **private data leakage operation** 以及 **network leakage operation**，我們希望 MalCatcher 偵測惡意程式能有不錯的判斷準確度，詳細監控了那些隱私資料的存取以及網路傳出如表 1，其中底線代表是屬於 **network leakage operation**，其餘的則是代表 **private data leakage operation**。

Private data leakage operations & Network leakage operations	**說明**
ACCESS_LOCATION	讀取手機目前的地理位置

CONTACT_NET	將通訊錄由網路送出
CONTACT_READ	讀取手機通訊錄
IMEI_NET	將 IMEI 碼由網路送出
IMEI_READ	讀取手機 IMEI 碼
IMSI_NET	將 IMSI 碼由網路送出
IMSI_READ	讀取手機 IMSI 碼
ISO_NET	將 ISO country code 由網路送出
ISO_READ	讀取手機 ISO country code
PHONE_NUMBER_NET	將手機本身的電話號碼由網路送出
PHONE_NUMBER_READ	讀取手機本身的電話號碼
SIM_NET	將 SIM 卡的序列號由網路送出
SIM_READ	讀取手機的 SIM 卡的序列號
SMS_NET	將手機中的簡訊內容由網路送出
SMS_READ	讀取手機中的簡訊內容

SMS_SEND	使用發送簡訊功能

表 1 private data leakage 以及 network leakage operation 列表

3.2 Android 系統架構

在此一章節主要解釋 Android 基本系統架構對於 Android 系統有基本的了解以及我們如何加入 logging function。

Android 系統是由許多不同部分的軟體集合而成並且運行在手機上的系統，如圖 5 所示，Android 最上層是應用程式層也就是我們平常所稱的 application 層，對於每個 APP 運行時都會藉由 Dalvik virtual machine 產生一獨立虛擬環境運行，因此每個 APP 運行時是獨立且互相無法干擾的狀態，若任兩個 APP 想要進行溝通的話必須透過 Android framework 中提供的 API 以及資料結構才能達成，最下層是 Linux 核心，是 Android 系統最核心的部分，任何跟手機硬體作互動的動作都是經由 Linux 核心來作溝通，所以當 APP 要使用到手機任何功能時皆必須呼叫 Android framwork 中的 API，API 再跟 Linux 核心作溝通最後由 Linux 核心跟手機硬體作溝通來完成 APP 所要求的工作最後將結果透過 API 回傳給 APP。

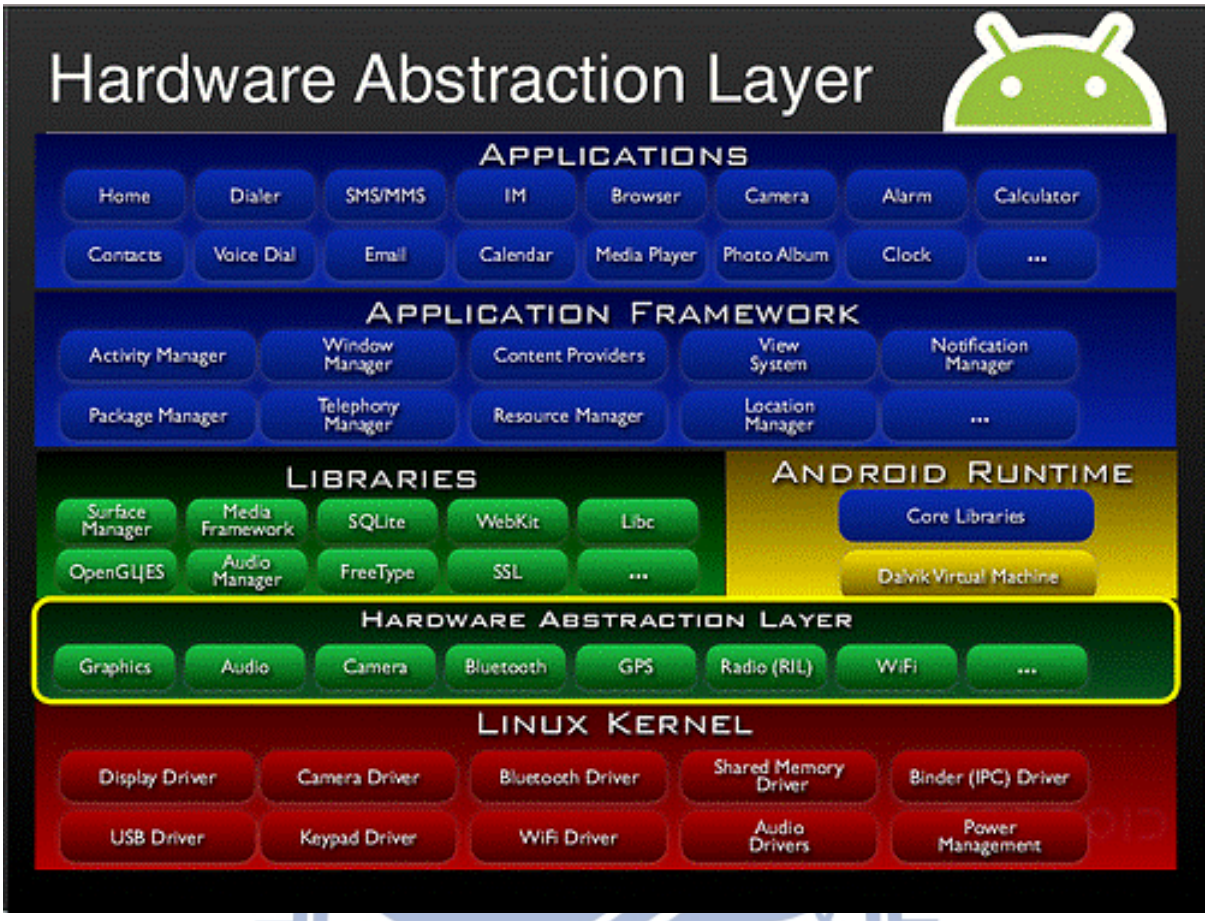


圖 5 Android 系統架構圖

在 Android 系統中有許多的隱私資料是利用一種叫”content provider”的資料結構儲存著，如通訊錄、簡訊內容等。Content provider 是 Android framewrok 中一種特殊的資料結構，主要功用是讓 APP 能有效地管理並儲存 APP 內部的資料，並且 Android framework 也提供了一些 API 使得 APP 能透過這些 API 存取或修改其他 APP 的 content provider 中的資料。

因此 content provider 存取的監控就顯得非常重要，由於我們需要監控 APP 是否有存取隱私資料，在 content provider 方面 MalCatcher 監控了 ContentResolver. java 中的 query()此一 API，在 query()中主要透過一 URI string 來開啟相對應的 content provider(如:” content://sms/inbox”)，因此當 APP 使用了 query()指令並且其中的 URI string 是跟使用者隱私資料相關的(如：通訊錄、簡訊等)，則 MalCatcher 會將此行為記錄下來。

另外有一些隱私資料則不是以 content provider 來儲存，針對這些隱私資料

Android framework 提供了 API 可以存取這些隱私資料，如目前位置、IMEI、IMSI 等，必須透過 LocationManager、TelephonyManager 中的 API 來存取相關資訊，因此針對這些 API 我們也在其中插入了記錄函式來監控這些行為。

3.3 Snort

Snort[15]是一套網路封包入侵檢測的開放原始碼軟體，snort 可以針對使用者設定的關鍵字(又或者稱為特徵碼)與通訊協定規則來偵測封包內容是否有相符條件的封包並記錄下來提醒使用者，1998 年，Martin Roesch 撰寫了此一軟體，原本其定位為簡單輕巧的入侵檢測軟體，近年來在 rule language 以及偵測功能的進步下，snort 已成為最精確與最富彈性的入侵偵測軟體之一[16]。

Snort 主要的運行平台是 Linux 上，但是由於 Android 手機跟 PC 不同的是它使用了 ARM 架構，因此我們利用 Snort 開放原始碼的特性，使用其原始碼並且透過 cross-compiling(或者稱為交叉編譯)技術將 snort 編譯為 ARM 架構上可執行的版本，然後安裝在手機上來針對 Android 網路封包進行監控，記錄是否有隱私資料的洩漏。

Snort 是一 signature-based 的入侵偵測軟體，因此我們另外撰寫了另一 Android APP 名為 NetworkMonitor，主要功能是用來存取我們所需監控的隱私資料，並且將這些隱私資料轉換為 snort rule 的格式，同時針對每一隱私資料的 rule 提供了原文編碼格式以及 base64 編碼格式以確保 snort 在監控封包能確實將這些可能以不同編碼方式傳出的隱私資料來被偵測出來。

3.4 自動化執行

由於取得的實驗樣本數有兩千筆，光是要讓這些實驗樣本全部都執行完可能需要 6~10 天的時間，如何有效率地讓機器自動化地執行就顯得非常地重要，因此在各個部分能讓機器自動化執行的部分我們盡量讓它能自動化執行，所有的自動化執行程式都是由 shell script 撰寫而成。

首先需要自動化執行的部分就是 APP 導入模擬器執行並且自動輸入使用者與手機互動的事件，並且當 APP 執行結束之後能在自動地將記錄下來的資訊由模擬器擷取出來並且導入下一個 APP 繼續執行，執行所有 APP 並且記錄其隱私資料存取以及網路洩漏的資訊非常的耗時，整體執行時間可能需要 5~6 天，人工導入執行非常耗時也不切實際，因此這一部分是必定需要完成自動化執行。再來第二部分就是針對這些所有記錄下來的資訊作整理並且轉換為一向量，每一個向量代表著一 APP 執行期間所有監控行為的執行狀況，向量中每一個 entry 皆代表著一種受監控的行為，其中 entry 中的數字代表著此一行為在執行期間被執行的次數，範例如下

3, 0, 0, 0, 13, 0, 1, 9, 0, 0, 1, 0, 2, 0, 0, 0

此一向量代表著一 APP 執行期間受監控的行為被執行的狀況，假設第一個 entry 代表著通訊錄被存取的行為，第二個 entry 代表著目前位置被存取的行為，則說明著此一 APP 在執行期間存取通訊錄總共存取了三次，而存取目前位置的次數為零次。

將 APP 受監控的行為紀錄資訊轉換為向量之後，為了方便之後資料處理，同時必須將這些向量集中在同一個檔案之中，並且因為實驗方法的關係，我們也必須將這些實驗樣本的向量數據分為 training set 以及 testing set 兩個檔案。

3.5 資料前處理

有了這些已經轉換為向量集合而成的檔案之後，接下來我們必須將這些檔案轉換為 Weka[17] 可接受的 ARFF(Attribute-Relation File Format) 檔案格式。Weka(Waikato Environment for Knowledge Analysis)是一由 JAVA 語言所撰寫而成的一機器學習分析免費軟體，由紐西蘭的 University of Waikato 開發而成，在軟體中提供了許多著名的機器學習演算法並且也提供導入使用者自己撰寫的演算法進入軟體中使用[18]。在本篇論文中，我們利用 Weka 此一軟體執行機器學習演算法來分析我們的實驗樣本數據，因此我們撰寫了一 C++語言程式來將我們收集到的向量數據檔案轉換為 ARFF 格式檔案以供之後導入 Weka 進行分析。

四、 系統實作

在本章節我們詳細地解釋以及描述了我們如何修改 Android 系統原始碼來監控隱私資料存取的行為，如何將 snort 原始碼交叉編譯成 Android 手機能夠執行的版本，自動化執行的程式詳細是如何實現以及撰寫，還有 Weka 可接受之 ARFF 檔案格式說明。

4.1 Android 系統原始碼修改

在此一小節中，我們詳細地解釋如何在 Android 系統原始碼中加入我們自行撰寫的記錄函式以及如何將已經過修改的 Android 系統原始碼進行編譯。

首先必須先在 Ubuntu 64bit 系統環境中安裝好所需的套件

```
sudo apt-get install git-core gnupg flex bison gperf libsdl-dev libesd0-dev
libwxgtk2.6-dev build-essential zip curl libncurses5-dev zlib1g-dev valgrind
lib32readline-gplv2-dev gcc-multilib g++-multilib libc6-dev-i386 lib32ncurses5-dev
ia32-libs x11proto-core-dev libx11-dev lib32z-dev pngcrush schedtool
```

之後回到使用者家目錄並且建立一名為”bin”的目錄，接下來執行下列指令準備開始下載 Android 原始碼

```
curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
chmod a+x ~/bin/repo
```

