國立交通大學

資訊科學系

碩 士 論 文

Android 上的隱私資料洩漏偵測

Detection of Leakage of Private Data on Android

研 究 生：陳彥宇

指導教授：曾文貴　教授

中 華 民 國 一 百 零 一 年 九 月

Android 上的隱私資料洩漏偵測

學生：陳彥宇　　　　　　　　　　　　　　　　指導教授：曾文貴

國立交通大學資訊科學與工程研究所碩士班

摘　　要

近年來智慧型手機、平板電腦科技蓬勃發展，擁有強大的功能，也讓這些移動裝置成為新的攻擊目標。其中最常見的攻擊是資料竊取與外流。本文提出 LeakDet，一個可安裝在 Android 行動裝置上，可偵測隱私資料洩漏的系統。LeakDet 能夠偵測封包內是否含有隱私資料，並且阻擋該封包流向的 IP。我們將最新版的入侵偵測系統(IDS)軟體 Snort 移植到 Android 手機上，並搭配 Snortsam 以及 Linux Kernel 內建的防火牆 Iptables 來達到入侵防禦系統(IPS)的功能。我們設計了一個 Android 應用程式名為 PrivacyGuardian。PrivacyGuardian 提供了操作介面，讓使用者能操作 IPS 以及自定義個人隱私資料。實驗部分，我們模擬實際的殭屍網路攻擊情境來竊取資料，證明了 LeakDet 能成功偵測攻擊封包並且阻擋該攻擊 IP。同時也對 LeakDet 在手機上的資源消耗量做了測試。

Detection of Private Data Leakage on Android

student：Yen-Yu Chen Advisor：Dr. Wen-Guey Tzeng

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

In recent years, mobile devices such as smartphone and tablet have become the target of attacks because of their powerful functions. To defense the data stealing attack, we propose LeakDet. LeakDet is a system that installed on a mobile device for detecting the leakage of private data. LeakDet can check if the packet contains private data and block the destination IP of that packet. We port the newest version of Snort, an Intrusion Detect System(IDS), on Android. We integrate Snort, Snortsam and the build-in firewall in Linux kernel, Iptables, to implement an intrusion prevention system (IPS). We also design an Android application called PrivacyGuardian for user to control the IPS and define his own private data. Last, we take an experiment to test the functionality of PrivacyGuardian by simulating the botnet attack environment. The experimental result shows that LeakDet can detect the packet which leaks private data and prevent the following attack by blocking the attacker's IP

誌　　　　謝

首先我要感謝曾文貴老師在碩士班這兩年中的指導，不論是在密碼學的領域或是做事方法與態度上，老師平日的言教身教皆令我獲益良多。還要感謝我的論文口試委員謝續平老師和黃育綸老師，給我許多改進論文的建議，讓我的研究更完整。謝謝實驗室的毅睿學長在平時研究、報告上還有其他許多方面給我的幫助。謝謝實驗室的宣佐學長和孝盈學姊在我的研究遇到問題時不吝與我討論。感謝實驗室的學弟妹在計畫方面的配合。感謝同學們在學習上互相討論，在實驗室事務上的互相幫助。在最後要感謝我的家人，在求學的兩年期間給我信賴與支持，讓我無後顧之憂地完成碩士班的學業。還要感謝我的貓 Luna 總是能給我精神上的鼓勵。最後，感謝所有曾經給過我幫助的人，有了大家的幫忙這篇論文才得以誕生。

Index

摘要 i

Abstract ii

誌謝 iii

Index iv

List of Figures vi

List of Tables vii

1 Introduction .... - 1 -

1.1 Android .... - 3 -

1.1.1 Android Architecture .... - 4 -

1.1.2 Application Components .... - 6 -

1.2 Android Security .... - 7 -

1.3 Intrusion detection system/Intrusion prevention system .... - 9 -

1.4 Snort .... - 10 -

1.5 Snortsam .... - 12 -

2 Related Works .... - 14 -

3 Background IPS .... - 18 -

3.1 System overview .... - 18 -

3.2 The architecture of the IPS .... - 19 -

4 Application Design .... - 21 -

4.1 Requirement Analysis .... - 21 -

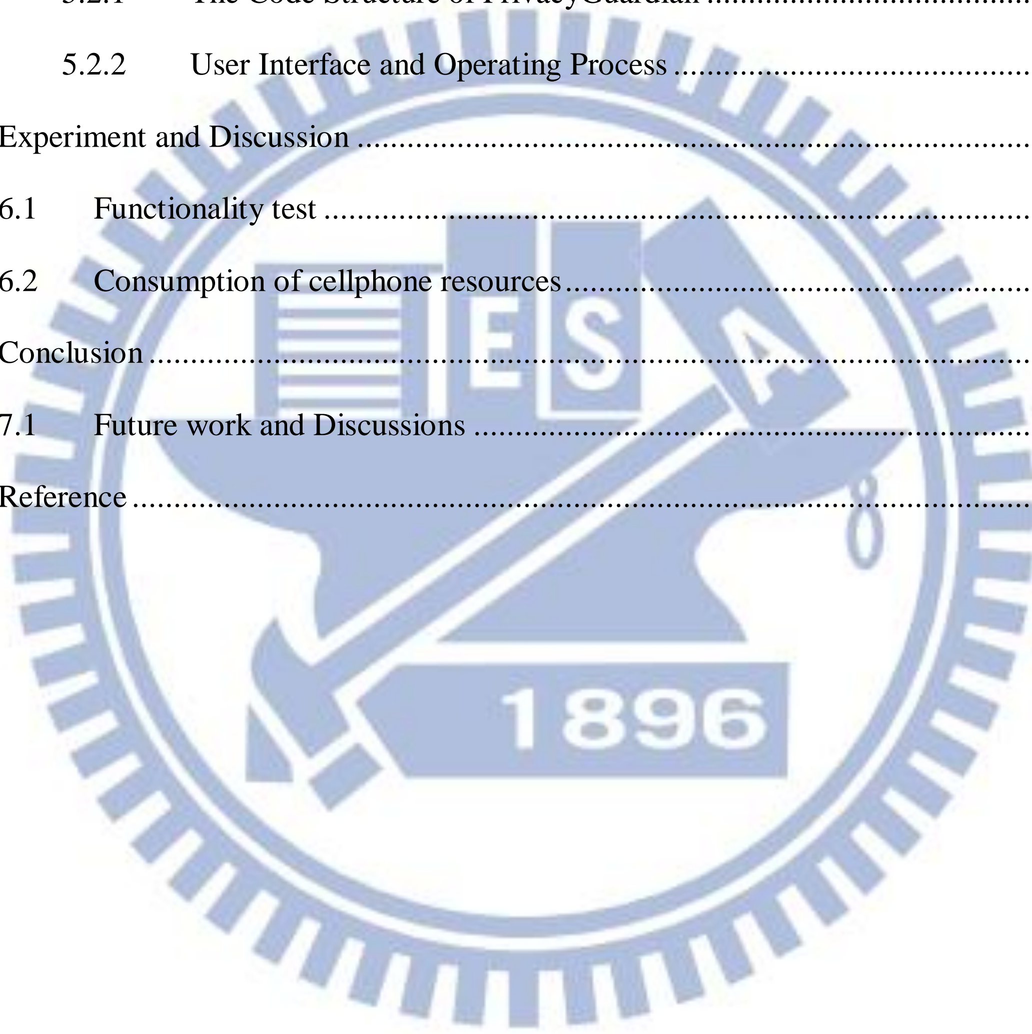
4.2 Generating of Snort Rules .... - 22 -

4.2.1 Process of Rule generating .... - 23 -

5 Implementation .... - 28 -

1896

5.1 Build the IPS on Android ....- 28 -
5.1.1 Cross-compile Snort on Android ....- 29 -
5.1.2 Cross-compile Snortsam on Android ....- 33 -
5.1.3 Run Snort and Snortsam on Android ....- 34 -
5.2 Implementation of PrivacyGuardian....- 34 -
5.2.1 The Code Structure of PrivacyGuardian ....- 34 -
5.2.2 User Interface and Operating Process ....- 38 -
6 Experiment and Discussion ....- 42 -
6.1 Functionality test ....- 42 -
6.2 Consumption of cellphone resources....- 44 -
7 Conclusion ....- 48 -
7.1 Future work and Discussions ....- 49 -
8 Reference ....- 51 -



List of Figures

Figure 1: Share of smartphone operating system in 2012 June ....- 4 -

Figure 2: Architecture diagram of Android Source: Android ....- 5 -

Figure 3: Structure of Snort rule ....- 11 -

Figure 4: Snort detection diagram ....- 11 -

Figure 5: Corporation of Snort, Snortsam and Iptables.....................................- 13 -

Figure 6: System overview ....- 18 -

Figure 7: The flowchart of the IPS ....- 19 -

Figure 8: The information flow through the IPS ....- 20 -

Figure 9: The requirements from the viewpoint of user ....- 22 -

Figure 10: The process of generating Snort rules..- 23 -

Figure 11: Structure of Classes in PrivacyGuardian ....- 35 -

Figure 12: The main activity of PrivacyGuardian...- 39 -

Figure 13: PrivacyGuardian informs user for activating Snort ....- 39 -

Figure 14: PrivacyGuardian informing user for stopping Snort ....- 40 -

.Figure 15: The report for the result of detection ....- 40 -

Figure 16: The interface for selecting private data...- 40 -

Figure 17: The interface of the file manager...- 41 -

Figure 18: The warning for selecting a file ....- 41 -

Figure 19: The trigger of botnet attack ....- 43 -

Figure 20: The victim connects to the botmaster ....- 43 -

Figure 21: The command of stealing data from the botmaster ....- 43 -

Figure 22: The firewall block the attacker's address ....- 44 -

List of Tables

Table 1: Matching of the data type and the extracted keyword ....- 25 -

Table 2: Recommend rule options for each data type ....- 25 -

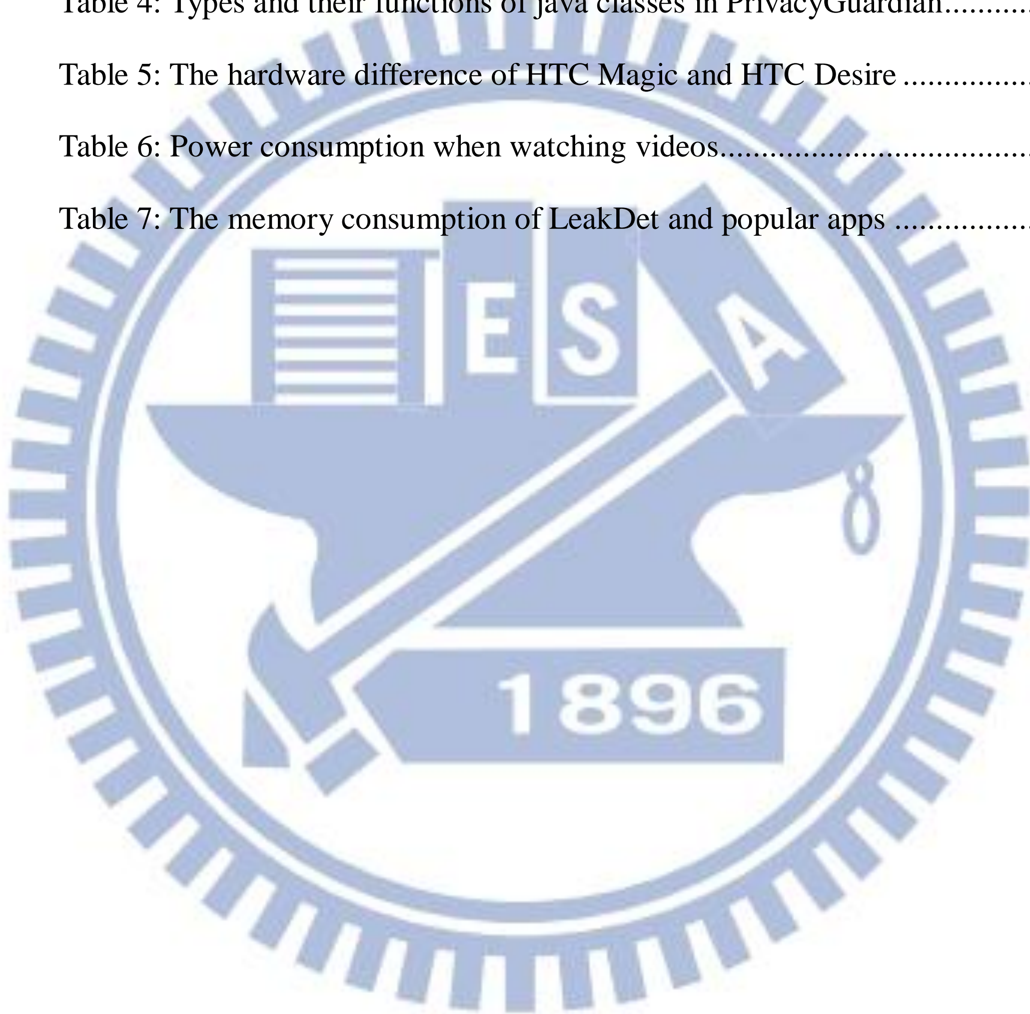
Table 3: The version of software and hardware we used...................................- 29 -

Table 4: Types and their functions of java classes in PrivacyGuardian...............- 35 -

Table 5: The hardware difference of HTC Magic and HTC Desire ....- 45 -

Table 6: Power consumption when watching videos..- 45 -

Table 7: The memory consumption of LeakDet and popular apps ....- 46 -



1 Introduction

As the smartphone becomes more and more popular, and its functionality grows more powerful, there are more and more people who choose smartphones to be the tool to handle their affairs of office or daily business. Especially on the condition of moving frequently between different places for work, smartphone is necessary. On the condition above, smartphone users usually store some of the important business data or personal private data in the device. Those private data will become the target of hackers.