建立一目錄，在此以”Android”為範例並且移至此目錄中

```
mkdir Android
cd Android
```

開始下載 Android 原始碼

```
repo init -u git://github.com/CyanogenMod/android.git -b ics
```

repo sync

下載完成後我們即可對 Android 原始碼進行修改。

在 Android 原始碼中，我們主要針對 frameworks 目錄底下找出各個使用者隱私資料存取行為的原始碼檔案加入記錄函式，其中記錄函式的程式碼如下圖 6 跟圖 7

```
1  public String getAppName(int pid)
2  {
3      String name="";
4      StringBuffer out = new StringBuffer();
5      try
6      {
7          Process exec=Runtime.getRuntime().exec("ps");
8          exec.waitFor();
9          BufferedReader stdout = new BufferedReader(new InputStreamReader(exec.getInputStream()));
10         String line="";
11         int read,flag=0;
12         char buf[] = new char[512];
13         while((line=stdout.readLine())!=null)
14         {
15             if(flag!=0)
16             {
17                 String[] ps=line.split(" +");
18                 if(Integer.parseInt(ps[1])==pid)
19                     name=ps[ps.length-1];
20             }
21             else
22                 flag=1;
23         }
24         System.out.print(out.toString());
25     }
26     catch(Exception e)
27     {
28     }
29     return name;
30 }
```

圖 6 getAppName 函式原始碼

由於 Android 內建的 API 只能讀取到 APP 執行期間的 process id(pid)以及 user id(uid)，為了讓我們能夠記錄詳細且清楚的行為資訊，所以我們希望能取得 APP 名稱。方法主要就是透過 Linux 指令”ps”取得當前 Android 系統中執行中的所有程序列表，其中包含了 pid 以及相對應的程序名稱，因此我們透過這一指令取得程序列表並且利用 API 取得的 pid 找出相對應的 APP 名稱。

```
public void FeatureLog(int uid,int pid,String msg)
{
    if(uid<=10000)
        return;
    String result="";
    String script="";
    Time t = new Time("GMT+8");
    String pakName = getAppName(pid);
    t.setToNow();
    String time =Integer.toString(t.year);
    time=time+"-"+Integer.toString(t.month)+"-"+Integer.toString(t.monthDay)
    +" "+Integer.toString(t.hour)+":"+Integer.toString(t.minute)+"/";
    result = time+pakName+"("+pid+")";
    result = result+": "+msg;
    Calendar now = Calendar.getInstance();
    StringBuffer output = new StringBuffer();
    try
    {
        Process exec = Runtime.getRuntime().exec("su");
        OutputStreamWriter out = new OutputStreamWriter(exec.getOutputStream());
        out.write("echo \""+result+"\" >>/data/logging/"+pakName+".txt\n");
        Log.v(Integer.toString(pid),"echo \""+result+"\" >>/data/logging/"+pakName+".txt\n");
        out.flush();
        out.write("chmod 666 "+"/data/logging/"+pakName+".txt\n");
        out.flush();
        out.write("exit\n");
        //out.close();
    }
    catch(Exception e)
    {
    }
}
```

圖 7 FeatureLog 函式原始碼

取得 APP 名稱之後，我們也希望能記錄此行為發生的時間，因此我們利用 Android 內建的 API 取得系統時間加入記錄資訊當中並且將此一記錄寫至以此 APP 名稱命名的文件檔中(在我們的例子中寫入至/data/logging/APP 名稱. txt 中)。由於 Android 系統中 uid 為 10000 以下的是代表系統內建程序，我們主要偵測的惡意程式都是第三方 APP，所以對於這些程序我們並不予理會。

修改完成後即可開始準備編譯原始碼，首先將編譯環境建立

```
source build/envsetup. sh
lunch full-eng
```

建立完成後即可開始編譯

```
make -j4
```

編譯完成後我們即可至 out/target/product/generic 目錄底下利用編譯好的 system.img、userdata.img、ramdisk.img 三個檔案導入模擬器建立我們修改好的 Android 監控環境。

4.2 Snort Cross-compiling

為了讓原本只能在 Ubuntu PC 下運行的 snort 移植到 Android 系統中執行來幫助我們監控網路封包是否含有隱私資料的洩漏，我們必須利用 snort 的開源原始碼進行交叉編譯(cross compiling)[19]來將 snort 編譯成 Android 可以執行的 ARM 版本。在這一小截中我們會詳細地解釋如何將 snort 原始碼編譯成 Android 可執行的版本。

為了在 Linux PC 系統環境進行交叉編譯，首先必須先至 Codesourcery 官網下載 sourcery g++ Lite 此一交叉編譯程式並且設定環境變數使的系統能使用其提供的功能指令，之後即可開始進行交叉編譯。

首先我們必須先將 snort 程式所需的 libraries 也都交叉編譯為 Android 系統上可使用的版本，而 snort 需要的 libraries 有 libpcap、libpcre、libdaq、libdnet、zlib、ncurses，至各個 library 以及 snort 官網下載其原始碼，接下來會一一解釋如何對這些 libraries 作交叉編譯。

I. Libpcap

首先必須在 Ubuntu PC 系統環境安裝 flex 以及 bison 套件

```
sudo apt-get install flex bison
```

之後即可對 libpcap 原始碼進行設定編譯參數

```
CC=arm-none-linux-gnueabi-gcc ./configure --host=arm-linux
--prefix=/your/target/directory --with-pcap=linux LDFLAGS="-static"
```

其中”—prefix”選項代表著編譯完成後的 library 的存放目錄位置，設定好參數後即可開始交叉編譯

```
make & make install
```

II. Libpcre

對其原始碼作編譯參數設定

```
CC=arm-none-linux-gnueabi-gcc ./configure --host=arm-linux
--prefix=/your/target/directory LDFLAGS="-static"
CXX=arm-none-linux-gnueabi-g++
```

設定完成後進行交叉編譯

```
make & make install
```

III. Libdaq

對其原始碼作編譯參數設定

```
CC=arm-none-linux-gnueabi-gcc ./configure -- prefix=/your/target/directory
--with-libpcap-includes=snort’s directory/out/include --with-libpcap-libraries=
snort’s directory /out/lib --host=arm-linux LDFLAGS="-static" --enable-static
```

設定完成後進行交叉編譯

```
make & make install
```

IV. Libdnet

對其原始碼作編譯參數設定

```
CC=arm-none-linux-gnueabi-gcc ./configure --host=arm-linux
--prefix=/your/target/directory --with-libpcap-libraries=snort’s
directory/out/lib/ LDFLAGS=”-static” CXX=arm-none-linux-gnueabi-g++
```

設定完成後進行交叉編譯

```
make & make install
```

V. Zlib

對其原始碼作編譯參數設定

```
CC=arm-none-linux-gnueabi-gcc ./configure --prefix=/your/target/directory -
static
```

設定完成後進行交叉編譯

```
make & make install
```

VI. Ncurses

對其原始碼作編譯參數設定

```
CC=arm-none-linux-gnueabi-gcc AR=arm-none-linux-gnueabi-ar
CXX=arm-none-linux-gnueabi-g++ ./configure --host=arm-linux
--target=arm-linux --prefix=/CrossCompile 工具之目錄
/arm-none-linux-gnueabi/libc/usr
```

設定完成後進行交叉編譯

```
make & make install
```

上述的 libraries 交叉編譯完成後使用下列指令對 snort 原始碼進行編譯參數設定

```
CC=arm-none-linux-gnueabi-gcc AR=arm-none-linux-gnueabi-ar
RANLIB=arm-none-linux-gnueabi-ranlib ./configure --host=arm-linux
--prefix=/your/target/directory LDFLAGS="-static"
CXX=arm-none-linux-gnueabi-g++ --with-libpcap-libraries=snort’s
directory/out/lib --with-libpcre-libraries= snort’s directory /out/lib
--with-daq-libraries= snort’s directory /out/lib --with-dnet-libraries= snort’s
```

directory /out/lib --with-mysql-libraries= snort’s directory /out/lib --with-libpcre-includes= snort’s directory /out/include --with-daq-includes= snort’s directory /out/include --with-dnet-includes= snort’s directory /out/include

設定完成後開啟編輯”snort 根目錄/src/Makefile”檔案將

LINK = $(LIBTOOL) --tag=CC $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) \

--mode=link $(CCLD) $(AM_CFLAGS) $(CFLAGS) $(AM_LDFLAGS) \

$(LDFLAGS) -o $@

改成

INK = $(LIBTOOL) --tag=CC $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) \

--mode=link $(CCLD) $(AM_CFLAGS) $(CFLAGS) $(AM_LDFLAGS) \

$(LDFLAGS) **-all-static** -o $@

修改完成並儲存之後進行交叉編譯

make & make install

編譯完成之後我們即可得到可在 Android 系統上執行的 snort 執行檔，之後我們就可在 Android 系統上監控網路封包來記錄 APP 執行期間隱私資料洩漏的行為。

4.3 自動化執行程序

由於收集到的實驗樣本數量眾多，單純依靠人力來完成這次實驗室非常耗時且不切實際的，因此我們另外撰寫了自動化程序來幫助我們完成一些單純、重複的工作，自動化程序主要分為二大部分，主要是以 shell script 撰寫而成，第一部分是將 APP 逐一導入模擬器執行並且將監控行為的記錄檔導出至 Ubuntu PC 上儲存，第二部分是對這些監控行為的記錄檔作整理轉換為向量數據並且集中分配為 training set 和 testing set 兩個檔案。

在第一部分的自動化程序中在我們導入 APP 進入模擬器執行並且監控其行為的過程中，我們必須在檔案系統中創建一個檔案來記錄這些行為的資訊，但是受限於 Linux 核心的使用者權限管理機制，我們無法任意地在我們想要的目錄位置創建、編輯檔案，因此首先我們透過卸載檔案系統，重新更改其讀取權限後在重新掛上檔案系統，使得我們能夠利用 super user 的權限來在檔案系統的任何位置創建、編輯檔案，同時我們也背景執行地啟動 snort 開始監控網路封包。

```
adb shell mount -o rw, remount -t yaffs2 /dev/block/mtdblock0
adb shell chmod 777 /system
adb shell "cat /system/bin/sh > /system/bin/su"
adb shell chmod 5677 /system/bin/su
adb shell /data/snort/snort -l /sdcard/snort_log -c /sdcard/snort. conf &
```

接下來我們就可以開始逐一地導入 APP 進入模擬開始執行，而這一部分主要是利用 Android 系統內建的一個指令”monkey”來達成，此指令可以產生隨機使用者輸入事件來跟 APP 作互動。

```
monkey -p［package name］-s［seed value］--throttle［delay time］100
```

其中”-p”選項是用來指定要執行之 APP 的 package 名稱，”-s”選項是用來指定給 monkey 指令產生亂數輸入時的 seed 值，若沒有指定則指令預設使用 0 為 seed 值，”--throttle”是用來設定每一個輸入事件之間的間隔時間，時間單位為毫秒

(milliseconds)，最後的整數的意思是在這執行過程中所要輸入的事件數量。

那我們如何在 APP 導入模擬器執行之前取得此 APP 的 package 名稱以供”monkey”指令執行使用呢?我們利用了 aapt 工具中的”dump”指令來將 APP 的 APK 檔中的相關的資訊擷取出來。

```
aapt dump badging [filename]
```

執行了”monkey”指令後 Android 系統即會自動地將 private data leakage operation 以及 network leakage operation 記錄下來。

第二部分主要是將這些記錄下來的行為資訊轉換為向量數據，因此首先我們必須先了解記錄下來的資訊格式，接下來能能把需要的資訊擷取出來，下圖 8 和圖 9 分別為我們記錄下來的 private data leakage operation 和 network leakage operation 格式範例

```
2013-0-16 5:30/com.amazon.kindle(1217): READ IMEI
2013-0-16 5:30/com.amazon.kindle(1217): READ phone number
2013-0-16 5:30/com.amazon.kindle(1217): READ SIM serial number
2013-0-16 5:30/com.amazon.kindle(1217): READ SubscriberID
```

圖 8 private data leakage operation 記錄範例

```

[**] [1:1003031:0] IMEI [**]
[Priority: 0]
01/23-01:36:58.659776 10.0.2.15:5555 -> 10.0.2.2:36649
PROTO:006 TTL:64 TOS:0x0 ID:8671 IpLen:20 DgmLen:174 DF
***AP*** Seq: 0x54D12089  Ack: 0x2BA016  Win: 0xFFFF  TcpLen: 20

[**] [1:1003030:0] ISO [**]
[Priority: 0]
01/23-01:36:58.659776 10.0.2.15:5555 -> 10.0.2.2:36649
PROTO:006 TTL:64 TOS:0x0 ID:8671 IpLen:20 DgmLen:174 DF
***AP*** Seq: 0x54D12089  Ack: 0x2BA016  Win: 0xFFFF  TcpLen: 20

[**] [1:1003031:0] IMSI [**]
[Priority: 0]
01/23-01:36:58.673867 10.0.2.15:5555 -> 10.0.2.2:36649
PROTO:006 TTL:64 TOS:0x0 ID:8674 IpLen:20 DgmLen:268 DF
***AP*** Seq: 0x54D12172  Ack: 0x2BA05E  Win: 0xFFFF  TcpLen: 20
```

圖 9 network leakage operation 記錄範例

從上述範例我們可以看到每一筆行為記錄中都有關鍵字可以讓我們看出 APP 執行了何種行為，記錄資訊部分主要根據自己對於行為的敘述不同而有所變化，因此如果讀者想要自己嘗試監控的話可以依照自己的習慣對於每種行為作更清楚的敘述。

4.4 資料轉換以及 Weka 應用

在 MalCatcher 中我們利用了一套機器學習分析的免費軟體”Weka”來幫助我們訓練 MalCatcher 使其擁有偵測惡意程式的能力，在本章節中我們會詳細地解釋如何利用 Weka 對我們收集到的資訊進行分析並且得到偵測惡意程式的判斷能力。

若要利用 Weka 來分析資訊，首先我們必須將這些資訊轉換成 Weka 能接受的檔案格式，也就是 ARFF(Attribute-Relation File Format)檔案格式，範例如圖 6

```
1  @RELATION private
2
3  @ATTRIBUTE ACCESS_LOCATION NUMERIC
4  @ATTRIBUTE CONTACT_NET NUMERIC
5  @ATTRIBUTE CONTACT_READ NUMERIC
6  @ATTRIBUTE IMEI_NET NUMERIC
7  @ATTRIBUTE IMEI_READ NUMERIC
8  @ATTRIBUTE IMSI_NET NUMERIC
9  @ATTRIBUTE IMSI_READ NUMERIC
10 @ATTRIBUTE ISO_NET NUMERIC
11 @ATTRIBUTE ISO_READ NUMERIC
12 @ATTRIBUTE PHONE_NUMBER_NET NUMERIC
13 @ATTRIBUTE PHONE_NUMBER_READ NUMERIC
14 @ATTRIBUTE SIM_NET NUMERIC
15 @ATTRIBUTE SIM_READ NUMERIC
16 @ATTRIBUTE SMS_NET NUMERIC
17 @ATTRIBUTE SMS_READ NUMERIC
18 @ATTRIBUTE SMS_SEND NUMERIC
19 @ATTRIBUTE class {MALWARE,NORMAL}
20
21 @DATA
22 0,0,0,0,2,0,2,285614,0,0,0,0,0,0,0,0,MALWARE
23 0,0,0,0,2,0,2,66,0,0,0,0,0,0,0,0,MALWARE
24 0,0,0,0,0,0,0,62,0,0,0,0,0,0,0,0,MALWARE
25 0,0,0,0,0,0,0,62,0,0,0,0,0,0,0,0,MALWARE
26 0,0,0,0,3,0,0,32,0,0,0,0,0,0,0,0,MALWARE
27 0,0,0,0,2,0,2,40,0,0,0,0,0,0,0,0,MALWARE
28 0,0,0,0,0,0,0,26,0,0,0,0,0,0,0,0,MALWARE
29 0,0,0,0,2,0,2,48,0,0,0,0,0,0,0,0,MALWARE
30 0,0,0,0,2,0,2,40,0,0,0,0,0,0,0,0,MALWARE
31 0,0,0,0,0,0,0,12,0,0,0,0,0,0,0,0,MALWARE
32 0,0,0,0,2,0,2,42,0,0,0,0,0,0,0,0,MALWARE
33 0,0,0,0,2,0,2,96,0,0,1,0,1,0,0,0,MALWARE
34 0,0,0,0,2,0,2,96,0,0,1,0,1,0,0,0,MALWARE
35 0,0,0,1,2,2,2,390,0,0,1,0,1,0,0,0,MALWARE
36 0,0,0,0,2,0,2,78,0,0,1,0,1,0,0,0,MALWARE
37 0,0,0,0,2,0,2,152,0,0,1,0,1,0,0,0,MALWARE
```

圖 10 ARFF 格式範例檔

“@RELATION”這項屬性是用來表示在此一檔案中的資訊是有關於何事何物，也可以想像成是檔案名稱，對於 Weka 分析不會造成任何影響，只是能方便使用者能快速地從這項屬性了解這些資訊的來源。”@ATTRITUBE”這項屬性主要是宣告讓 Weka 了解在此檔案中的每筆資料中的各個元素是代表什麼意義，例如第三行”@ATTRIBUTE SMS_NET NUMERIC”宣告了每筆資料中的第一個元素是表示簡訊經由網路洩漏出去的行為並且此元素是數字，同時由於我們為了讓 MalCatcher 能透過多筆已知類別的資料來訓練得到偵測惡意程式的能力，因此在最後我們額外多加了一個屬性，也就是圖 10 中的第 19 行”@ATTRIBUTE class

{MALWARE, NORMAL}”，宣告了一項名為”class”的屬性並且此屬性的質只能是”MALWARE”或是”NORMAL”，透過這項屬性我們能告訴 Weka 每筆資料是屬於哪一類別的記錄，”MALWARE”表示是屬於惡意程式，”NORMAL”表示是屬於正常 APP。”@DATA”這項屬性表示了在這一屬性之後的文字是需要被分析的資料，以圖 10 為例，在第 21 行”@DATA”以下的每一行文字都代表著一筆需要被分析的資料。

將資料轉換為 Weka 可接受分析的檔案格式之後，接下來至 Weka 官網下載 Weka 最新的主程式並且安裝，為了能使用 Weka 中的”LibSVM”演算法，至 http://www.csie.ntu.edu.tw/~cjlin/libsvm/ 此一網址下載由第三方作者撰寫 LibSVM 壓縮檔並解壓縮，之後將其中的 libsvm.jar 以及 wlsvm.jar 複製到 Weka 安裝目錄中，利用記事本之類的軟體開啟 Weka 安裝目錄中的”RunWeka.ini”將

```
cmd_default=javaw -Dfile.encoding=#fileEncoding# -Xmx#maxheap# #javaOpts#
-classpath "#wekajar#;#cp#" #mainclass#
```

改成

```
cmd_default=javaw -Dfile.encoding=#fileEncoding# -Xmx#maxheap# -classpath
"#wekajar#;wlsvm.jar;libsvm.jar;#cp#" #mainclass#"
```

之後再將

```
cmd_console=cmd.exe /K start cmd.exe /K "java -Dfile.encoding=#fileEncoding#
-Xmx#maxheap# -classpath "#wekajar#;#cp#" #mainclass#"
```

改成

```
cmd_console=cmd.exe /K start cmd.exe /K "java -Dfile.encoding=#fileEncoding#
-Xmx#maxheap# -classpath "#wekajar#;wlsvm.jar;libsvm.jar;#cp#" #mainclass# "
```

完成之後透過 RunWeka.bat 執行 Weka 程式並且點即”Exploer”選項出現主畫面如下圖 11，透過”Open file…”選項來選擇想要用來訓練 MalCatcher 的 ARFF 格式檔案，選擇完成之後左下方即會出現此檔案中所宣告的所有屬性列表，同時右

下方會顯示出此一屬性在這一群資料中的狀況，我們可以透過右下方的列表對此檔案中的屬性進行刪除、修改，接下來透過畫面上方標籤移動到”Classify”畫面，如圖 12，在此一頁面中我們可以利用 Weka 程式提供的多種不同的機器學習演算法來分析導入的資料，訓練 MalCatcher 偵測惡意程式的能力，透過 Classifer 下的”Choose”選向來選擇想要使用的演算法，選擇完成之後可以點擊右邊的輸入方框來對演算法做進一步的參數設定，設定好之後為了確保訓練的效果，使用了 Cross-validation 選項其中數值預設為 10，選擇完成之後即可透過”Start”選向來分析資料訓練 MalCatcher，執行的結果會顯示在畫面右邊的視窗之中，訓練完成後我們可以透過”Supplied test set”選向來選擇 testing set 檔案，之後從右下方的視窗中隊相對應的演算法點擊右鍵並且選擇”Re-evaluate model on current test set”來對 testing set 資料進行分類測試，同樣地結果會顯示在右方視窗中。