The cause of data leaking can be divided into two kinds according to the motivation. For the first kind, the user inadvertently leaks the private data. For example, he synchronized the privacy data on a public computer so that the data could be access by any stranger. For the second kind, data is stolen by the attackers. The inadvertent leakage can be prevented by user's awareness of protecting the privacy. But it's more difficult to defense a real attack carried out by the hackers.

The measures of hacker's attack can be various. Hackers designed malicious software (malware) to steal the private data stored in the mobile devices. These malwares usually pre-tend that they were normal tool or game software in the software market by showing the interesting information of the malware to attract the user's attention and get their trust before being installed. Once the user install a malware, the malware can do everything it declared in its manifest file. For example, make the phone calls, access the private data and send them out through the internet or some covert channel. Hackers can also attacks the system vulnerabilities directly via malwares or the internet to get the supreme authority in the system, thus he can control the whole system.

In general, there are many ways for attackers to transfer the private data from the cellphone to themselves. The most common way is transferring data through the internet. This

may be the most convenient way because almost all of smartphones are able to connect to the internet. An attacker can easily send out private data with a malicious app or program. On PC, the data transferred through the internet might be filtered or blocked by firewalls or detection systems, but there is lack of such kind of protection. Except the internet, the data may be stolen by the program installed in the PC while the phone is synchronizing with the PC. Attacker can also use covert channel like manipulating volume or screen of the cellphone with an application, and receive these digitalized data with another application. With covert channel, attacker can bypass many security mechanisms such as permission checking and the isolation between applications. In this research, we only focus on detecting and preventing the leakage transferred through the internet.

Try to solve the data leaking problem, we want to detect and block the private data leakage. In this paper, we proposed a data leakage detection and prevention system on the Android operating system, called (LeakDet). This system combined the intrusion prevention system (IPS) and an Android application we designed to help user to control the IPS. We install Snort in the Android phone as the packet filter of the IPS, and use Snortsam to communicate between Snort and the Iptables – the firewall we use. So once an outgoing packet with private data in it is detected, LeakDet can block the destination IP of the packet after detecting the private data leakage. We block the IP address which belongs to the attacker to prevent the follow-up attacks or data stealing, and meanwhile reduce the overhead of Snort.

For the rest of this chapter, we'll introduce the Android system and its security mechanisms. Then we'll give a glimpse of the intrusion detection system and the software we used in LeakDet: Snort, Snortsam and Iptables.

1.1 Android

Android is a Linux-based operating system primarily designed for mobile devices such as smartphones and tablet computers utilizing ARM processors. It is the most popular open source operating system for the mobile devices now. Figure 1 shows the share of smartphone operating system in 2012 Q2 in the America. According to the report from Nielsen, during 2012 June, 51.8 percent of Smartphone owners had a handset that runs on Android operating system. 34.3 percent of smartphone owners use an Apple iPhone, and Blackberry owners rep-resented another 8.1 percent of the smartphone market.

Android has a large community of developers writing applications (apps) that extend the functionality of the devices. Hence, there are lots of programming resources and discussions on the internet that can be referenced. Apps can be downloaded from third-party sites or through online stores such as Google Play (formerly Android Market), the app store run by Google.

We decide to develop our LeakDet on Android platform because of its popularity in the world, rich of the resource of development community and also because that it is open source. The most important, there has been some security problem in Android, such as the loose process of putting an app on the market and the access control of sensitive resources which relied on user’s awareness of security. These features of Android make attackers easier to performing a success attack on Android platform than on Apple platform. In such a mobile platform, an effective defense mechanism is needed. Next we will introduce the architecture of Android and the components of application.

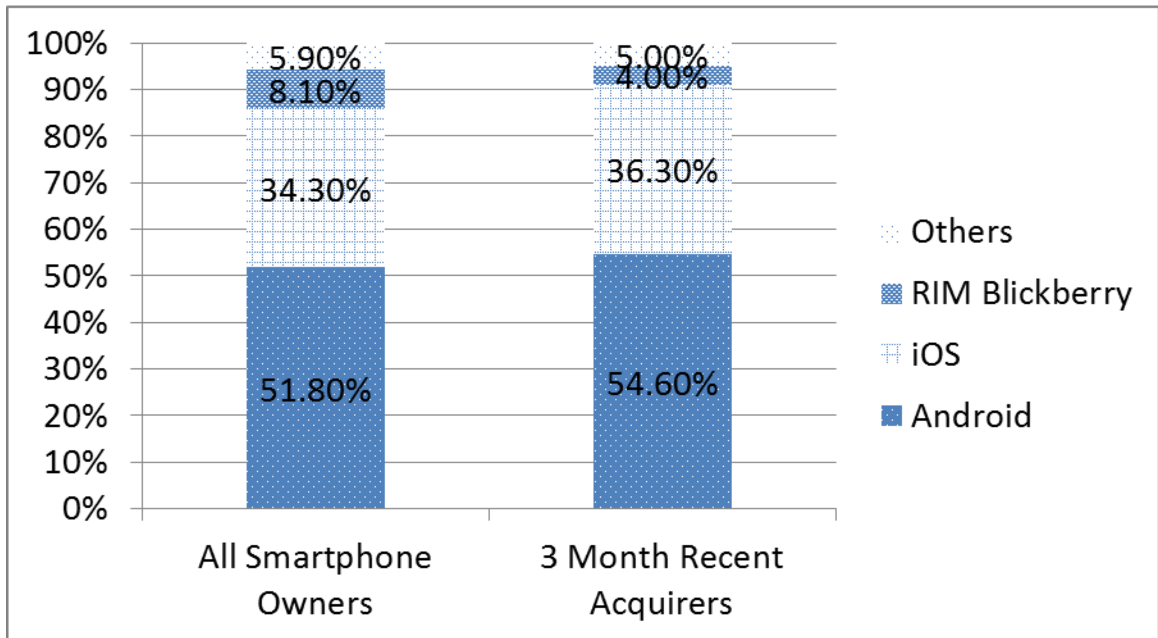


Figure 1: Share of smartphone operating system in 2012 June

Source: Nielsen Wire [9]

1.1.1 Android Architecture

The architecture of Android operating system is consisted of four parts, shown in Figure 2. We will introduce each layer separately this section.

The under layer in Android is Linux kernel, Android use and only use the Linux kernel of version 2.6.x. Like the operating system in PC, the kernel handles task management, file management, memory management and so on. Android also provide driver module of hardware as hardware abstraction layer to make the independence of Android framework and Linux kernel.

The second layer is middleware, the bridge of communication between operating system and Android application. Middleware layer consists of two parts: libraries and virtual machine. Libraries part provides the functionality of the JAVA programming language. For example, Android uses OpenCORE as its base media framework and use SQLite as its database system. The libraries are usually written in C. In virtual machine layer, Android application runs in its

own process, with its own instance of the Dalvik virtual machine (DVM). DVM is a register based VM, and is optimized for low memory requirements. DVM doesn't execute JAVA bytecode but files in the Dalvik executable (.dex) format.

In the application framework layer, Android provides its application framework APIs for programmers to develop applications on Android. The last layer, application layer, represents Android applications that can be installed and executed in the devices.

Except only implement our IPS on Android, we also designed an application as the frontend of our LeakDet for user to manipulate the IPS easily. We are going to introduce the components of Android application in the next section.

Figure 2: Architecture diagram of Android

Source: Android – Wikipedia [2]

1.1.2 Application Components

An Android application consists of one or more of the following classifications of components. The four class of components are: activity, service, broadcast receiver and content provider.

Activity is the most common component for developers and users in Android. An activity represents a single screen with a user interface. For example, a contacts application might have one activity that shows a list of contacts, another activity to edit the contact list. Although there might be any number of activities in an application, each activity is independent of the others. Different application can start any one of these activities (if the contact application allows it).

Service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service doesn't provide a user interface. For example, a service might play music in the background while the user is in a different application, or it might download data over the network without blocking user interaction with an activity. An-other component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

Content provider manages shared application data. You can store the data in the file sys-tem, a SQLite database, on the web, or any other persistent storage location your application can access. Through the content provider, other applications can query or even modify the da-ta (if the content provider allows it). For example, the Android system provides a content provider that manages the user's contact information. As such, any application with the proper permissions can query part of the content provider (such as ContactsContract.Data) to read and write information about a particular person.

Broadcast receiver is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or the booting was completed. Although broad-cast

receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. A broadcast receiver is a “gateway” to other components and is intended to do a very minimal amount of work itself.

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an intent. Intents are the messengers that request an action from other component, whether the two components belongs to the same application or another.

After introducing Android system, we are going to discuss the mechanism of Android’s security in the next section.

1.2 Android Security

Android has a unique security model, which focuses on putting the user in control of the device. Three main security mechanisms play an important role in security model of Android system. They are traditional access control, isolation and permission-based access control.

In traditional access control, Android provides the security lock. The security lock requires user to draw a predetermined graph on the screen to unlock the screen, instead of enter the traditional alphabetical password. If too many fail login attempt occurred, the device will lock its screen permanently. User also needs to enter the PIN number of the SIM card right after he boots the cellphone. He must enter the password correctly or the SIM card will be locked after a certain times of fail attempt. The password system is enough to protect the device from casual attacks. However, Android didn’t encrypt the data stored in the SD card, also didn’t provide any option of data encryption. If the attacker can access the Android mobile device physically, he can simply eject the SD card from the device and obtain all the data stored in it. This way can bypass all password control enabled on the device.

Android employed an isolation system to ensure that apps only access the approved sys-tem resources. Each Android app runs within its own virtual machine, and each virtual

ma-chine is isolated in its own Linux process. The isolation system ensure that any application installed in Android system cannot access or modify the resource belong to another application. This mechanism also prevents applications from accessing or modifying the operating system kernel, ensuring that a malicious app can't get administrator-level control over a device. The isolation system has some exceptions: 1). Apps may obtain the list of apps installed on the device. 2). Apps may read the contents of the user's SD card, which typically holds the user's media files, installed programs, and possibly documents. Apps may read all of the data on the SD card regardless of which app created a particular piece of data, all apps can read that data. 3). Apps may launch other applications on the system. With such a strict isolation system, an application can still access system resources by declaring a set of permissions. We will discuss the permission-based access control system next section below.

Android predefined a series of permissions corresponding to system resources that needed to be supervisory controlled. By default, most Android applications can do very little without explicitly requesting permission. Before installing an app, the list of system resources that the app require to access will be present to user in non-technique language, and the user can decide whether or not to allow the app to be installed based on their tolerance for risk. Note that user cannot grant part of the permissions the app required; he can only choose to accept all or reject all. Once the installation is allowed, the app is granted permission to access all of the requested subsystems.

The permissions-based access control system seems to be robust, but the system actually let the user take all the responsibility of making the decisions that the set of permissions re-quested by the app is safe or not. If a user accidently allows a malicious application to run on his mobile device, the malware can do anything it request. For example, consider a malware which requests permission to access the phone-state (including IMEI, IMSI, SIM card number …) and also to access the internet. In a legitimate scenario, an app might use the device's unique ID to identify the user and provide customized content, so this malware might

pretend that it was a harmless and useful tool. Since the permissions of accessing the phone-state and internet are common, it’s hard for user to tell whether this combination of permissions is dangerous or net. Once the user decides to install this malware, the private information of the cellphone will be read and leaked through the internet.

The permissions-based access control system is powerful, but it relies on the user to make important security decisions. Unfortunately, most users are not technically capable of making such decisions. Our LeakDet can help to reinforce the security system; defend part of attacks like data stealing.

1.3 Intrusion detection system/Intrusion prevention system

Intrusion detection system is a tool which helping us to detect possible attack pattern on a system. An IDS monitors network flows or system activities for malicious activities and report to system manager.