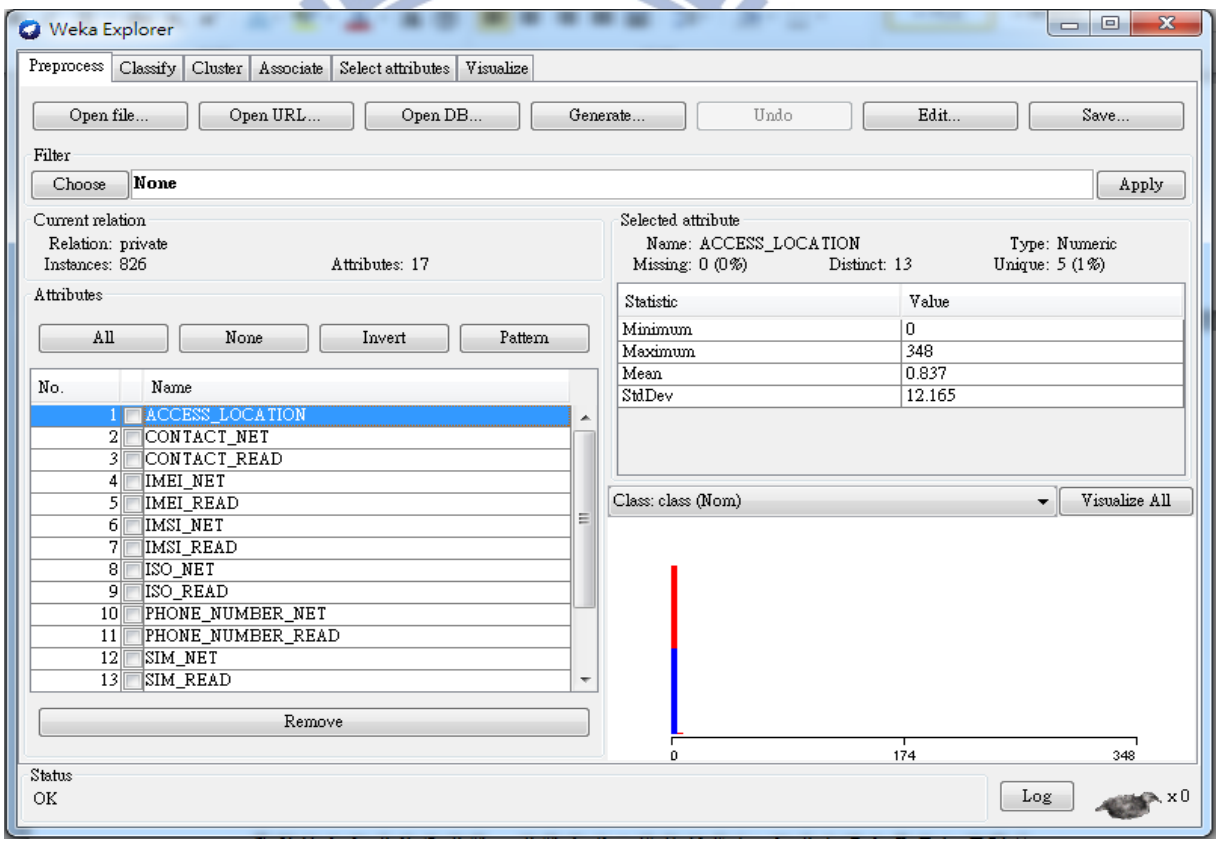


圖 11 Weka 程式 Preprocess 畫面

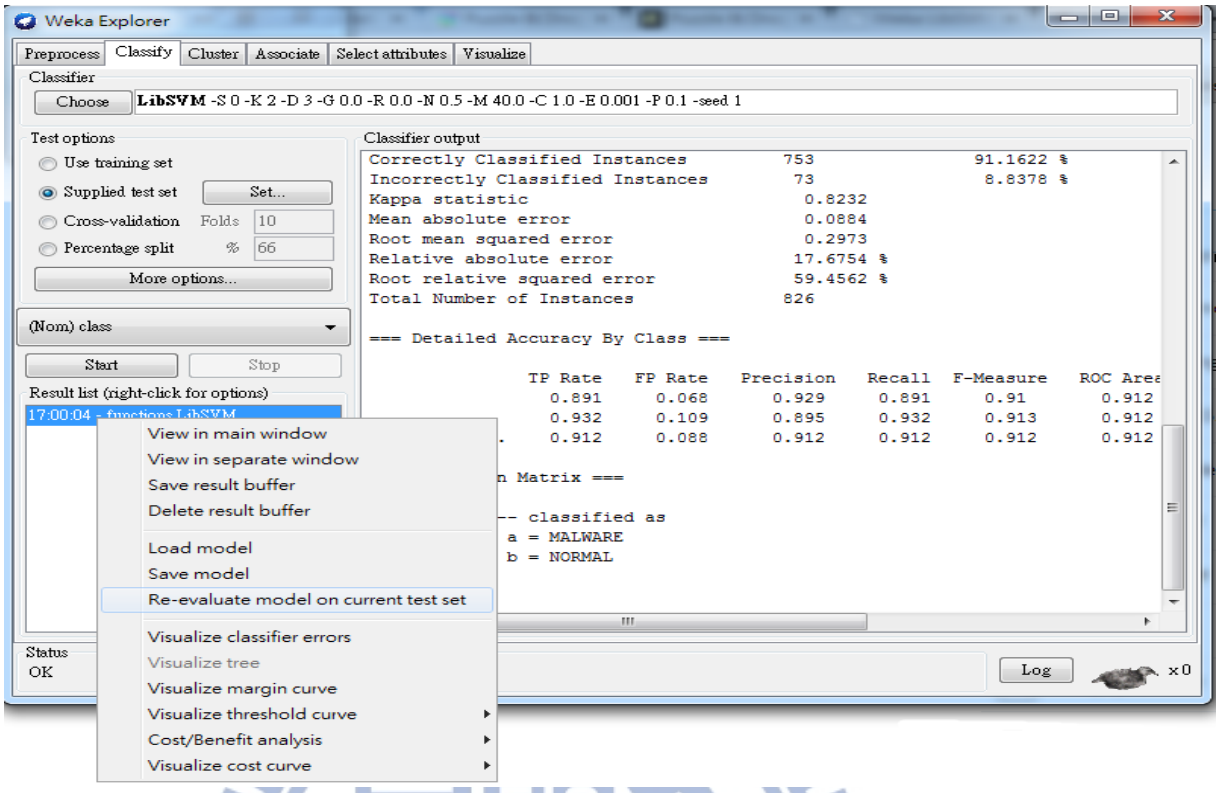


圖 12 Weka 程式 Classify 畫面

五、 實　　驗

本篇論文實驗的目的為藉由已經過修改的 Android 系統運行在模擬器上以及安裝 snort 網路封包監控軟體，在模擬器上執行 APP 來記錄 APP 執行期間 private data leakage operation 以及 network leakage operation 執行狀況，並且以這些資訊來去判斷 APP 是否為惡意程式。同時我們也實作了 Iker Burguera 等的 Crowdroid[11]方法來比較 MalCatcher 以及 Crowdroid 兩者的判斷準確度。

在這次實驗中，收集到了 1260 個真實的惡意程式以及 825 個正常的 APP 來完成這次實驗，非常感謝 Yajin Zhou, Xuxian Jiang 提供了惡意程式以及陶嘉仁[13]提供了正常的 APP。

在這次實驗中採用了多種的機器學習演算法來去觀察何種機器學習演算法對於本次的實驗的判斷準確度有較大的幫助。

5.1 實驗環境介紹

本次實驗環境主要分為三個部分，Android 模擬器端、Linux 端、Windows 端，在接下來的小節會針對這三大部分做說明並且詳細地說明如何建置本次實驗的環境。

5.1.1 Android 模擬器端

在本次實驗中使用了 Android 4.0.3 CyanogenMod 系統版本，並且修改了其原始碼加入了自定義的記錄函式以及搭配使用了 snort 來記錄 private data leakage operation 以及 network leakage operation，詳細記錄如表 1。

另一方面，由於我們也常適時做出 Crowdroid 的方法，因此也必須監控 APP 執行期間的 system call 執行狀況。我們在模擬器中同時利用了一 Linux 指令”strace”[20]來監控並記錄 system call 執行狀況，Android 系統中的 system

call 種類在 Android 系統源始碼中的 bionic/libc/SYSCALLS.TXT 有詳細地記載。

5.1.2 Linux 端

在 PC 上我們採用了 Ubuntu 12.04 64bits 的 Linux 系統版本，CPU 為 Intel i5-3570，RAM 的大小為 8GB，硬碟空間大小為 2TB，並且在其上安裝好 Java JDK7、C/C++、Android SDK 以及 Eclipse 等工具，在 Linux 系統上的主要工作是將前置設定、將 APP 導入模擬器中執行和記錄資訊處理的自動化。

5.1.3 Windows 端

在另一 PC 上我們安裝了 Windows7 64bits 的 Windows 系統版本，並且安裝最新 Weka 機器學習分析軟體，在 Windows 端上最主要的工作為將已整理好並且已轉換為 ARFF 檔案格式的記錄資訊檔案導入 Weka 中執行機器學習演算法。

5.2 實驗環境建置

首先到 Ubuntu 官網[21]下載 Ubuntu12.04 64bits 系統版本的安裝映像檔並且安裝系統，接下來打開終端機開始安裝建置 Java 環境，將 http://www.duinsoft.nl/pkg 此一網站加入 Ubuntu 套件來源

```
sudo sh -c "echo 'deb http://www.duinsoft.nl/pkg debs all' >> /etc/apt/sources.list"
```

重新載入 Ubuntu 套件來源清單

```
sudo apt-get update
```

```
sudo apt-key adv --keyserver keys.gnupg.net --recv-keys 5CB26B26
```

sudo apt-get update

開始安裝 Java JDK 7

sudo apt-get install update-sun-jre

設定 Java JDK 7 相關環境

sudo mkdir -p /usr/lib/jvm/

sudo cp -R /usr/local/jdk1.7.0* /usr/lib/jvm/

確保系統使用了 Java JDK 7

sudo update-alternatives --install /usr/bin/javac javac /usr/lib/jvm/jdk1.7.0_07/bin/javac 1

sudo update-alternatives --install /usr/bin/java java /usr/lib/jvm/jdk1.7.0_07/bin/java 1

接下來至 Android 官網[22]下載 Android SDK

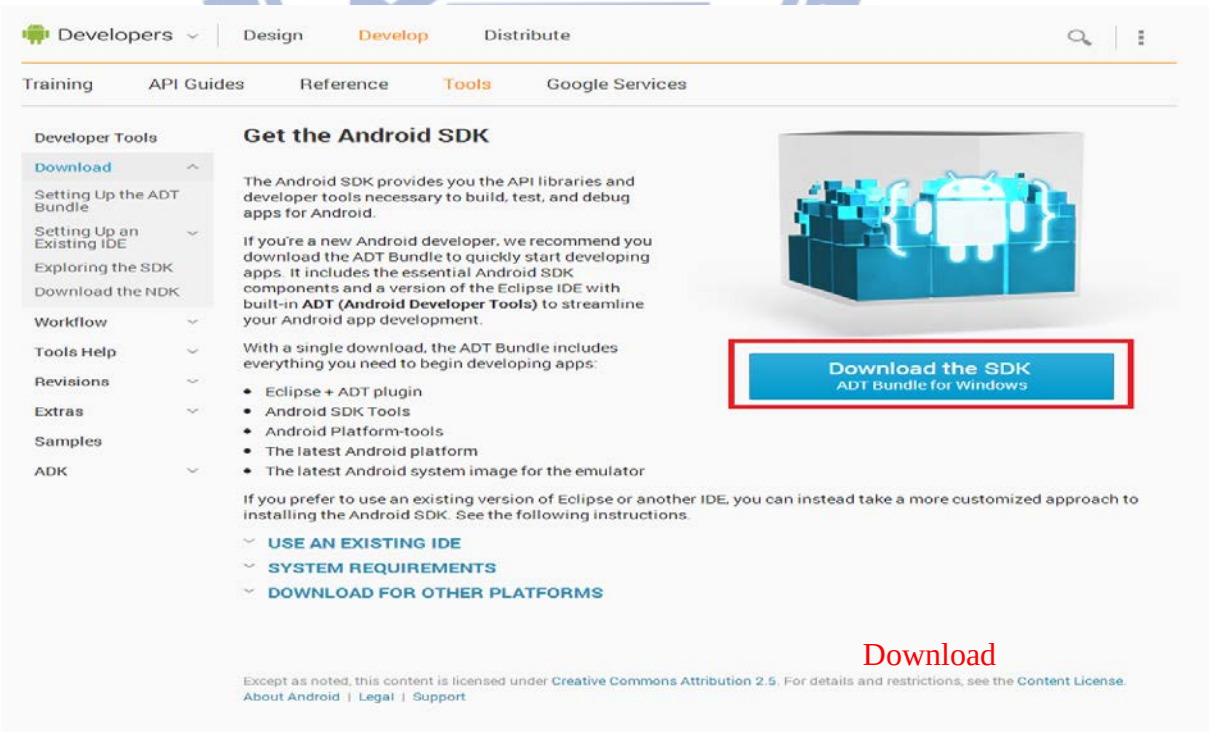


圖 13 Android SDK 官網下載點(來源:[22])

下載完成後解壓縮至使用者家目錄，之後開啟使用者家目錄底下的.bashrc 設定環境變數

```
gedit ~/.bashrc
```

在檔案最後加入

```
export PATH=$PATH:~/adt-bundle-linux-x86_64/sdk/tools: ~/adt-bundle-linux-x86_64/sdk/platform-tools
```

儲存後關閉檔案，開啟終端機並執行下列指令開啟 Android sdk 管理介面(如圖 14)

```
android sdk
```

勾選並下載安裝 Android4.0.3 系統平台相關檔案

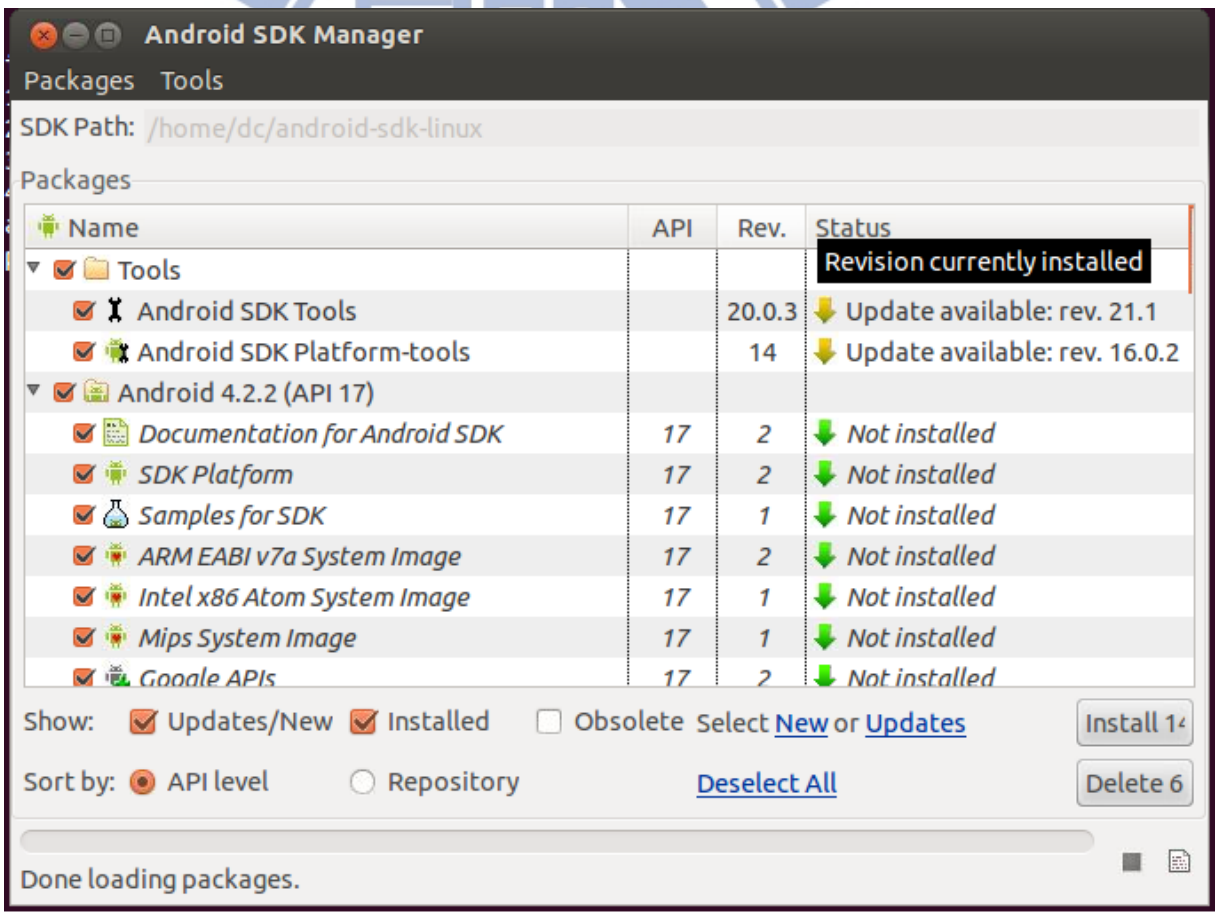


圖 14 Android SDK 管理介面

開啟 Android 模擬器管理介面(如圖 15)

```
android avd
```

新增一 Android4.0.3 系統版本之模擬器，模擬器名稱在本實驗中為 testemu

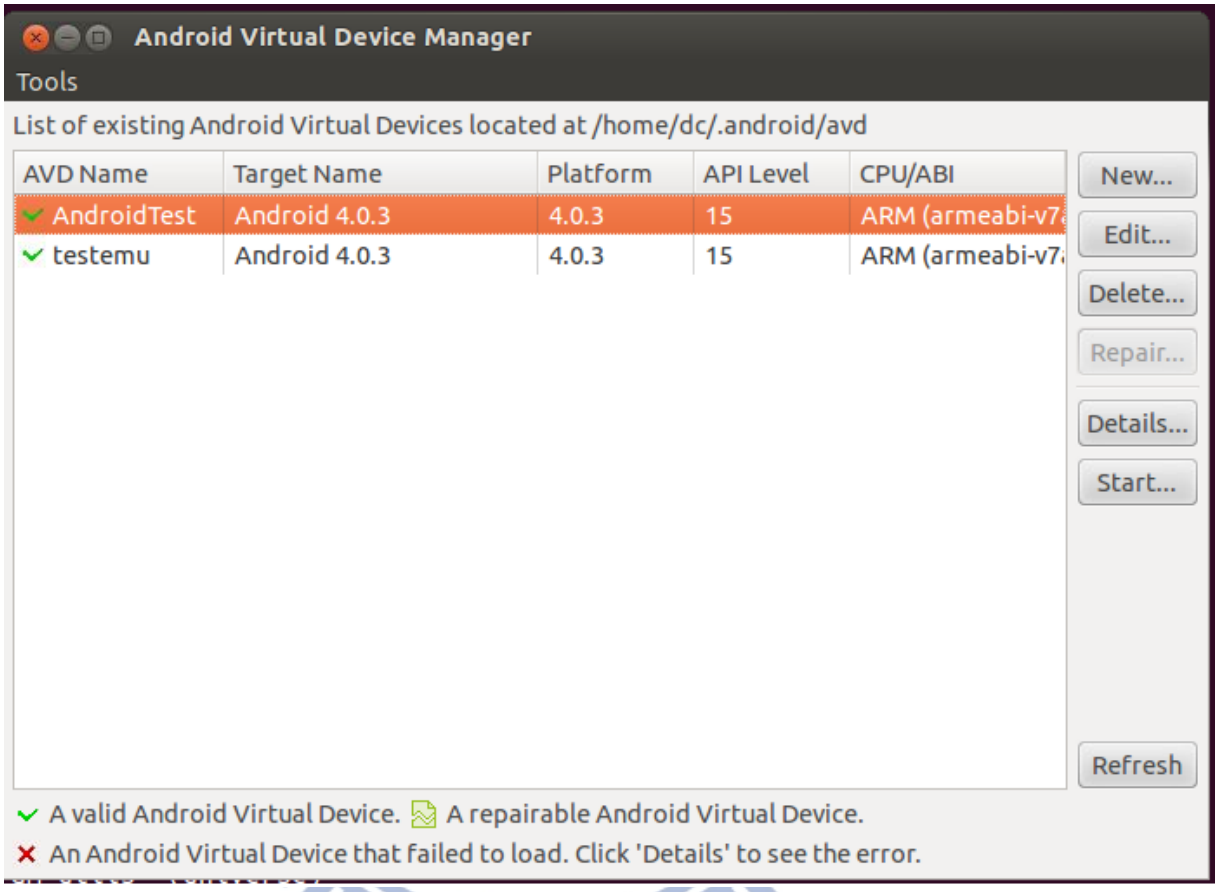


圖 15 Android 模擬器管理介面

將已經撰寫好的程式以及實驗樣本放置在使用者家目錄中，詳細檔案列表如下

檔案/目錄名稱	說明
autotest.sh	自動化地將 samples 中所有的 Malware APP 一個個依序地導入模擬器中執行

autotest_normal. sh	自動化地將 Normal_APP 中所有 normal app 一個個依序地導入模擬器中執行
samples	存放所有惡意程式樣本
Normal_APP	存放所有正常 APP 樣本
CyanogenMod	存放修改過後的 Android system images
SysVector	將記錄下來的 APP 執行 system call 資訊整理成 vector
PrivateGen	將記錄下來的 APP 執行 private data leakage operations 整理成 vecotr
PrivArffTrans	將 Private data leakage operations vector list 轉換成 Weka 可讀的 arff 檔格式
ArffTrans	將 System call vecotor list 轉換成 Weka 可讀的 arff 檔格式
snort	可在 Android 上執行的 snort 執行檔
snort. conf	Snort rule 設定檔

NetworkMonitor. apk	產生詳細 snort rule 檔的 APP

表 2 MalCatcher 目錄以及檔案列表與說明

粗體表示是一目錄資料夾，開啟終端機並且切換到 CyanogenMod 目錄餅且執行下列指令來藉由已經修改過的 Android 系統映像檔來啟動模擬器

```
emulator -avd testemu -system system. img -data userdata. img -ramdisk ramdisk. img &
```

執行下列指令來進入 Android 模擬器的 command console 並且建立 /data/logging/snort 以及/sdcard/snort_log 還有/sdcard/rules 三個目錄，其中 /data/logging/snort 是用來放置 snort 執行檔的地方，/sdcard/snort_log 式之後 snort 監控網路封包後記錄資訊儲存的位置，/sdcard/

```
adb shell
mkdir /data/logging/snort
mkdir /sdcard/snort_log
mkdir /sdcard/rules
```

安裝產生 snort rule 的 APP: NetworkMonitor. apk

```
adb install NetworkMonitor. apk
```

將 snort 執行檔 **snort** 以及 snort 設定檔 **snort. conf** 各別放入模擬器中的**/data/snort** 以及**/sdcard** 目錄中，並且將 **snort** 權限改為 777

```
adb push snort /data/logging/snort/snort
adb push snort. conf /sdcard/snort. conf
adb shell chmod 777 /data/snort
```

操作模擬器執行 NetworkMonitor APP，點選**設定規則**來存取模擬器中使用者相關

的隱私資料並且產生相對應的 snort rules，畫面如圖 16。

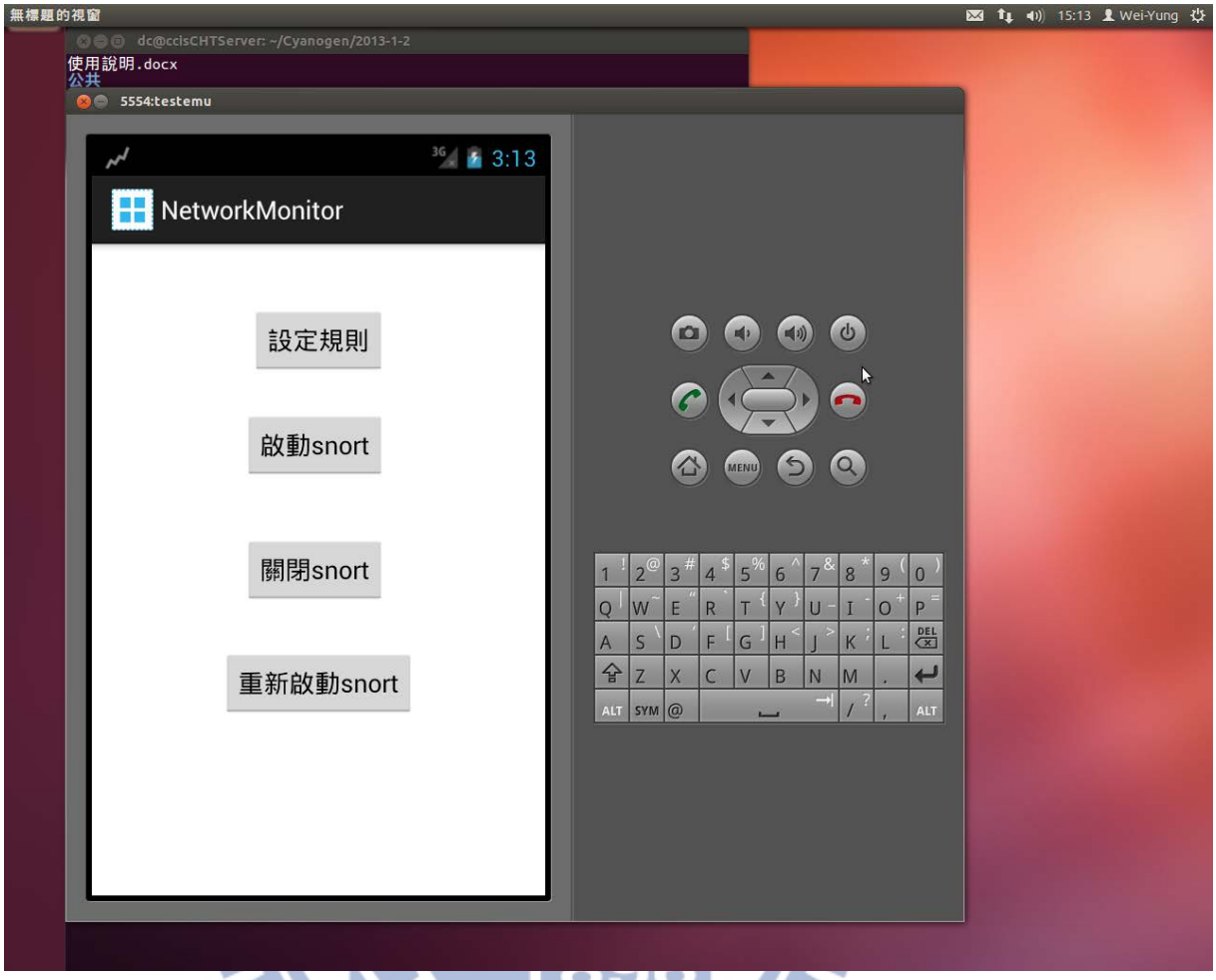


圖 16 NetworkMonitor 介面

經由以上步驟即可將本實驗的實驗環境建立完成，之後可執行 autotest_normal.sh 開始自動化地導入正常 APP 進模擬器執行或者執行 autotest.sh 開始自動化地導入惡意程式進入模擬器執行。

5.3 實驗樣本

在這次實驗中非常感謝 Yajin Zhou, Xuxian Jiang[12]提供大量的惡意程式樣本以及陶嘉仁[13]提供大量的正常 APP 樣本來進行實驗。

正常 APP 根據 Google Play 對 APP 的分類，總共分為 18 類，詳細如表 3，主要是從各 18 大類的免費熱門下載排行榜下載而來[13]，因此正常 APP 實驗樣本皆

為目前市面上使用者很常使用的 APP。

類別名稱	APP 數量
Books	17
Business	44
Communication	70
Education	26
Entertainment	50
Finance	50
Game	46
Media	39
Music	54
News	27
Personalization	44
Photography	39
Productivity	78
Shopping	35
Social	48
Tool	41
Transportation	66
Travel	51
Total	**825**

表 3 各類別正常 APP 數量分布表

惡意程式根據攻擊目的以及攻擊方法的不同，分為 49 類[12]，每一類的名稱主要是以此一類攻擊中第一個被發現的惡意程式名稱來命名，詳細列表如表 4

類別名稱	APP 數量
ADRD	22
AnserverBot	187
Asroot	8
BaseBridge	122

BeanBot	8
Bgserv	9
CoinPirate	1
CruseWin	2
DogWars	1
DroidCoupon	1
DroidDeluxe	1
DroidDream	16
DroidDreamLight	46
DroidKungFu1	34
DroidKungFu2	30
DroidKungFu3	309
DroidKungFu4	96
DroidKungFuSapp	3
DroidKungFuUpdate	1
EndofDay	1
FakeNetflix	1
FakePlayer	6
GamblerSMS	1
Geinimi	69
GGTracker	1
GingerMaster	4
GoldDream	47
Gone60	9
GPSSMSSpy	6
HippoSMS	4
Jifake	1
jSMSHider	16
Kmin	52
LoveTrap	1
NickyBot	1