In a passive system, if the intrusion detection system (IDS) detects a potential attack, it logs the information or signals an alert on the console. In a reactive system, also known as an intrusion prevention system (IPS), the IPS auto-responds to the suspicious activity by resetting the connection or by controlling the firewall to block network traffic from the suspected malicious source.

There are two detection techniques of IDS: anomaly-base and signature-base. Anomaly-based IDS first determines normal network activities like what protocol or port should be use, what IPs usually connects to each other and how much bandwidth is generally used. If there is any anomalous been detected, the IDS will alert the administrator. Signature-based IDS monitors the packets and compare them with the predefined attack signatures.

IDS systems can also be categorized in three types depends on the information they monitor: Network intrusion detection system (NIDS), Host-based intrusion detection system

(HIDS) and Stack-based intrusion detection system (SIDS). An NIDS is an independent platform that identifies intrusions by examining network traffic and monitors multiple hosts. In an NIDS, sensors are located at choke points in the network to be monitored and capture all network traffic, analyze the content of individual packets for malicious traffic. An HIDS consists of an agent on a host that identifies intrusions by analyzing system calls, application logs, file-system modifications and other host activities and state. In an HIDS, sensors usually consist of a software agent. An SIDS consists of an evolution to the HIDS systems. The packets are examined as they go through the TCP/IP stack and, therefore, it is not necessary for them to work with the network interface in promiscuous mode.

The IDS we used in LeakDet is Snort. Snort is an NIDS and belongs to Signature-based IDS which matches the packets with a series of predefined snort rules. Because we focus on detecting the private data transferred through the internet, Snort must have the ability to monitors the network traffic.

1.4 Snort

Snort is a popular open-source Intrusion Detection System (IDS). It can be also Intrusion Prevention System (IPS) after doing certain configuration or with the help of other applications. Snort is capable of performing real-time traffic analysis and packet logging on networks. It has the ability to perform protocol analysis, content searching or matching, and can be used to detect a variety of attacks, such as buffer overflows, stealth port scans, CGI attacks, and much more. Snort performs detection according to the rules written on configure files.

Snort can be configured in three main modes: sniffer, packet logger, and network intrusion detection. In sniffer mode, the program will read network packets and display them on the console. In packet logger mode, the program will log packets to the disk. In intrusion detection mode, the program will monitor network traffic and analyze it against a set of rule defined by the user. The program will then perform a specific action based on what has been

identified. For real-time traffic analysis, Snort must be configured in the intrusion detection mode.

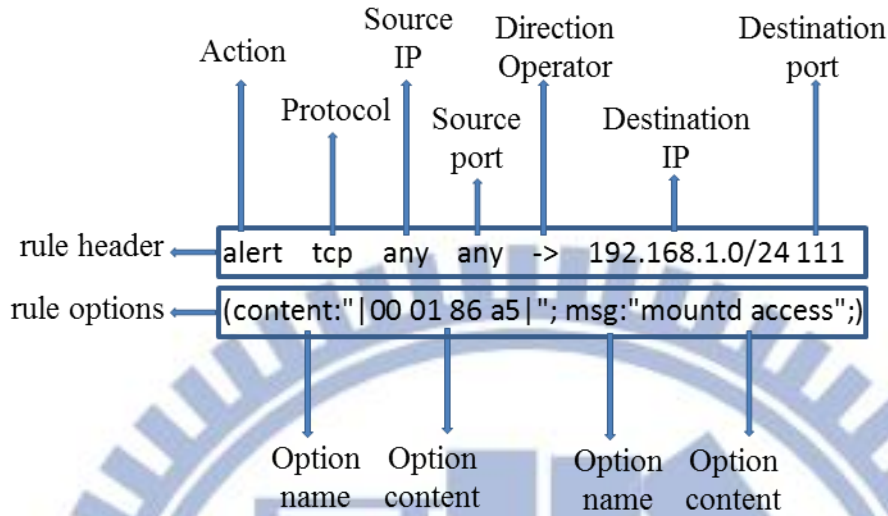


Figure 3: Structure of Snort rule

As Figure 3 shows, Snort rules are divided into two logical sections, the rule header and the rule options. The rule header contains the rule's action, protocol, source and destination IP addresses and net masks, and the source and destination port information. The rule option section contains alert messages and information in which parts of the packet should be inspected to determine if the rule action should be taken.

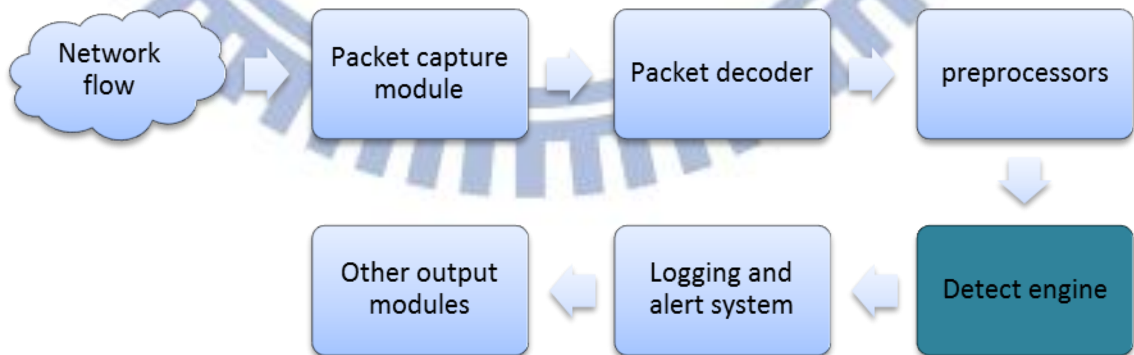


Figure 4: Snort detection diagram

Figure 4 shows the detection process of Snort. The packet capture module captures the packets flow on target interface. A packet is presented as the form of packet structure in Snort. The decoder decodes packets to packet structure. The preprocessors check or modify the packets for countering certain means that evading from or attack the IDS. Detect engine reference the rules and do pattern matching to find that if any packet matches a rule. Snort will take action like logging or making alert to the system manager if there is a match, the action depends on Snort rules.

In this paper, we use Snort as a part of the IPS to detect packets that contains part of the private data. The data is chosen by user. We extract keywords from the chosen data, and set the keywords as the content of the rules.

1.5 Snortsam

Snortsam is an agent that allows Snort to block intruding connections by reconfiguration of firewalls. To directly control the firewall software, Snortsam should be installed on the same machine of the firewall and run with the authority of administrator. To combine Snort and Snortsam, first the Snort has to be patched with the output module provided by Snortsam. Then configure the "Snort.conf" with an "output plugin" option followed the IP address of Snortsam.

For each of the Snort rule that request for a blocking action are extended with the option "fwsam". When a rule triggers a block, the Snort sensor sends an encrypted TCP packet to one or more Snortsam agents that are running on the firewalls. The agent performs certain checks, and if allowed, the agent will request the firewall to block the reported IP address. Figure 5 shows the cooperation of Snort and Snortsam.

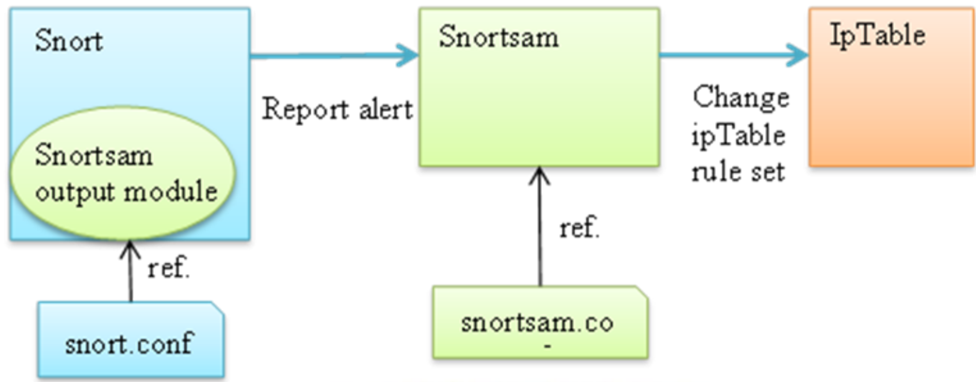


Figure 5: Corporation of Snort, Snortsam and Iptables

2 Related Works

To avoid the leakage of private data, several concepts of solutions have been proposed by researchers, which can be roughly categorized to a few types. The first type of solutions is identifying the malware before the user install it, this way include static and dynamic analysis of applications. The second type is modifying the operating system, make the OS have the ability to detect and track the leakage of data. The attack will be traced or stopped once they're activated. The third type is to partially constrain the functionality of an application or disable the dangerous permission of an application. This may be achieved by modify the Android system or by manipulating the apk file of target application. The rest part of the application can still function normally. This kind of solutions are usually achieved by modify the Android system.

For malware detection of Android applications, the concept is to observe the application into detail and then analyze the data collected, and then distinguish that if the application is malware with the result of the analysis, like comparing the data to the threshold or observing if there are the specific features exist in the collected data. There are two common approaches in application analysis: static analysis and dynamic analysis. The difference of the two approaches is depends on whether the application is executed or not during the testing. If the analyzed data is collected in the runtime of application, it belongs to dynamic analysis. Otherwise, the method belongs to static analysis.

For statically analysis the applications, [11] provides an framework for analyzing the Dalvik bytecode, and detecting the privacy-violating information flow. The authors analyzed the Android API and identified a meaningful set of private information sources and sinks Android Applications.

[13] provides a static analysis tool called AndroidLeaks which analysis the Dalvik

bytecode of applications for finding potential privacy leaks in Android applications. The analysis can be divided into two steps: the permission miner and taint analysis. First the permission miner checks if the application contains a combination of permissions that could leak private data. Then AndroidLeaks performs taint analysis to determine if information from a source of private data may reach a network sink. The taint analysis is done by the help of WALA, a program analysis framework for Java source and byte code.

[12] propose an Android Application Sandbox which is able to perform both static and dynamic analysis on Android programs to automatically detect suspicious applications. Static analysis scans the software for malicious patterns without installing it. Dynamic analysis executes the application in a fully isolated environment, like sandbox, which intervenes and logs low-level interactions with the system for further analysis.

Applying the application-analysis approach to identify the malicious application may be efficient and is benefit from almost unlimited computing resource compared to mobile device. Due the security mechanism of Android, it's possible that the user install the malicious application from third market even the application has been identified as a malware. Once the malware has been installed, it can still leak the private data if only it has the permissions. So we also need a real time detect system to detect data leakage.

[20] propose a system-wide dynamic taint tracking and analysis system. This system tracks the flow of privacy sensitive data via labeling (tainting) these data from privacy-sensitive sources and transitively applies labels as sensitive data propagates. As the tainted data are transmitted over the network, the system will log the data, the application which sent the data, and the destination IP. TaintDroid is efficient but it cannot protect the system from attack but only records the data and inform the user.

In [21], the authors propose a privacy model in smartphone. They design and implement an Android privacy-mode. The privacy-mode provide user the fine-grained control and runtime re-adjustment capability to specify what kinds of user information can be accessible to

untrusted apps.

The leak detection systems like [20] and [21] are able to detect the leakage of private data, but they are not easy to user to install, and can block attack but only detect it.

[3] is also an implementation for control the permission of applications installed on the cellphone. It doesn't need to be root or make any change to Android system. The method is easy. First, Appshield obtains the manifest file packaged in the apk file of target application. Then it delete the Android permissions which selected by the user. Last, repackage the modified manifest with the original apk and install it on the cellphone. If the target application wants to execute the function that requires the deleted permissions, it won't pass the permission check. This might cause the crash of target application. Appshield can really prevent malicious applications to leak the private data on mobile device simply by eliminate its permission. But the application may not function normally while permissions, which should be declared, are disabled.

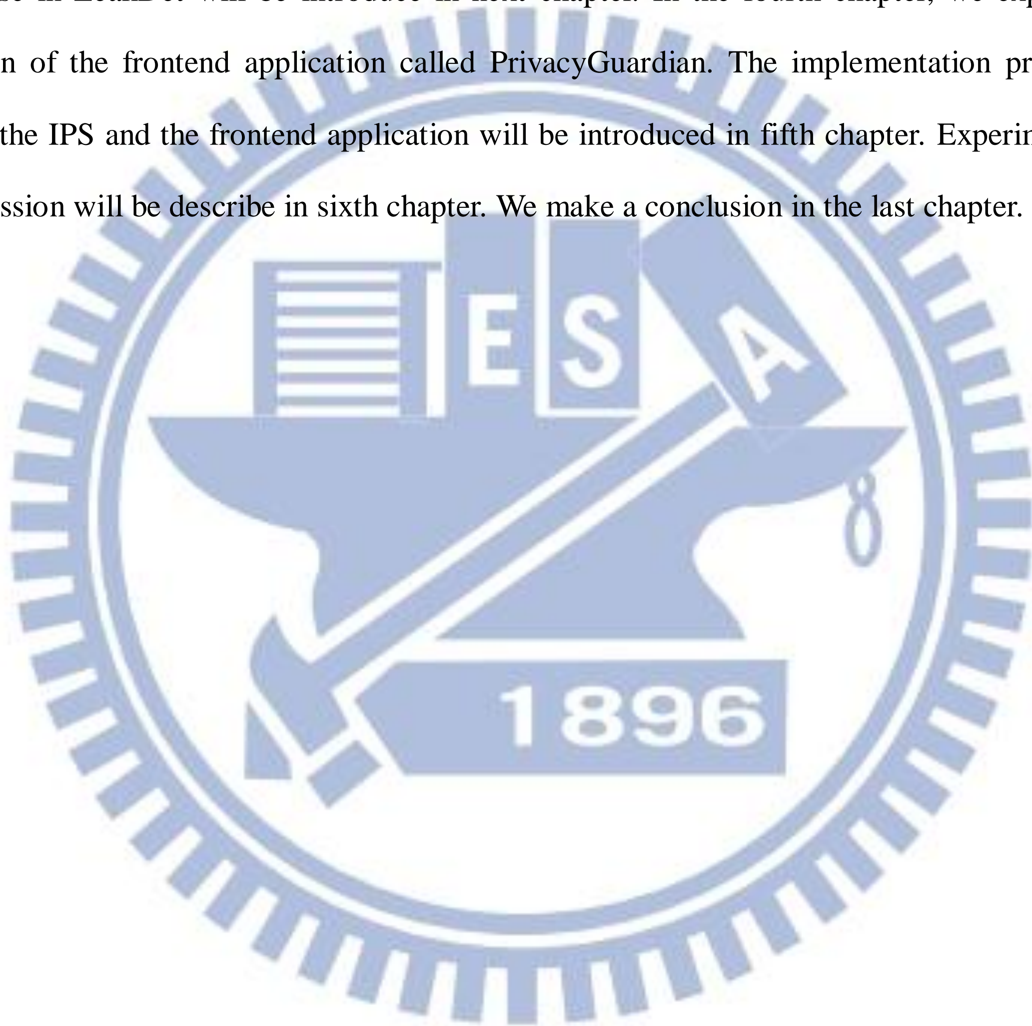
Now there are many applications or actual practices using traffic monitor or filter software to protect mobile devices from attack or malicious connection. [15] use Snort to capture the network flow and analysis them for finding certain attack patterns. They focus on tracing information (IMEI, phone number, credit card information) leakage in HTTP and identifying connection attempts to command-and-control servers. The problem of [15] is like all other detecting systems using only Snort: they can only detect but can't defense those attacks.

For now, to the best of our knowledge, there is no application or practical implementation which uses traffic monitor software like Snort or firewall to protect cellphone from private data leakage. We also find that there is seldom a system which takes defend operation after the leakage of data has been discovered. Furthermore, most of the detection mechanisms require the modification in operating system level. It's not easy for general user to apply on his own mobile device.

In this paper, we design a tool called LeakDet to detect the leakage of private data, and

protect cellphone from leakage. We not only monitor and detect the data leakage, but implement the defense method by applying an IPS to block the following attack triggered by attacker after the first data stealing. We combine Snort and firewall to implement this IPS and port it on Android platform to prevent data leakage.

For the rest of the paper, we discuss the design and implementation of LeakDet. The IPS we use in LeakDet will be introduce in next chapter. In the fourth chapter, we explain the design of the frontend application called PrivacyGuardian. The implementation process of both the IPS and the frontend application will be introduced in fifth chapter. Experiment and discussion will be describe in sixth chapter. We make a conclusion in the last chapter.



3 Background IPS

3.1 System overview

To solve the problem of data leakage on an Android smartphone, we designed LeakDet. LeakDet is composed of two parts: the IPS and the android application. We apply the IPS to detect the leakage of private data. The IPS is composed of Snort, Snortsam and Iptables. Snort capture outgoing packets and match the content and features of the packets with the predefined signatures (snort rules). If a packet matches the signature, it will be dropped and Snortsam will execute Iptables to add the rules to block the destination IP address of the packet afterwards. We also designed an application called PrivacyGuardian for user to operate the IPS easily, including activating the IPS, stopping the IPS or defining the private data to generate corresponding signature in the IPS. Figure 6 shows the system overview, user execute the IPS and define his private data through the interface provided by PrivacyGuardian. The application also accesses the private data, which was defined by user, and generate signature for the IPS to detect the privacy leakage. The IPS detects data leakages and blocks the suspicious IP, and report the detecting result to the user. In the following sections, we'll discuss the architecture of the IPS.

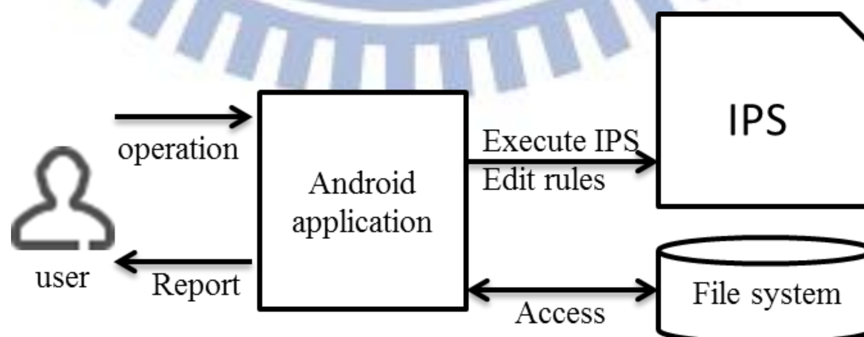


Figure 6: System overview

3.2 The architecture of the IPS

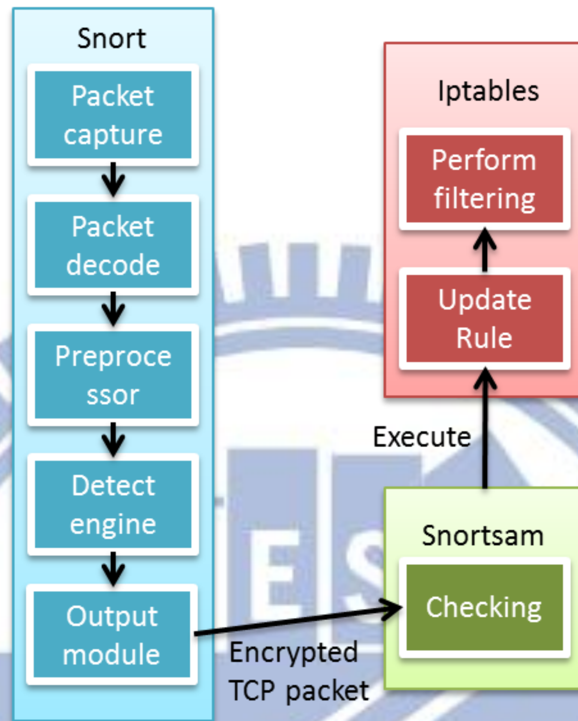


Figure 7: The flowchart of the IPS

The IPS consists of Snort, Snortsam and Iptables. The three components are all installed on the smartphone to implement the IPS. Snort is originally an intrusion detection system (IDS) software, and is used as a packet filter in our IPS. Snortsam is an agent that receives the request from Snort, and executes Iptables to add the firewall rules. Iptables is a build-in firewall in Linux kernel. Iptables can block packets with a set of predefined rules. Figure 7 show the flow diagram of packet filtering in the IPS. After capturing and preprocessing the packets, Snort tries to match them with the Snort rules. If the packets contain certain parts of the private data, Snort will take corresponding reactions, like “make alert”,” record this packet” or “drop this packet”, depends on how Snort has been configured. In our case, the packet will be recorded, and then Snortsam will receive messages sent from Snort. The massage including

the information of the packet, which IP should be banned and how long the IP will be banned. Snortsam executes Iptables for adding rules to block the packet's destination IP. If the attacker activated another attack through the internet, the attack will be blocked by the firewall. Figure 8 shows the information flow through the IPS. Once the packet which contains the private data is detected, the IP of that packet will be blocked. Therefore, the following data transfer will be blocked and the attacker won't active any furthur attacks to the protected cellphone.

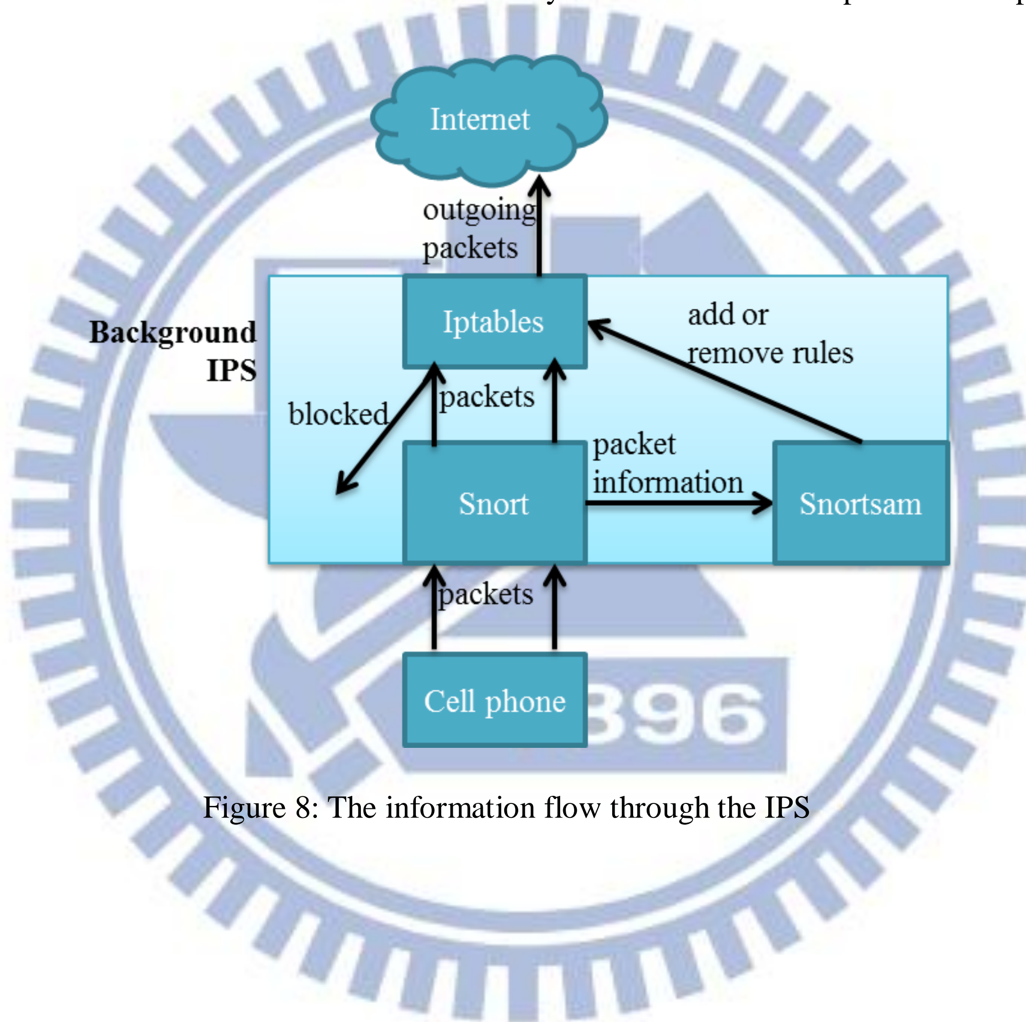


Figure 8: The information flow through the IPS

4 Application Design

We design an Android application called PrivacyGuardian as frontend for helping user to manipulate the IPS. In this chapter, we discuss how to design the application. First we start at analyzing the user's need and necessary functions of PrivacyGuardian to operate the IPS. Then we set up the Structure of PrivacyGuardian. The process of generating Snort rules and the components of this process will be described in detail. In the end of next chapter, we will introduce the user interface and the operational process of PrivacyGuardian by showing the screenshots.

4.1 Requirement Analysis

The design begins with the analysis of requirement. Because PrivacyGuardian is an Android application that helping user to operate the IPS, it must be able to activate the IPS, and access the file system to modify the configuration file that referenced by Snort and Snortsam.

We conclude some functionality that our application should have: 1). Being able to execute the Linux shell command to activate or stop Snort and Snortsam in our device since the two executable files are the main part of our IPS. Also provide a control panel for user to decide when to start the IPS. 2). Provide an interface for user to define his private data. The interface includes a file manager for user to select his files like photos or personal documents as the private data, and also a list of private system resources, like user's contacts, location information and so on. 3). Provide a process to extract keyword from the private data chosen by user, generate Snort rules according to the keywords and the type of private data, and write the Snort rules to the configuration file of Snort in the file system. 4).Provide an interface to report the result of detection to user. Figure 9 shows these requirements from the viewpoint of a user.