NickySpy	2
Pjapps	58
Plankton	11
RogueLemon	2
RogueSPPush	9
SMSReplicator	1
SndApps	10
Spitmo	1
Tapsnake	2
Walkinwat	1
YZHC	22
zHash	11
Zitmo	1
Zsone	12
Total	**1260**

表 4 各類別惡意程式數量分布表

在本實驗中主要使用了 supervised machine learning algorithm 來訓練以及測試 MalCatcher，因此我們將所有的實驗樣本分為兩大部分，分別為 training set 以及 testing set，其中每一個 set 中皆含有 412 個正常 APP 樣本以及 412 個惡意程式樣本，總共 824 個樣本。Training set 主要是用來執行 machine learning algorithm 訓練 MalCatcher 學習惡意程式與正常 APP 之間的不同特徵來判斷這兩類。Testing set 主要用來測試訓練完成後的 MalCatcher 實際對於惡意程式的判斷準度的優劣。

由於不管是正常 APP 或是惡意程式，其類別都非常繁多而且每一類別之間的 APP 行為也會有一定程度的不同，根據陶嘉仁研究中提到[13]，同一類別 APP 所宣告的 permissions 常常會有很大的相似度，不同類別之間的 APP 所宣告的 permissions 可能差異就會比較大。由此我們判斷同一類別的 APP 行為應該也會有一定的相似度，而不同類別的 APP 之間的行為可能就會有比較大的差異。因此我們相信不管是 private data leakage operation 或是 network leakage operation 在同類別的 APP 之間的呼叫種類以及數量也是會有很大的相似度。

為了讓經由 machine learning algorithm 的 MalCatcher 的判斷準確度盡量地提高，我們希望讓 training set 中的樣本數能夠有足夠多的數量並且盡量包含到所有的類別，使得 MalCatcher 能盡可能的學習到每個類別的行為特徵，但是也不能分配過多樣本到 training set，因為如果 testing set 中的樣本數太少的話可能會導致測試 MalCatcher 判斷準確度時無法確實地表現出來。

基於上述原因，我們希望不管是正常 APP 或是惡意程式都平均地分配到 training set 以及 testing set，也就是說 training set 和 testing set 擁有相同數量的各類別正常 APP 樣本以及惡意程式樣本。如此一來，不管是 training set 或是 testing set 中都有足夠多的樣本數量，而且也涵蓋到了所有的類別，相信這樣在訓練和測試 MalCatcher 偵測惡意程式行為的判斷準確度時都能達到不錯的效果。

針對正常 APP 以及惡意程式，我們個別對這兩類的樣本標上標號，奇數的樣本分配到 training set，偶數的樣本分配到 testing set。如此一來就能使得 training set 以及 testing set 中擁有相同數量的各類別正常 APP 樣本以及惡意程式樣本。

5.4 實驗方法

根據 Yajin Zhou, Xuxian Jiang 的研究發現[12]，惡意程式常常會使用到正常 APP 不常使用的權限功能，如寄送簡訊、存取通訊錄等，也得知了惡意程式的目的主要是為了從受感染的手機上取得利益(如寄送小額付費簡訊)或是竊取手機中的使用者隱私資料，因此惡意程式相較於正常 APP 有較大的可能會去存取較多使用者隱私資料。基於這樣的特性，在此實驗中我們記錄了 APP 執行期間的 private data leakage operation 以及 network leakage operation，希望藉由這些資訊能讓 MalCatcher 有效地偵測惡意程式。

一開始我們希望觀察看看單純監控 private data leakage operation 來偵測惡意程式的效果如何，但是由於正常 APP 中也有許多的 APP 會存取使用者隱私資料，如果單純監控 private data leakage operation 來偵測惡意程式很有可能會造成誤判，因此接下來我們同時監控 private data leakage operation 以及 network leakage operation 來觀察 MalCatcher 偵測惡意程式的效果有沒有比單純監控 private data leakage

operation 的方法效果來得好。

在 Iker Burguera 等人提出的 Crowdroid 一文中[11]，提到其監控 APP 執行期間的 Linux system call 執行狀況來偵測惡意程式的方法，經過他們的實驗得到的判斷準確度非常地不錯，但是文中所使用的實驗樣本數太少並無法得知此方法是否真的能有效地偵測出目前已知的惡意程式，因此我們同時實作了 Crowdroid 所使用的方法並且利用本次實驗中大量實際的惡意程式以及正常 APP 樣本來查看 Crowdroid 的判斷準確度實際上的效果如何，並且跟 MalCatcher 的判斷準確度做比較。

此外我們也嘗試著另外一種可能，根據我們的推測，由於 Linux system call 的種類太過繁多，因此單純監控 Linux system call 來偵測惡意程式的效果可能普普通通，因此我們試著同時監控 Linux system call 以及 network leakage operation 來查看此種方法的判斷準確度的效果如何。

綜合上述所提到的方法，本次實驗總共有五大部分，分別如下

i. Private data leakage operation

ii. Private data leakage operation + network leakage operation

iii. System call operation

iv. Network leakage operation

v. System call operation + network leakage operation

我們也嘗試著對這五大部分的實驗使用不同的 machine learning algorithms 來觀察各種演算法套用在我們的方法上的效果如何，詳細使用了那些演算法如下表(表 5)

Algorithm
NaiveBayes
Adaboost+NaiveBayes
REPTree
Adaboost+REPTree
K-nearest neighbors
Adaboost+K-nearest neighbors
SVM

表 5 應用的 machine learning algorithms 列表

每個 APP 樣本導入模擬器的執行時間為 5 分鐘，過程完全自動化，主要是透過 Android 模擬器中”Monkey”指令來產生隨機的使用者輸入事件來與 APP 互動，總共產生 100 個使用者輸入事件，每個事件的間隔時間為 3 秒鐘。在 Weka 設定方面，每個 machine algorithm 執行時會以 training set 進行 cross-verification(10 回合)來訓練系統，訓練完成後，再以 testing set 進行測試。

5.5 實驗結果

本次實驗結果主要分為四大部分，分別為 false negative rate、false positive rate、total accuracy rate，false negative 指的是在 testing set 的惡意程式樣本集合中被 MalCatcher 誤判為正常 APP 的比例，false positive 指的是在 testing set 的正常 APP 樣本集合中被 MalCatcher 誤判為惡意程式的比例，total accuracy 指的是在 testing set 中的所有樣本中被 MalCatcher 正確分類的樣本比例，也就是說原本是惡意程式樣本確實被分類為惡意程式以及原本是正常 APP 確實被分類為正常 APP 佔所有 testing set 中的實驗樣本比例。

我們針對上一小節所說明的五大部分實驗分別作出了上述所提到的 false negative rate、false positive rate、total accuracy rate 的實驗數據，並且製作出了比較表(表 6、表 8 以及表 7)還有比較圖表(圖 17、圖 18 以及圖 19)

False Negtive Compare

Algorithm	MalCatcher	Crowdrdoid	PrivateDataLeakage	Network Leakage	System Call+Network Leakage
NavieBayes	3.87%	7.99%	67.80%	3.87%	6.78%
Adaboost(NavieBayes)	5.33%	8.23%	67.07%	3.87%	5.57%
REPTree	3.63%	27.85%	65.38%	5.81%	2.91%
Adaboost(REPTree)	3.15%	21.79%	64.65%	6.05%	2.66%
KNN	8.47%	23.24%	62.71%	7.50%	13.80%
Adaboost(KNN)	6.78%	25.67%	61.50%	6.05%	19.85%
SVM	9.93%	11.38%	64.65%	8.47%	80.87%

表 6 False Negative Rate 比較表

False Positive Compare

Algorithm	MalCatcher	Crowdroid	PrivateDataLeakage	Network Leakage	System Call+Network Leakage
NavieBayes	12.59%	66.83%	7.51%	16.46%	36.32%
Adaboost(NavieBayes)	11.86%	64.65%	9.44%	15.01%	19.61%
REPTree	10.90%	19.61%	4.84%	11.38%	12.11%
Adaboost(REPTree)	10.65%	15.98%	5.08%	12.35%	4.12%
KNN	14.29%	22.52%	8.72%	12.35%	23.24%
Adaboost(KNN)	16.95%	22.28%	5.60%	15.01%	22.03%
SVM	9.69%	76.27%	5.81%	9.20%	0.48%

表 7 False Positive Rate 比較表

Total Accuracy					
Algorithm	MalCatcher	Crowdroid	PrivateDataLeakage	Network Leakage	System Call+Network Leakage
NavieBayes	91.89%	62.59%	62.35%	89.83%	78.45%
Adaboost(NavieBayes)	91.40%	63.56%	61.74%	90.56%	87.41%
REPTree	92.74%	76.27%	64.89%	91.40%	92.49%
Adaboost(REPTree)	93.10%	81.11%	65.13%	90.80%	96.61%
KNN	88.62%	77.12%	64.29%	90.07%	81.48%
Adaboost(KNN)	88.14%	76.03%	66.46%	89.47%	79.06%
SVM	90.20%	56.17%	64.77%	91.16%	59.32%