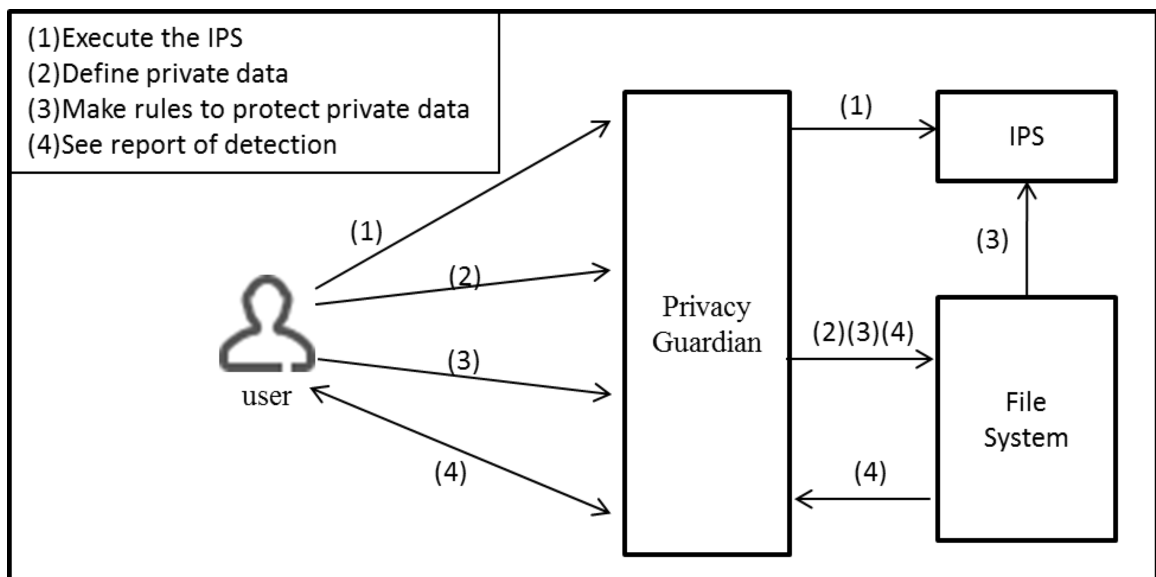


Figure 9: The requirements from the viewpoint of user

4.2 Generating of Snort Rules

In this section, we will describe the design of how PrivacyGuardian generates Snort rules from private data. Snort rule is important because the quality of rules affects the efficiency and detection rate of IPS. The rules will be stored in the device and referenced by IPS to detect the leakage of corresponding private data.

The figure 10 represents the process we design to generate Snort rules in PrivacyGuardian. First, the user chooses files and system resources listed in PrivacyGuardian as his private data. These private data will be accessed and read by PrivacyGuardian. The key extractor extracts the keyword from each chosen file and every entry of private system resources. Then the rule converter converts the keywords to Snort rules. Due to that Snort asks for unique rule ID, the converter has to make sure each rule get different rule ID. Rule options can be different according to the private data type of the extracted keyword. Last step, rule writer writes the Snort rules into *.rules files. With these steps, our LeakDet can detect the leakage of the chosen files with its IPS. The implementation of each component will be discussed in chapter 5.

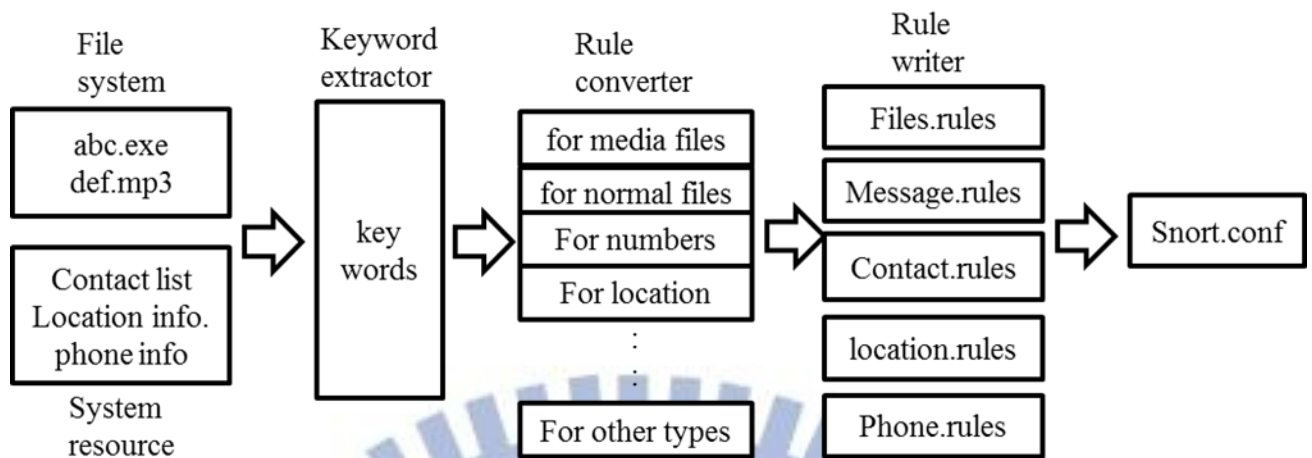


Figure 10: The process of generating Snort rules

We discuss the implementation in detail this section. First we explain how we implement the process of rule generating. And then we will show the screenshots and describe to operating process of PrivacyGuardian.

4.2.1 Process of Rule generating

The process consists of three steps. First, the extractor reads data from private data sources and extracts the keyword from files or each entry of private data sources. Then the converter generates the Snort rule according to the data type of the keyword. Finally the writer writes the rules into categorized rule files. We discuss the details of those components in the following sections.

4.2.1.1 Keyword Extractor

First, the user defines his private data with the interface of PrivacyGuardian. For selecting personal file, PrivacyGuardian provide a file manager for user to browse through the directories and select files he thinks which is private. Considering users usually don’t understand the Linux operating system and it’s common that users and applications only access and stored their data

in the sdcard. So we make a restriction that user can only choose files under the /sdcard directory as his private file. For selecting other system resources like contacts, cellphone information or location information as the private data, user can select the data type by check the checkbox listed in the interface of PrivacyGuardian. Since PrivacyGuardian needs to extract the keyword from the private data, it needs declare the corresponding permissions in manifest file.

After the user's private data is defined, the extractor accesses those private data and extracts a string as keyword from these private data. For the user's private files, the extractor opens the file; reads the input stream of the file and output a string as the keyword represents the file. For the private system resources, the extractor accesses these resources by declaring a content resolver and querying the resources with the resolver. If the resource contains two or more entry like contacts, the extractor will query for all the entries and output one string for each entry as the keyword.

There are several ways to select a string from a file, like random selection, or via the text mining approach. To ensure the result of experiment, we choose the simplest one – only retrieve the first few bytes to be the keyword of a file as the keyword of the file. For the private system resources, we choose the most critical part of the entry as the keyword of this entry. Table 1 shows that which part of the private data will be extracted according to the type of the private data.

Table 1: Matching of the data type and the extracted keyword

Data type	**Extracted keyword**
Contacts	Phone number
Short Message Service (SMS)	The message content or the phone number
Location information	The latitude and longitude
IMEI	The number itself
Files	The first few bytes of the file

4.2.1.2 Rule Converter

After the keyword being extracted, the converter generates the complete Snort rules with these keywords. To form a Snort rule, the converter prefixes and suffixes the keyword to match the format of snort rules. The added strings include the rule action, protocol, source/destination IP address, source/destination port number, and rule options. For increasing detection rate and reducing the system overhead, we can set different options for each class of file types. Table 2 shows the different type of data and the Snort rule options which night help the detection.

Table 2: Recommend rule options for each data type

Types of private data	Options
Contacts and SMS	content:” keyword”; depth: n;
Video, Audio, Graphs	content: “keyword”; dsize: n<;
Location Information	content:”123”; content:”456”;within:n;
General	content: “key”;

The “content” option is one of the most important features of Snort. It allows the user to set rules that search for specific content in the packet payload and trigger response based on

that data. To search for the leakage, we set the keywords we extracted to content option and optimize the rule with adding other options according to the type of private data.

For the rules of contact and SMS keyword, since we believe that the packet length might not too long for sending the phone number, we set the option “depth” to constrain the search area. The “depth” option allows the rule writer to specify how far into a packet Snort should search for the specified pattern. A depth of n would tell Snort to only look for the specified pattern within the first n bytes of the payload.

For detecting private file such as video, audio, graph, we use “dsize” option. The dsize keyword is used to test the packet payload size. This may be used to check for abnormally sized packets. In our case, since the media file size is usually large, the “dsize” option may help us to filter out some small file and increase the efficiency of the IPS.

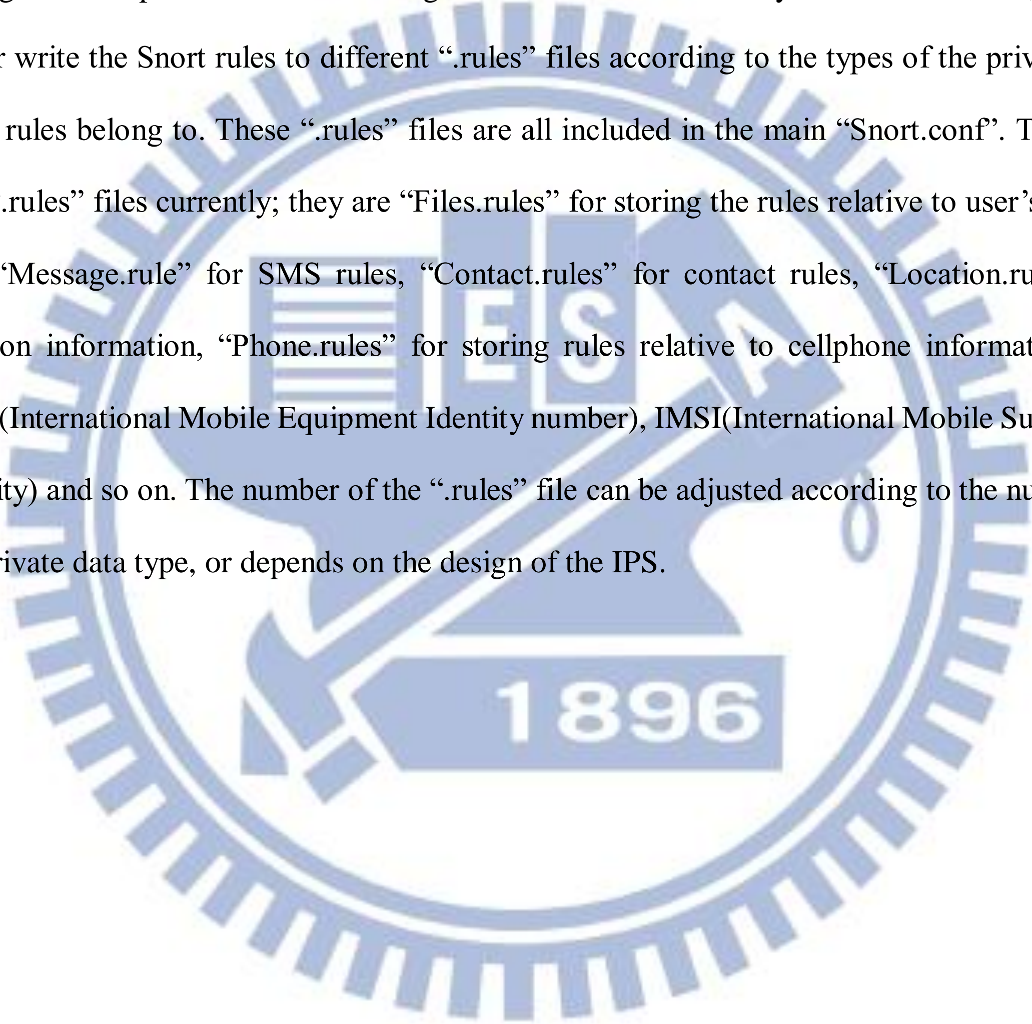
The keyword of location information is composed of two strings: latitude and longitude. The distance of the two keywords might not be too long. To specify this, characteristic we use the option “within”. The within option is a content modifier that makes sure that at most n bytes are between pattern matches using the content option.

Except the options we mention above, there still other method to increase efficiency of the detection. For example, we only detect the outgoing packets with specifying direction operator from the IP address of the localhost to any foreign IP.

Because the data could be encoded to different code while it was transferred via the internet, we use a few common coding methods, like ASCII, UTF-8…, to encode the keyword and generate new rules for each encoded keywords. Hence, the rule converter may output several rules with only one original keyword.