表 8 Total Accuracy 比較表

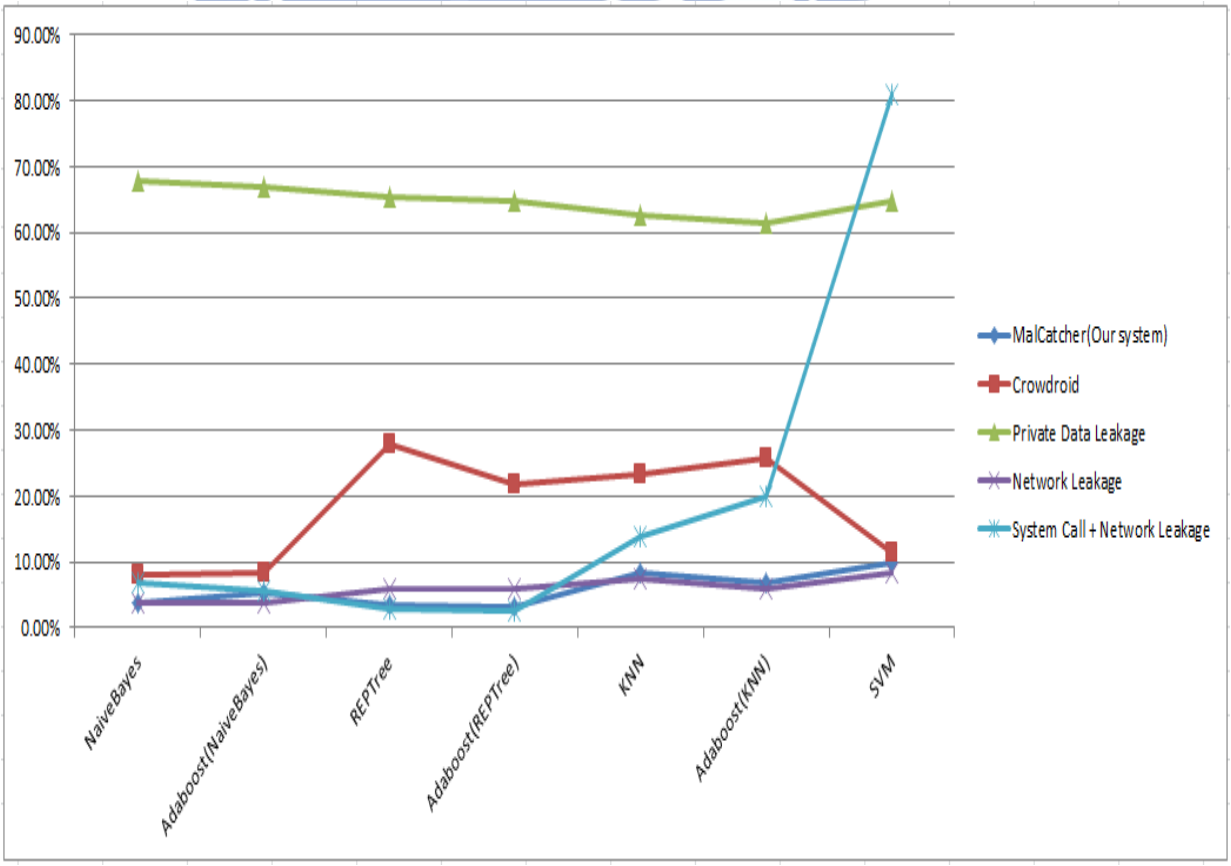


圖 17 False Negative Rate 比較圖

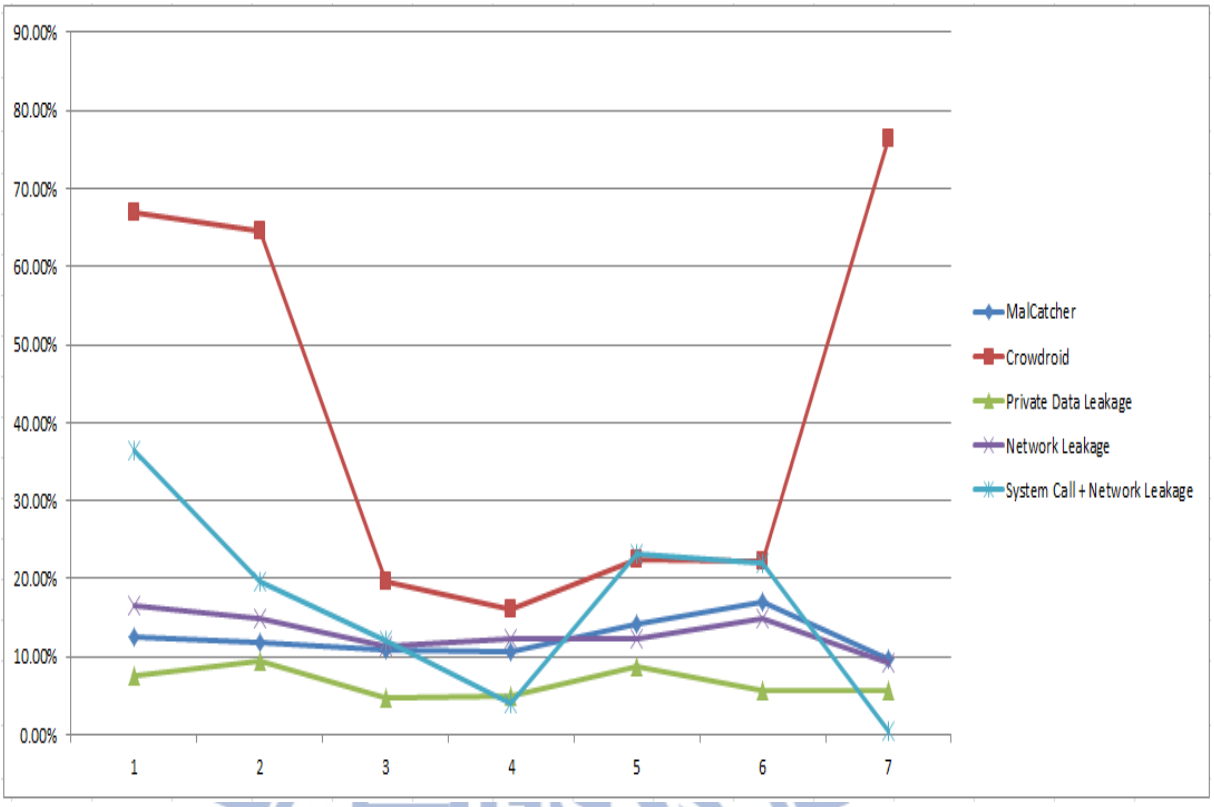


圖 18 False Positive Rate 比較圖

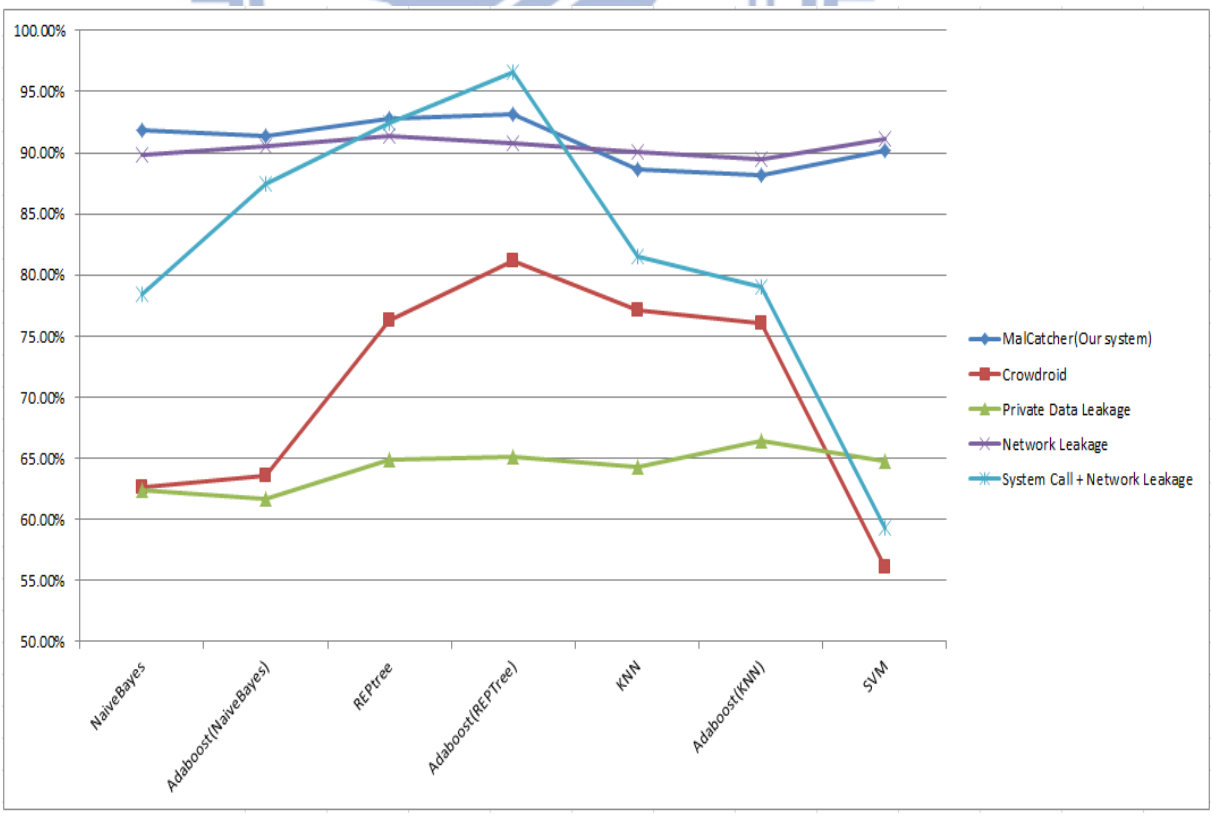


圖 19 Total Accuracy 比較圖

5.6 結果分析與討論

在這一小節我們針對不同行為監控之間的比較以及不同機器學習演算法之間的比較這兩大部分作分析與討論，從中觀察 MalCatcher 在不同的行為監控以及不同的機器學習演算法下訓練出來的效果會有什麼樣不同。

5.6.1 不同行為監控之間比較

以 total accuracy 來看，單純看 system call 的判斷準確度大約 6~7 成左右，雖然有一定程度可以偵測出惡意程式，但是還是有很大的改善空間，這也代表了正常 APP 和惡意程式的 system call 的執行行為有些相似，無法很有效地利用機器學習演算法訓練 MalCatcher。

另一方面，在大部分的機器學習演算法下，單純看 system call 的判斷準確度還比單純看 private data leakage operation 要好，這代表著惡意程式跟正常 APP 存取隱私資料的行為的相似度超過我們的預期，也間接地證明了先前推測可能發生的問題，也就是存取很多的隱私資料並不代表就是惡意程式，許多正常 APP 也常常會去存取隱私資料，因此單純透過 private data leakage operation 的監控來偵測惡意程式是不可行的，有非常大的機率會造成誤判。

透過 network leakage operation 的監控來偵測惡意程式的效果相較前面所提到的兩個方法要好上許多，基本上都可以達到約 9 成的判斷準確度。這也證實了我們先前的推測，也就是若攻擊者想取得使用者儲存在手機上的隱私資料則必須透過網路將這些隱私資料傳出取得。

因為惡意程式的攻擊目的有些並不一定只想竊取使用者儲存在手機中的隱私資料，也有可能想透過背景地替使用者寄送小額付費簡訊至特定的收費號碼來取得利益等，因此我們也嘗試著同時監控 private data leakage operation 以及 network leakage operation 的行為偵測惡意程式來查看是否有更進一步地改善判斷的準確度，我們得到的實驗結果也顯示了同時監控這兩種行為的效果又更加地改善了判斷準

確度，使得 MalCatcher 最高能達到約 9 成 3 的準確度。

另外，上述有提到單純看 system call 的效果雖然不是很好但是還是有一定的判斷準確度，所以我們試著結合判斷效果較好的 network leakage operation 來查看是否能改善監控 system call 行為的判斷準確度，我們的實驗結果也顯示了確實大大地改善了單純監控 system call 行為的判斷準確度，由原本 6~7 成的準確度提升到了 7~8 成接近 9 成的準確度，甚至在特定的機器學習演算法終能達到 9 成 3 的準確度，由此可見，network leakage operation 的行為監控在偵測惡意程式中扮演著非常重要的角色。

雖然 system call 加上 network operation 的行為在特定演算法下能達到比 private data leakage 加上 network leakage operation 的效果好，但是 private data leakage operation 中包含的特徵行為總共有 9 個，而 system call 中包含的特徵行為共有 216 個，需監控的特徵行為數量相差非常多，這可能會在造成在執行機器學習演算法分析上的時間效率變慢，因此我們建議在利用 private data leakage operation 加上 network leakage operation 的行為來偵測惡意程式就已足夠。

5.6.2 不同機器學習演算法之間比較

在 NaiveBayes 演算法下，由於 NaiveBayes 演算法必須每個特徵行為事件必須互相獨立、互不相干的才能發揮其演算法的效果，不管在 system call 或是 private data leakage operation 或是 network leakage operation 這些監控行為中，每個特徵行為之間的相依性不高，因此在 NaiveBayes 演算法下表現出來的判斷準確度相較其他演算法不會比較差。

在 REPTree 演算法下，因為 REPTree 演算法透過所有的特徵行為資訊建立了決策樹之後，會進行 reduced error pruning 的動作將一些不會改善甚至惡化分類結果的 subtree 刪除掉，使得在不會降低判斷準確度的情況下大大地縮小了決策樹的大小，同時達到了執行效率以及判斷準確度的提升，因此跟其他演算法相較之下有著較好的判斷準確度。

在 KNN 演算法下(其中 K 在我們實驗中設定為 5),判斷準確度也相當地不錯，由此我們可以推論這是由於惡意程式雖然種類繁多，每一類中常常有多個樣本可能進行同樣的攻擊，但可能為了規避目前已知的偵測方法而演化出了多個不同手法來去逃過這些偵測機制，因此使得惡意程式的行為模式非常相似而產生聚集的狀況。另一方面,我們也可以推論由於正常 APP 之間的行為跟惡意程式的行為相較之下，正常 APP 之間的行為模式較相似，所以正常 APP 的行為也產生了聚集的現象，因此利用 KNN 演算法來訓練 MalCatcher 也能得到不錯的效果。

在 SVM 演算法下，雖然從 KNN 演算法看來正常 APP 和惡意程式的行為都會有聚集的現象，但是可能由於正常 APP 與惡意程式兩群體之間距離非常接近且有某些部分可能會有交集的情況，而這種情況造成 SVM 演算在在行為向量空間中無法找出一個適合的 hyper plane 來將這兩類的樣本分開，因此造成訓練 MalCatcher 偵測惡意程式的效果在某些監控行為中不是很好。

另外，我們也同時使用了 Adaboost 演算法來試著強化 NaiveBayes、REPTree、KNN 等演算法來看是否能再更進一步的提高各演算法的判斷準確度，Adaboost 中我們設定為共有 10 個回合，每個回合的數據集合採重新採樣(Re-sampling)機制，然而 Adaboost 對於演算法的優化效果沒有很好，除了在 network leakage operation 中對於 REPTree 演算法有較顯著的改善效果之外,對於其他演算法的改善有限甚至可能會較原先演算法的效果要來得差一點。

經過我們的實驗，在這些我們使用的機器學習演算法中，REPTree 演算法是比較適合運用在偵測惡意程式行為的，所以我們建議可以利用這兩種演算法來分析惡意程式行為資訊。

5.6.3 ROC Curve

ROC Curve 是一可以用來清楚明瞭地看出一個系統最後訓練出來的判斷效果的方法，它是一利用 true positive rate(y 軸)以及 false positive rate(x 軸)組成的二維線性圖，其中 ROC Curve 下方面積越大就代表著訓練出來的效果越好，這是因為若當系統的 false positive rate 越低時，系統的 ture positive rate 能夠越高的話，代表著系統的誤判率會越低，也就代表著系統的判斷效果越好。因此我們利用 ROC Curve 來觀察比較 MalCatcher 和 Crowdroid 兩者利用 Adaboost + Reptree 演算法訓練出來的效果好壞，如圖 20、圖 21。

從下面兩張圖中我們可以很明顯地看出來，MalCatcher 的 ROC Curve 下方面積比 Crowdroid 的 ROC Curve 的下方面積大了許多，並且 MalCatcher 能夠在 false positive rate 很低的情況下達到相當高的 true positive rate，由此可知我們的系統 MalCatcher 偵測惡意程式的能力比 Crowdroid 來得要好，並且判斷準確度相當地高。

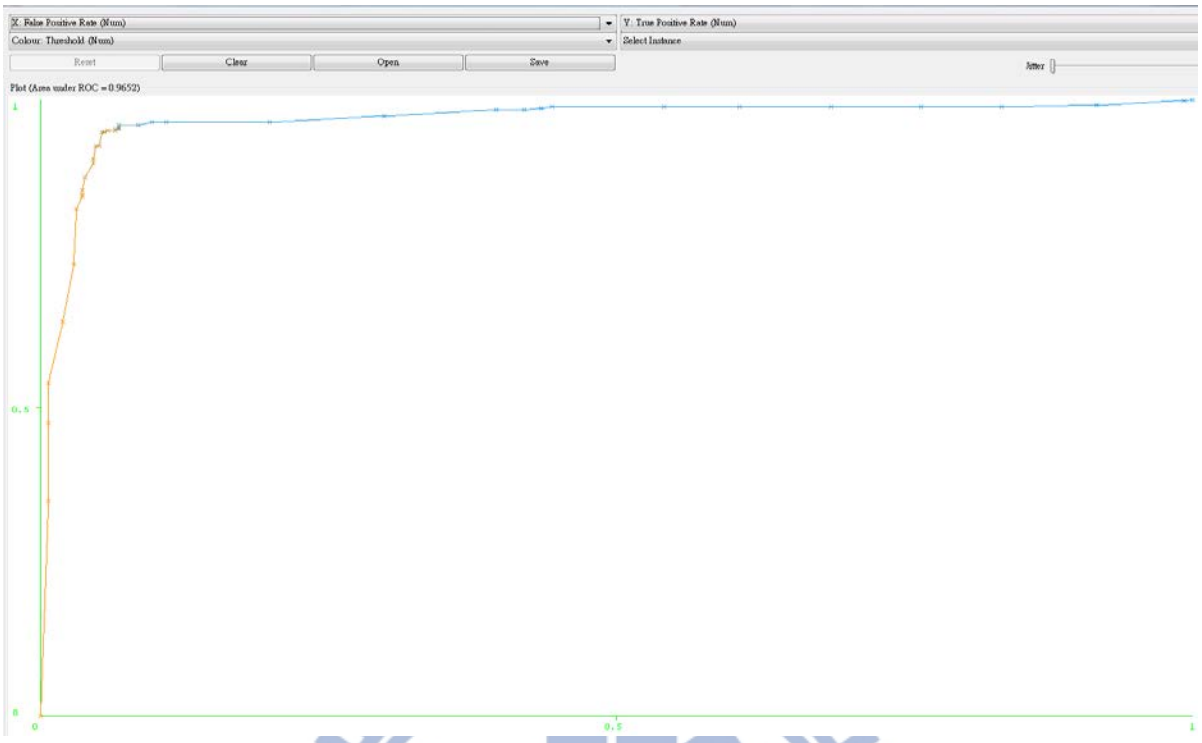


圖 20 MalCatcher 的 ROC Curve

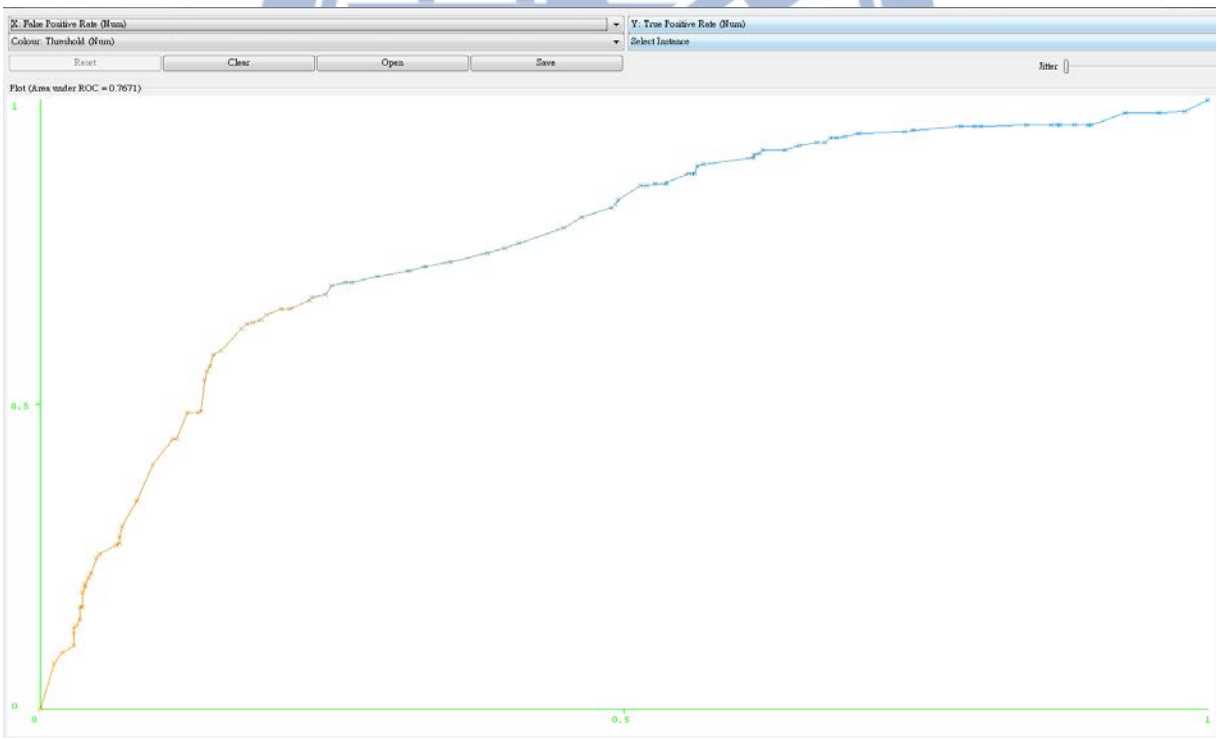


圖 21 Crowdroid 的 ROC Curve

六、 討　　論

Android 系統安全的問題目前越來越多人在研究，也有許多的研究者的研究也有了成果，可利用的研究資源也因為這些研究成果的出現變得越來越多，早期的相關研究可能由於惡意程式的發展尚未活絡而使得研究者無法取得充足的惡意程式樣本進行實驗來驗證他們的方法，只能研究者自己撰寫惡意程式或是利用研究者自己自身收集的少數的惡意程式樣本來進行實驗。多虧了 Yajin Zhou, Xuxian Jiang[12]提供了這些大量的實際的惡意程式以及陶嘉仁[13]提供了大量的正常 APP，我們才能完成這次的實驗。

從這次的實驗結果我們得知了偵測網路封包有無洩漏使用者隱私資料是偵測惡意程式行為的一個非常重要的指標，因此網路封包監控工具是否能確實的將這些隱私資料洩漏的封包偵測出來就變得很重要，snort 是一個很強大的網路封包監控軟體，能不能發揮它最大的效能、功用主要依賴於監控者對於監控的規則訂定的好壞，目前我們對於 snort 的監控規則訂定只要是參考陳彥宇[19]的研究中的規則訂定，也就是以明文以及 Base64 編碼的使用者隱私資料為關鍵字來訂定偵測規則，其他編碼型態的封包內容可能就會有監控遺漏的可能，因此在訂定規則方面可能還需要再加強，若能訂定出非常有效的偵測規則相信能監控到完整的網路封包使用者隱私資料洩漏行為。另外，加密過的網路封包在我們方法中無法進行監控，目前還沒有找到方法可以幫助我們來解析加封包來監控其中有無使用者隱私資料，因此若惡意程式將這些使用者隱私資料利用加密封包傳出我們的系統是無法偵測出來的，這一部分的問題可能會在我們未來的研究中嘗試去解決。

七、結　　語

我們提出了一個新的惡意程式行為偵測的方法，主要是透過修改 Android 系統原始碼，加入記錄行為資訊的函式來監控 APP 執行期間存取使用者隱私資料的行為以及透過將 snort 交叉編譯為 Android 系統可執行之版本來監控網路封包中有無洩漏隱私資料的行為來判斷是否為惡意程式，我們也實作出了一套名為 MalCatcher 的系統來驗證我們的方法。我們取得了共 1260 個實際真正的惡意程式[12]以及 825 個正常 APP[13]來進行我們的實驗，實驗結果也顯示我們的方法能夠有效地偵測出惡意程式，判斷的準確度甚至能達到 9 成 3，這樣的結果也大大超出了我們的預期。同時從實驗中我們也可以看出運用不同的機器學習演算法來訓練我們的系統偵測惡意程式會大大地影響其判斷的準確度，因此在選擇機器學習演算法時必須小心謹慎地選擇。另外，因為在先前的相關研究中幾乎沒有利用如此大量實際的惡意程式來測試其方法的效果，在我們的研究中透過這些大量實際的惡意程式樣本，我們得到的結果真正地反映出了我們的方法實際偵測惡意程式的效果。

八、 參考文獻

[1] Gartner Says Worldwide Mobile Phone Sales Declined 1.7 Percent in 2012. http://www.gartner.com/newsroom/id/2335616

[2] Google Play. http://en.wikipedia.org/wiki/Google_Play

[3] Apple App Store. http://en.wikipedia.org/wiki/App_Store_(iOS)

[4] Juniper Networks Inc. Malicious mobile threats report 2010/2011. Technical report, Juniper Networks, Inc., 2011.

[5] F-Secure, Mobile Threat Report, Q4 2012 http://www.f-secure.com/static/doc/labs_global/Research/Mobile%20Threat%20Report%20Q4%202012.pdf

[6] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, Yael Weiss, "Andromaly: a behavioral malware detection framework for android devices", Journal of Intelligent Information Systems, Volume 38, Issue 1, p 161-190, 2012

[7] Min Zhao, Fangbin Ge, Tao Zhang, Zhijian Yuan, "AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android", ICICA, p 158-166, Qinhuangdao, China, 2011

[8] Takamasa Isohara, Keisuke Takemori, Ayumu Kubota, "Kernel-based Behavior Analysis for Android Malware Detection", CIS, p 1011- 1015, 2011

[9] Takamasa Isohara, Keisuke Takemori, Ayumu Kubota, "Kernel-based Behavior Analysis for Android Malware Detection", CIS, p 1011- 1015, 2011

[10] Gianluca Dini, Fabio Martinelli, Andrea Saracino, Daniele Sgandurra, "MADAM: a Multi-Level Anomaly Detector for Android Malware", MMM-ACNS, p 240-253, St. Petersburg, Russia, 2012

[11] Iker Burguera, Urko Zurutuza, Simin Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for Android”, SPSM, p 15-26, New York, USA, 2011

[12] Yajin Zhou, Xuxian Jiang, "Dissecting Android Malware: Characterization and Evolution", IEEE Symposium on Security and Privacy(S&P), p 95-109, San Francisco, CA, 2012

[13] 陶嘉仁, “Android 程式權限分析”, 碩士論文, NCTU, 台灣, 2012

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, David Wagner, “Android Permissions Demystified”, CCS, p 627-638, New York, USA, 2011

[15] Snort, http://www.snort.org/

[16] Snort wiki introduction, http://zh.wikipedia.org/wiki/Snort

[17] Weka, http://www.cs.waikato.ac.nz/ml/weka/

[18] Weka wiki introduction, http://en.wikipedia.org/wiki/Weka_(machine_learning)

[19] 陳彥宇, “Android 上的隱私資料洩漏偵測”, 碩士論文, NCTU, 台灣, 2012

[20] Linux command “strace” wiki introduction, http://en.wikipedia.org/wiki/Strace

[21] Ubuntu official website, http://www.ubuntu-tw.org/modules/tinyd0/

[22] Android official website, http://developer.android.com/sdk/index.html