4.2.1.3 Rule Writer

After the rules of each selected private data are generated. The rule writer writes the Snort rules into the configuration file. Snort rules can be written on the configuration file “Snort.conf” directly, or can be written on another configuration file which is included in Snort.conf via adding include option in the main configuration file. To make this system more flexible, the rule writer write the Snort rules to different “.rules” files according to the types of the private data those rules belong to. These “.rules” files are all included in the main “Snort.conf”. There are five “.rules” files currently; they are “Files.rules” for storing the rules relative to user’s private file, “Message.rule” for SMS rules, “Contact.rules” for contact rules, “Location.rules” for location information, “Phone.rules” for storing rules relative to cellphone information like IMEI(International Mobile Equipment Identity number), IMSI(International Mobile Subscriber Identity) and so on. The number of the “.rules” file can be adjusted according to the number of the private data type, or depends on the design of the IPS.



5 Implementation

We discuss the implementation of LeakDet in his chapter. In section 5.1, we introduce how we build the IPS. Then the implementation of PrivacyGuardian, including interface and rule generating, will be discussed in section 5.2.

5.1 Build the IPS on Android

In section5.1, we discuss how to build the IPS on Android. Since Snort uses Libpcap to capture the packets on the network interfaces and then search for the keywords, it needs to be run with root authority or the execution will be denied due to the Linux security mechanism. To build our system environment, we have to root the Android phone first. We also cross-compile Snort and Snortsam to build executable that can be run on Android system.

We follow the process proposed in [26] to root the Android phone and cross-compile Snort and Snortsam to Android platform. Since the version of the smartphone we use was the same as the smartphone used in [26], we don't have to change any part of the rooting process. In cross-compilation, since we use the latest version of Snort, we have to modify the process proposed by [26]. Furthermore, we solve the dynamic-link problem and cross-compiled the latest version of Snort successfully. The version of the software and hardware we used is summarized in Table 3. The process of cross-compiling Snort and Snortsam is shown in following sections.

Table 3: The version of software and hardware we used

Operating system on host machine	Ubuntu 11.04
Smartphone version	HTC Magic 32A
Mod used on smartphone	CyanogenMod
Cross-compiler	Sourcery G++ Lite
Snort version	2.9.0.3

5.1.1 Cross-compile Snort on Android

In this section, we'll explain how to cross-compilation Snort. First we cross-compiled the libraries which are required by running Snort, including libpcap, libpcre, libdaq, libdnet and zlib. Second, we patched Snort with the patch provided by Snortsam and cross-compiled Snort itself.

Before all the cross-compilations, we install the tool-chain Sourcery G++ Lite, whuch can be downloaded from [4].

5.1.1.1 Libpcap

We started on cross-compiling libpcap. The installation of libpcap requires bison and flex to be installed in the machine. They can be simply downloaded and installed with apt-get command. We use the latest libpcap version 1.2.1. Since the version is different from the version used in [26], we need to add a new flag "**--with-pcap=linux**" to the configuration command.

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

Now we explain the option used in the configure step. The option “CC” is used to assign the c compiler for compilation, we use the c compiler provided by the tool-chain Sourcery G++ Lite . The option “-host” is used to assign the target platform, in our case, the target platform is arm-linux. The option “-prefix” is used to assign the location of the executable after the compilation is done. The option ”LDFLAGS” is used to inform the compiler to link libraries statically. If we link the libraries dynamically as default, the program cannot be executed if there is no libraries need for the execution of the program on target machine. In the case of porting Snort on Android, there might not be some critical libraries for Snort to execute on the Android, so we need to link all the necessary libraries statically. After configuration, we do the step of compilation and installation.

5.1.1.2 Libpcre

Libpcre is the library used bt Snort to handle the formal expression. For compiling libpcre, the option is the same as the option we used in compiling libpcap. We only need to and additional option “CXX=arm-none-linux-gnueabi-g++” to assign the c++ compiler for compilation.

We use the following command to configure the cross-compilation of libpcre:

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

5.1.1.3 Libdaq

Snort use Data Acquisition library (DAQ) for packet I/O. The DAQ replaces direct calls to PCAP functions with an abstraction layer that facilitates operation on a variety of hardware and software interfaces without requiring changes to Snort. For compiling libdaq, we have to use “--with-libpcap-includes” and “--with-libpcap-libraries” to assign the directory of libpcap library file and libpcap header file to configure the compiler.

We use the following command to cross-compile libdaq:

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

After we run the following command, there might be an error message "cannot run test program while cross compiling". This is because the test program has been compiled to run on ARM and can't be run on X86 instruction set. We fix the problem by mark the code which cause the error in the configure file to avoid the execution of the test program.

5.1.1.4 Libdnet

Libdnet provides a simplified, portable interface to several low-level networking routines. For compiling libdnet, we assign the library directory of libpcap.

We use the following command to configure the cross-compilation of libdnet:

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

5.1.1.5 Zlib

Zlib is a software library used for data compression. Last we compile zlib, which don't need other flags. Assign the compiler, host, prefix and LDFLAGS to configure and compile.

We use the following command to configure the cross-compilation of zlib:

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

5.1.1.6 Snort

Before cross-compiling Snort, we have to patch Snort and install the necessary tools. To patch Snort, we download Snort patches from [7]. Unzip the patch file to directory of Snort and run the patch command.

```
patch -p1 < snortsam-2.x.x.diff
```

We also need libtool and autoconf to compile the patched Snort. After downloading and installing the two tools with apt-get, we run the following command to activate them.

```
make distclean
libtoolize --automake --copy
aclocal -I m4
autoheader
automake --add-missing --copy
autoconf
```

Now we can finally start to compile Snort. For configuring the compilation of Snort, we not only assign the c compiler with “CC” option, but also assign other tools with the following option: “CXX”,”AR” and ”RANLIB”. These tools are also provided by Sourcery G++ Lite. Furthermore, we have to assign the directories of libraries we cross-compiled previously to the cross-compiler.

We use the following command to cross-compile Snort:

```
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=/libpcap/lib
--with-libpcre-libraries=/libpcre/lib
--with-daq-libraries=/libdaq/lib
```

```
--with-dnet-libraries=/libdnet/lib
--with-libpcre-includes=/libpcre/include
--with-daq-includes=/libdaq/include
--with-dnet-includes=/libdnet/include
```

If the error message "cannot run test program while cross compiling" shows again, check the configure file and mark the code that cause the error. The reason has been explained above.

We encountered the same problem as "Porting Snort on Android". The executable we generate was dynamically linked even though we set LDFLAGS to "static". The executable cannot run on the machine without the necessary libraries installed on it. After we did some research on the Makefile, we found that Libtool was used by Snort to link libraries while compilation. This seems to be the reason that causes the problem of statically linking. We trace the procedure of compilation by observing Makefile and Makefile.log, and find that the Makefile in "/src" use libtool to link libraries. To solve the problem of statically linking, we add the flag "-all-static" in the libtool command.

5.1.2 Cross-compile Snortsam on Android

To compile Snortsam, we only have to run makesnortsam.sh. But when it goes to cross-compile, we should modify the makesnortsam.sh. Open the makesnortsam.sh and change the "Linux*) gcc ${CFLAGS}..." to "Linux*) arm-none-linux-gnueabi-gcc ${CFLAGS}….". The purpose of this change is to change the default compiler.

There is another problem. Snortsam use the function "system()" to execute Iptable. "system()" calls the executable "sh" in Linux system to execute the input command. Unfortunately, The location of "sh" was fixed to /bin/sh in "system()" of C standard library, while the executable "sh" of Android system exist in /system/bin/sh. To solve this problem we manually modify ssp_iptables.c, the source code of Snortsam. We write a "system()" of our

own version to replace the “system()” of C standard library.

5.1.3 Run Snort and Snortsam on Android

Since we want to activate Snort and Snortsam via the Android app, first we have to put the executable into the target Android mobile device. Considering that normal users may not be skilled in using Android SDK to put files into Android mobile device, we put the executable in the device while installing the application.

We use the following steps to put the executable into the device and execute them: 1). Include the executable in the assets folder. 2). Use getAssets().open("Executable Name") to get an InputStream. 3). Write it to /data/data/app-package-name, where the application has access to write files and make it executable. 4).Use Runtime.exec() to run "/system/bin/chmod 744 /data/data/app-package-name/yourapp". 5). Use Runtime.exec() to execute Snort and Snortsam.

5.2 Implementation of PrivacyGuardian

In this section, we introduce the design and detail of the code structure, interface and the operating process. The code structure of PrivacyGuardian will be clarified first. We will describe the usage of each java class and explain how we achieve the function we design in chapter 3. Then we show the interface and operating process of PrivacyGuardian.

5.2.1 The Code Structure of PrivacyGuardian

There are seven java classes in PrivacyGuardian, and can be classified to two types according to the role they play. As Table 4 shows, three interface classes handle the display of screen and handle the events triggered by the control panel. Other classes are tools to accomplish the functions of PrivacyGuardian we design. In the following sections we introduce the code structure by explaining each java classes into the detail. Figure 11 shows

the code structure of PrivacyGuardian.

Table 4: Types and their functions of java classes in PrivacyGuardian

Type	Class name	Function
Interface	MainActivity.java	Main menu
	PrivacySelectActivity.java	An interface for user to select the type of private data to be monitored.
	FileSelector.java	A file selector for user to select private files to be monitored.
Tools	Api.java	APIs to execute shell commands
	FileService.java	Read or write the files
	ResourceFetcher.java	Get system resources
	RuleConverter.java	Convert the keywords to Snort rule

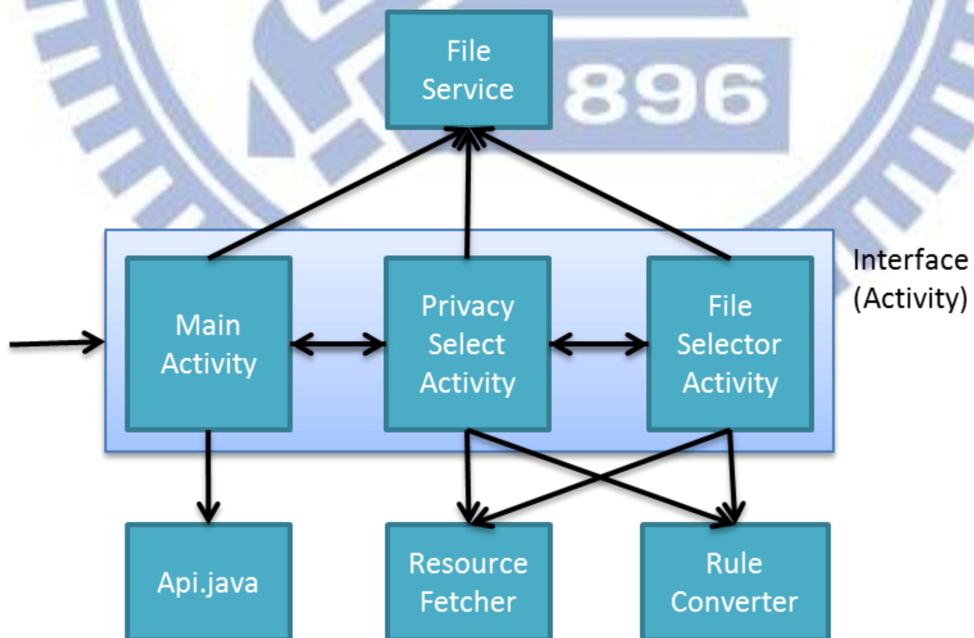


Figure 11: Structure of Classes in PrivacyGuardian

5.2.1.1 Main Activity

The main activity is the view that user sees while he start PrivacyGuardian. In this activity, we defined several buttons to handle the request for switching on or off the IPS, for the reporting, and for the further definition of the private data. We write the handler to process the needed functions while the buttons are pressed. To execute the shell commands for starting or stopping the IPS, we call *run_command* function in api.java with the shell command as the input. We analysis the log file of the IPS and generate the information of the detection, and display on the screen. To go another activity, we create an intent instance and call *startActivity()* with the intent as input.

5.2.1.2 Privacy Select Activity

This is an activity for user to select his own private data. Since the private files are better chosen in the file selector, we provide a several options of private data, like contact information, SMS information and so on as the form of checkbox. We defined the listener to handle the event while the check state of a checkbox is changed. Once the state of a checkbox is changed, the handler checks the state and calls the rule generating process when the state is changed to checked from unchecked. The rule generating process is done with the help of some function classes. There is also a button to enter the file selector, which triggered the intent of start a new activity while pressed.

5.2.1.3 File selector

The file selector is an interface for user to choose his private files. The selector consists of an activity that defines the handler to handle the file choosing functions while a file is selected, and an adaptor that defines template and components for each entry. The chosen files will be remembered in each time with maintenances of a record file. After the user choose his

file and press the confirm button, the rule generate process will be activated. We can change the variable *root* to constrain the directory that user can access with this selector.

5.2.1.4 Api

The Api class is designed to handle the requests of executing shell commands. We reference the source code of DroidWall to construct this class. When *run_command()* is called, the program will create an instance of *ScriptRunner()* to execute "*su*" in Linux and perform some processing of input script string to ensure executing the script correctly.

5.2.1.5 File Service

This class handles all the file reading and writing in PrivacyGuardian. After user select his private files, the function *readSDcardFile()* will be called and then output a keyword for each file selected. Here the FileService class plays the role of the keyword extractor in our design. While the rules are generated, the function *writeSnortConf()* will be called to write the rules into configure files. The file name of configure file is specified in the input parameter. Here the FileService class plays the role of the rule writer in our design. In addition, *readSDcardListFile()* reads the log file of the IPS, and store them in the list data structure for further parsing.

5.2.1.6 Resource Fetcher

ResourceFetcher class is programmed to fetch system resources and output the keywords of these resources. The rule generating process triggered in the PrivacySelectActivity calls functions in ResourceFetcher. ResourceFetcher plays the role of the keyword extractor in our design.

5.2.1.7 Rule Converter

Rule converter converts keywords to rules. The keyword might be from different private resources, so the function *convertPrivacyToRule()* takes an identifier as input to figure out which private source the keyword belong. We predefine several strings of prefix and postfix and create Snort rules simply concat the strings and the keyword together. RuleConverter also has to ensure that every rule get an unique ID

5.2.2 User Interface and Operating Process

We will introduce the process of operating PrivacyGuardian and describe the interface this section.

Figure 12 shows the main activity user can see once he activates PrivacyGuardian. The main activity contains five buttons. The first three buttons from top is for user to control the IPS, include Snort and Snortsam. User can turn on, turn off or restart the IPS with just tapping the buttons with his finger. The main activity of PrivacyGuardian will show the toast - a view containing a quick little message - to inform user when Snort and Snortsam is started or stopped.

Figure 13 and 14 shows how PrivacyGuardian informs user with a toast while turning on and off the IPS. Furthermore, user can switch to another activity to define his private data by tapping the fourth button from top, or access the report of the detection by tapping the button.

Figure 15 shows how PrivacyGuardian report the result of detection to user. The report notifies user which IP has been blocked by the IPS and the Snort rule’s ID which triggered this block.

Figure 16 shows the activity for a user to select the private data. There is many predefined types of private data, including contacts, short messages, phone information and the information of location. Once user press the checkbox, the rule generating process will be

taken, generating Snort rule for each entry of the chosen type. If user unchecks the checkbox, the IPS will stop detecting the corresponding type of data

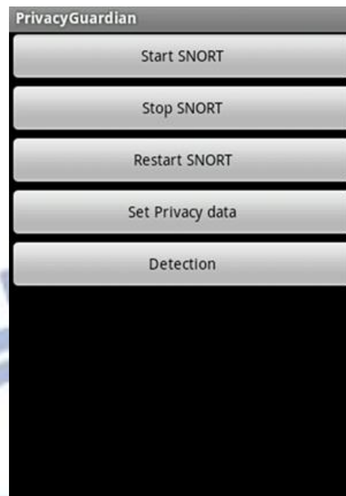


Figure 12: The main activity of PrivacyGuardian

Figure 13: PrivacyGuardian informs user for activating Snort

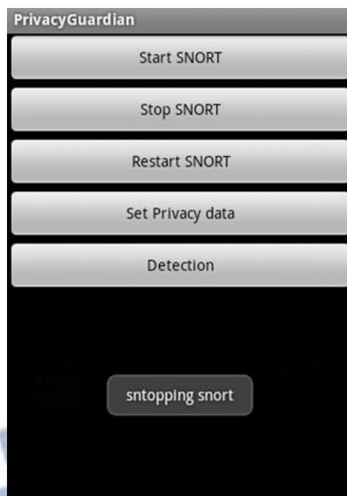


Figure 14: PrivacyGuardian informing user for stopping Snort

.Figure 15: The report for the result of detection

Figure 16: The interface for selecting private data

To define personal private files, user can press the button in the second activity and go to the file selector. The interface of file selector is presented as a file manager, allowing user to browse through all the directories under the directory “/sdcard”. User can select his private data by checking the checkbox. He can choose any number of the file to be his private file. After user press the confirm button, the rule generating process will be taken, generating Snort rules to detect the leakage of these chosen files. The selector will remember the chosen file, so user doesn’t need to re-check those files that he want to protect while he re-enter the file selector. The interface of file selector is shown in Figure 17 and 18.

Figure 17: The interface of the file manager

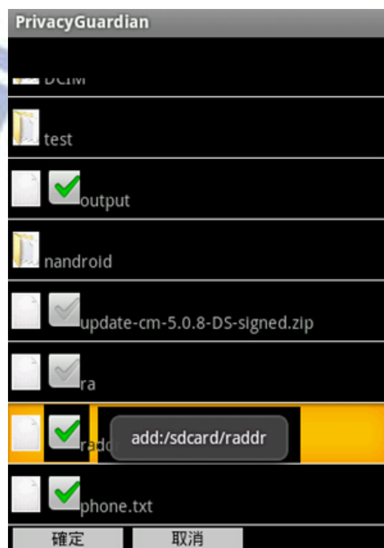


Figure 18: The warning for selecting a file

6 Experiment and Discussion

To verify that our LeakDet is able to detect the leakage and block the destination IP of the leakage after the detection, and that how much system resources LeakDet will consume, we took some experiments described below.

6.1 Functionality test

The purpose of this experiment is to test if IPS works as we expected. The IPS in LeakDet should not only detect the leakage of private data but be able to block the incoming network flow from the attacker IP with the help provided by firewall.

We use the botnet environment proposed by [25] to construct a scenario of attack. We install LeakDet and run it on victim cellphone. Meanwhile we trigger the attack and make the victim cellphone become a zombie controlled by botmaster. After the botmaster getting control of the victim, he tries to steal the data stored in user's sdcard. Then we observe if the IPS can detect this data stealing. We will describe the process of experiment in detail in the rest section.

After LeakDet is installed in the victim cellphone, we choose a file located in sdcard as the target private file for user. PrivacyGuardian generate the corresponding Snort rules and write in configuration file of the IPS. Then we start the IPS for detection the leakage of the target file.

We set up the botnet environment on botmaster's server, the environment includes a web server for running attack code to exploit the mobile device which connects to it, a command and control (C&C) server for a bot to connect and listen to the botmaster's command. We also installed and run the application provided by [25] on the cellphone. The application pretends that it's an application for looking up the weather (Figure 19). The real purpose of the application is to connect to web server which provides the access to the malicious webpage which contains and executes the code for buffer overflow attack. Once the attack code is

executed, the victim cellphone will connect to the C&C server and is controlled by botmaster with the authority of administrator. Figure 20 shows the victim is connected to botmaster's server.

Figure 19: The trigger of botnet attack

```
who
1          bot       127.0.0.1          <- botmaster
2          bot       111.251.172.204
```

Figure 20: The victim connects to the botmaster

After the victim connecting to the C&C server, the botmaster can browse the sdcard of the victim user and view the content of all the files by sending the commands to his bots. The botmaster send command to victim cellphone for viewing the target private file: "pw.txt". After receiving the cat command, the victim cellphone try to execute the shell command cat to print the content of the target file and then transfer to botmaster through the internet. Figure 21 shows the command from botmaster.

```
system/xbin/cat /sdcard/pw.txt >all
```

Figure 21: The command of stealing data from the botmaster

Since we have already defined “pw.txt” as the private data in PrivacyGuardian, Snort can detect the leakage of target file with referencing the rules. Then the IP of the botmaster has been block by the IPS due to the configuration in our IPS. The result is reported to use, too.. Figure 22 shows the list of all the rule chains of the Iptables, and the address (ccis32.cs.nctu.edu.tw) has been written in both source and destination section of the rule chain, which means the packets from or to the botmaster’s IP can’t pass the firewall.

```
iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
DROP       all  --  anywhere              ccis32.cs.nctu.edu.tw
DROP       all  --  ccis32.cs.nctu.edu.tw  anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
DROP       all  --  anywhere              ccis32.cs.nctu.edu.tw
DROP       all  --  ccis32.cs.nctu.edu.tw  anywhere
```

Figure 22: The firewall block the attacker's address

Through this experiment, we know that LeakDet functions properly. LeakDet detects the leakage according the predefined rules with its Snort part, block attacker’s IP with its firewall part, finally reports to user with the interface provided by PrivacyGuardian.

6.2 Consumption of cellphone resources

Because the system resources like the CPU time, memory consumption, battery consumption, in mobile devices are limited, the consumption of running a program is important for an application. We take some experiments to test that how much resource our LeakDet will consume. The resources consumption we tested includes battery power, CPU consumption and memory consumption. We also perform a comparison between new and old cellphones. Table 5 is the hardware data of the two cellphones we use in the experiments.

Table 5: The hardware difference of HTC Magic and HTC Desire

	HTC Magic	HTC Desire
RAM	288MB	768MB
standby time	420 HRs	460HRs
Battery capacity	1340mAh	1600 mAh
CPU	528MHz (Single-core CPU)	1.5 GHz (Single-core CPU)

To test the battery consumption, we run LeakDet on both cellphones for one hour. We play the video with YouTube application while testing because we want to simulate the scenario of heavy network flow of a normal user. We also compare the result to the power consumption while only watching the video. Table 6 shows the testing result.

Table 6: Power consumption when watching videos

	Run with LeakDet	Power consumption(% / hour)	Battery lasting time(hour)
HTC Magic	No	35.8%	2.8
	Yes	45.89%	2.17
HTC Desire	No	19.2%	5.2
	Yes	27.7%	3.61

The memory consumption is shown in Table 7. LeakDet consists of two parts: the frontend written by JAVA and the IPS part, which consists of Snort and Snortsam written by C. The Iptables is also the part of IPS, but we don't count its memory consumption to the total because the Iptables is embedded in the Linux kernel and run with the operating system. We count the memory consumption of both parts and compare the summation of memory

consumption to other popular apps.

Table 7: The memory consumption of LeakDet and popular apps

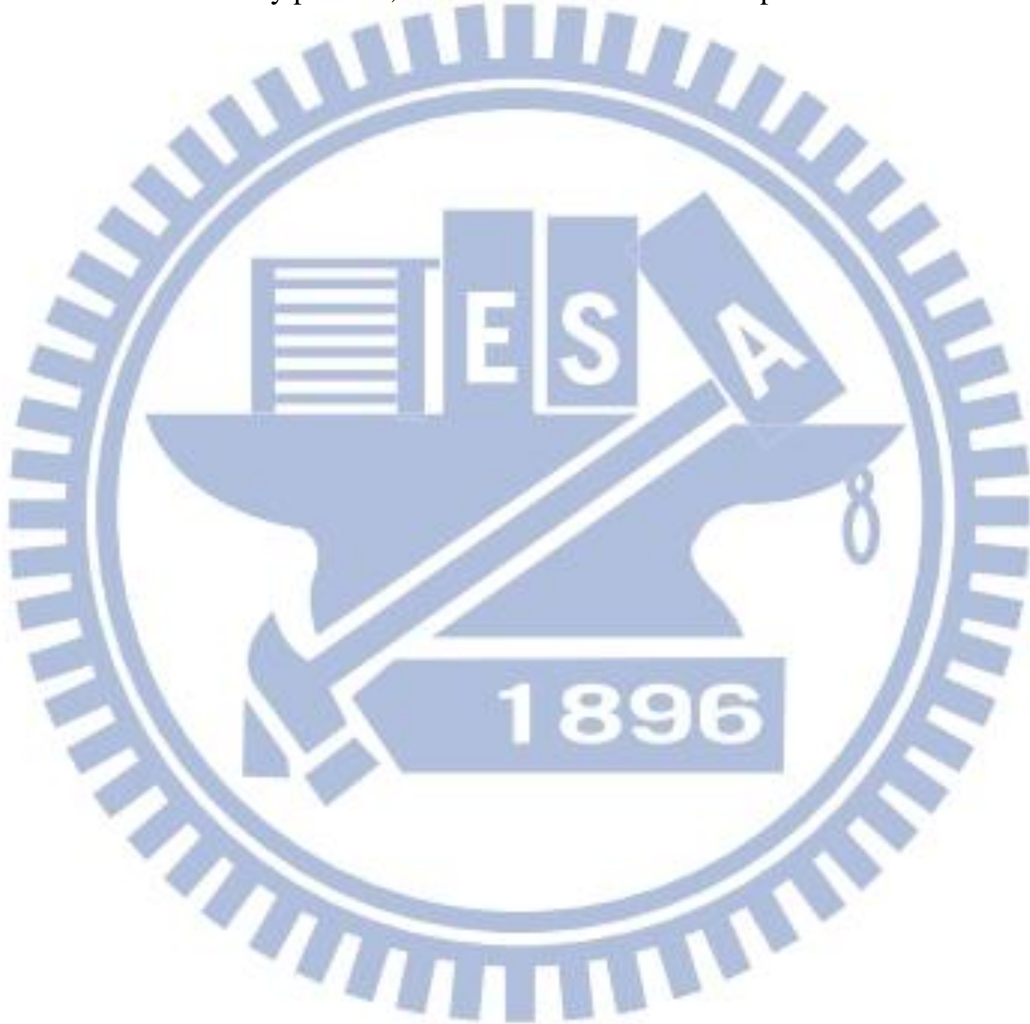
Apps	Android app (KB)	Snort (KB)	Snortsam (KB)	Total (KB)	Compare to LeakDet (%)
LeakDet	21824	11392	388	36228	115%
Facebook	28980			28980	100%
Line	30736			30736	104%

The result shows that the power consumption is significantly high when there is a lot of network flow. Through our observation, we find that the power consumption of LeakDet is in proportion to the amount of the monitored packet in the network flow and the number of rules. If the network traffic is heavy like the situation of watching the video stream on smartphone, the power consumption will be significant high. On the other hand, if the amount of network flow is low, the power consumption may be low, either.

In memory consumption, we compare the memory consumption with two popular applications: Facebook and Line. Mention that the memory consumption of the Facebook and Line we calculated is only the memory usage while they're running in background. The total memory consumption of LeakDet is a little bit higher than other apps. Since the app part of LeakDet may not always in executing, the real background memory consumption of LeakDet is less than a half to the background memory consumption of Facebook and Line. Furthermore, an HTC Desire mobile device has 768MB of RAM, the loading of memory in executing LeakDet might not be a problem.

To test if there is any latency issue while executing LeakDet, we turn the IPS on and play some games on the cellphone while connecting to the internet. The result is that we don't sense any obvious latency in the game play with LeakDet executing in the background and connecting to internet. The result shows that LeakDet cause almost no latency.

Through these experiments, we find that the power consumption is the most critical bottleneck of the system resource. To reduce the consumption of the electricity, there are some improvements can be done. First way is to reduce the number of the rules. Since the purpose of our LeakDet is to detect the leakage, we can make the rules detect only outgoing packets. Second is to reduce the packet that needs to be checked by the IPS. We can set more options to filter out the unnecessary packets, so that the resource consumption can be reduced.



7 Conclusion

The growing number of smartphone and its user make smartphones under threaten of attacks, especially the leakage of privacy. The vulnerable Android security mechanisms also increase the risk of attacks on Android phone. To protect the user privacy, we propose LeakDet, a tool for detecting the leakage of private data on Android.

LeakDet consists of two parts: an IPS for detecting the leakage and blocking the attack afterwards, and a frontend called PrivacyGuardian for user to define his own private data and control the IPS. In its IPS part, LeakDet collect packets and analysis them with a set of predefined rules to detect the private data leakage through the internet. Once the leakage is identified, LeakDet takes an instance response, and block the packet with its destination or source IP address belongs to the attacker. In the PrivacyGuardian, the private data or information the user chooses will be converted to some rules. These rules are for the IPS to detect the leakage of chosen data. With LeakDet, the data stolen can be detect and the user will be informed in the same time. Also, the following attack from the same IP will be blocked because the firewall is set to block it.

To verify the functionality of LeakDet, we simulate and construct a botnet attack environment. We apply the attack of stealing private data with the botnet on an Android cellphone and try to detect it with LeakDet. The result of the experiment proves that our LeakDet can not only detect the leakage of the private data, but block the attacking IP by the firewalls. We also take some test to make sure the power and memory consumption of LeakDet. We find that the power consumption is in proportion to the network traffic flow and the number of the rules the IPS referenced. The power consumption is almost negligible while in a few network flows and a little number of rules, but it will consume a lot of electricity with the heavy traffic.

To the best of our knowledge, LeakDet is the first implementation that detects and prevents the leakage of private data with the help of IPS on Android. Different from most of other solutions, LeakDet don't require user to wipe out his original operating system and install a modified version of operating system but only require for root authority. It's easy for user to install and start to apply it on detection work.

7.1 Future work and Discussions

In the future work, we find that there may be some directions to improve the performance of LeakDet. We conclude two concepts that might able to improve LeakDet in the future: 1). Reduce the resources consumption of running LeakDet. 2). Improving the accuracy of the detection. It seems the two concepts are conflicted since we need to do more analysis and comparison to reach high accuracy while this may cause more consumption of system resources.

Since the power consumption is the most serious problem and is in proportion to the network traffic and the number rules, the trivial solution is to reduce both of them. To reduce the packet number, we can set more strict rules to filter out packets before they analyzed to reduce the calculation efforts. The options of Snort rules like "depth", "distance", "offset", "within" can help to reduce the keyword searching area.

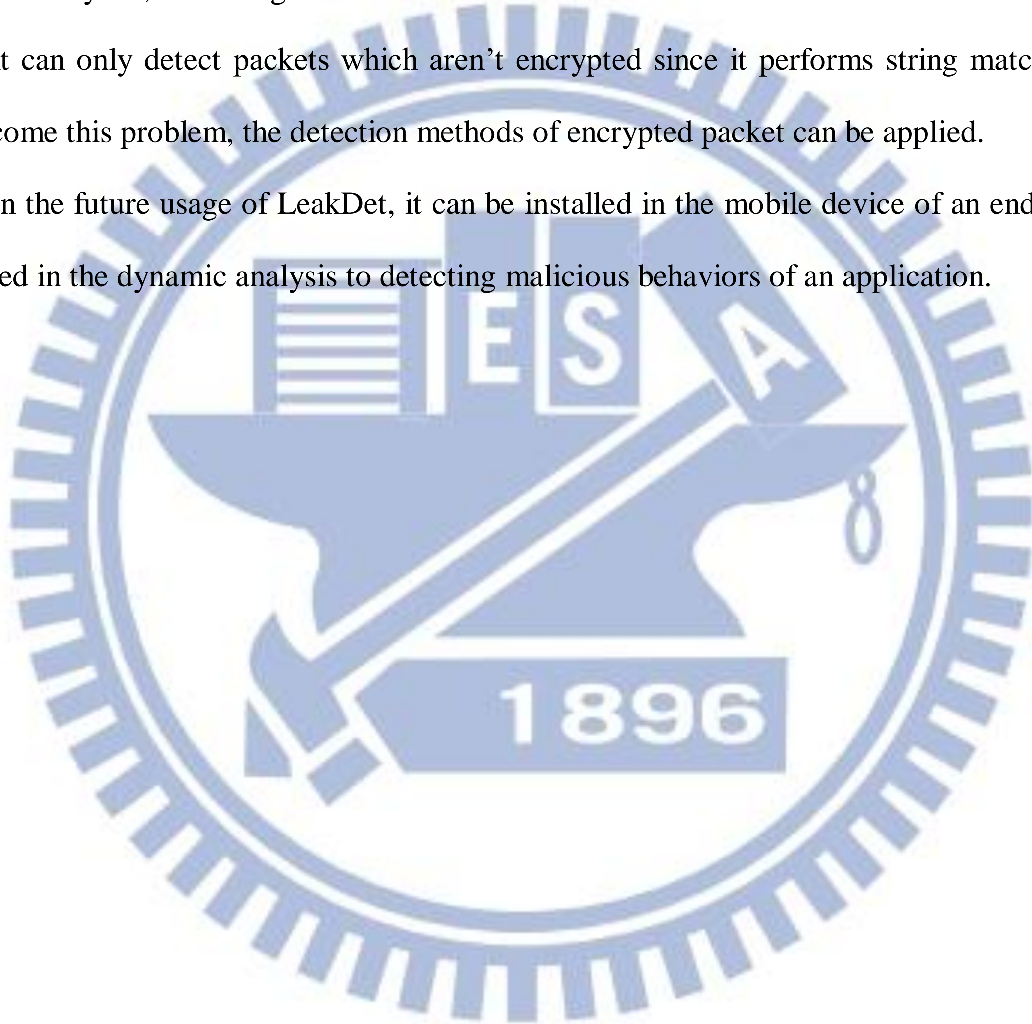
While the cellphone disconnecting to internet, it is not necessary to keep the IPS running. We can add a function to monitor and start the IPS only when the cellphone is connected to internet. Also we can inform user when they are going to use the application or system functions which will lead to a lot of network traffic. This will reduce the unnecessary waste of electricity.

Snort is an effective IDS, and provide many strong functionalities. Some of these functions or modules don't really effect the detection of private data leakage on mobile device. In PC, the redundant modules might not cause the serious problem in consumption of system

resources. But in mobile devices, the redundant function is a waste of resources. To reduce this redundancy, we can study the structure of Snort and remove the irrelevant function or module.

To enhance the accuracy of detection, more encoding type can be add to the rule generating process to identify more potential leakage. The pattern of leakage packet can be further analyzed, and design better rules for detection. The most trivial defect of LeakDet is that it can only detect packets which aren't encrypted since it performs string matching. To overcome this problem, the detection methods of encrypted packet can be applied.

In the future usage of LeakDet, it can be installed in the mobile device of an end user, or be used in the dynamic analysis to detecting malicious behaviors of an application.



8 Reference

1. Android Developers (2012/8/10)
   http://developer.android.com/index.html

2. Android - Wikipedia
   http://en.wikipedia.org/wiki/Android

3. Appshield
   http://www.globalscape.com/products/appshield.aspx

4. Codesourcery (2012/8/10)
   http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview/

5. Droid Wall
   http://code.google.com/p/droidwall/

6. Snort (2012/8/10)
   http://www.snort.org/

7. Snortsam (2012/8/10)
   http://www.snortsam.net/

8. Snortsam Documentation (2012/8/10)
   http://doc.emergingthreats.net/bin/view/Main/SnortSamDocumentation

9. Two Thirds of New Mobile Buyers Now Opting For Smartphones Comments Feed | Nielsen Wire
   http://blog.nielsen.com/nielsenwire/online_mobile/two-thirds-of-new-mobile-buyers-now-opting-for-smartphones/

10. XDA Developers
    http://forum.xda-developers.com/

11. Christopher Mann, Artem Starostin, "A framework for static detection of privacy leaks in android applications", Annual ACM Symposium on Applied Computing, pp. 1457-1462, March 2012

12. Thomas Bl¨asing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak, " An Android Application Sandbox system for suspicious software detection", International Conference on Malicious and Unwanted Software, pp. 55- 62,

Oct. 2010

13. Clint Gibler, Jonathan Crussell, Jeremy Erickson, Hao Chen, “AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale”, International Conference on Trust and Trustworthy Computing, pp. 291-307, June 2012

14. Samruay Kaoprakhon, Vasaka Visoottiviseth, “Classification of audio and video traffic over HTTP protocol”, 9th international conference on Communications and information technologies, pp. 1534-1539, Sept. 2009

15. Danny Iland, Alexander Pucher, Timm Sch¨auble, “Detecting Android Malware on Network Level”, Dec 2012

16. Tobias Limmer, Falko Dressler, “Improving the performance of intrusion detection using Dialog-based Payload Aggregation”, IEEE Conference on Computer Communications Workshops, 2011, pp. 822 – 827, April 2011

17. Giuseppe Aceto, Alberto Dainotti, Walter de Donato, Antonio Pescap´e, PortLoad: Taking the Best of Two Worlds in Traffic Classification”, IEEE Conference on Computer Communications Workshops, pp. 1-5, March 2010

18. Gianluca La Mantia, Dario Rossi Finamore, Alessandro Finamore, Marco Mellia, Michela Meo, “Stochastic Packet Inspection for TCP Traffic”, IEEE International Conference on Communications , pp. 1 - 6, May 2010

19. S. Chakrabarti, M. Chakraborty, I. Mukhopadhyay, “Study of snort-based IDS”, International Conference and Workshop on Emerging Trends in Technology, pp. 43-47, Feb 2010

20. William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth., ”TaintDroid, An Information-Flow Tracking System for Realtime Privacy”, the USENIX Symposium on Operating Systems Design and Implementation, Oct. 2010

21. Yajin Zhou, Xinwen Zhang, Xuxian Jiang, Vincent W. Freeh, “Taming information-stealing smartphone applications (on Android)”, international conference on Trust and trustworthy computing, pp. 93-107, June 2011

22. Carsten Willems,Thorsten Holz, Felix Freiling, “Toward Automated Dynamic Malware Analysis Using CWSandbox”, Security & Privacy, IEEE, pp. 32-39, March-April 2007

23. Symantec - Mobile Security Research Report, Carey Nachenberg, VP, Fellow, June 2011

24. Snort Manual, The Snort Project, July 2011

25. Liu, En-Bang, Tzeng, Weng-Guey," Mobile Botnet Detection on Android", 國立交通大學，碩士論文, 民國 100 年

26. Lo, Jih-Hong, Tzeng, Weng-Guey," Porting Snort on Android", 國立交通大學，碩士論文, 民國 98 年