Contents

List of Figures

List of Tables

1 Introduction

About 9 months ago, we began to survey the security issues on Android. We found that there are no network security applications on Android. So we started to try to find a way to provide the network security application on Android. At first, we study about the architecture of Android, and how an application works on it. After the study and many considerations, we finally decided to port Snort to Android, and the way we used is easily applied to the various editions of Android, that is also convenient for the application promotion. All the Android cell phones will have the protection of Snort after the publication of this thesis.

In this thesis, we have cross-compiled the Snort to the Google Android cell phone. Hence, the Android platform has the first network security application. In order to provide the security of surfing the internet, we use the string-matching functionality of Snort to check the packets that flow into our Android cell phone.

After the cross-compilation procedures, we also give some experimental results about the power consumption and the packet capturing functionality of Snort to prove that the Snort can retain its advantage even when it works on cell phones.

With this kind of application framework, we can port more security applications to the Android platform.

Figure 1: The icon of Android

1.1 Android

The Smart phone has been developed for a long time. However, the operating systems on cellphone are independently developed and closed to public access. Until Google released the open source operating system called Android in 2007, the development of the cell phone was finally unlocked to the public and become more and more popular. In addition to Android, there are some other open source cell phone operating systems, such as Symbian and Windows Mobile. However, they only release the application interface for developers to build their own applications. Google Android is more open because it releases not only the Application interface, but also the operating system source codes, so users can even add their own codes to the source code or fix the bugs in the source code.

Android is an operating system which is designed for embedded systems, such as cell phones or notebooks. Android is developed by Open Handset Alliance (OHA) which is led by Google. Users can develop the application they want on Android. Furthermore, they even can fix problems and bugs of the Android functionalities. Figure 1 is the logo of the Android.

With various editions of API, Android system has faced a manage problem. Its application has to choose the application level in order to let user know if this application can work properly on their Android cell phone. For example, if the API level of the Android Cell phone is 2.0, the application whose API level is 1.6 may not work correctly on the cell phone. It is because that the application may use some functions which is not supported by the cell phone. Besides this problem, since every user can get the source code of Android through the internet, the system security is another big challenge to Android.

In this section, we introduce the architecture of the Android. Android consists of the native embedded Linux system and some application frameworks which are written in the JAVA language. We can further divide the architecture into four layers as Figure 2 shows. We introduce each part of them in a bottom-up manner.

The first layer is the Linux kernel layer, Android uses the LINUX Kernel 2.6. The kernel is in charge of hardware drivers, network and power management, and the tasks of the operating system such as thread management, memory management, file management, etc.

The second layer is the library layer. The second layer consists of many open source code libraries. For example, the libc, the library of OPENSSL, SQLite and Webkit that is responsible for webpage viewing. This layer also contains some media libraries like OPENGL which is for the display of graph and video. Android runtime libraries which provide the JAVA core libraries and the Dalvik virtual machine which can convert the JAVA bytecode into

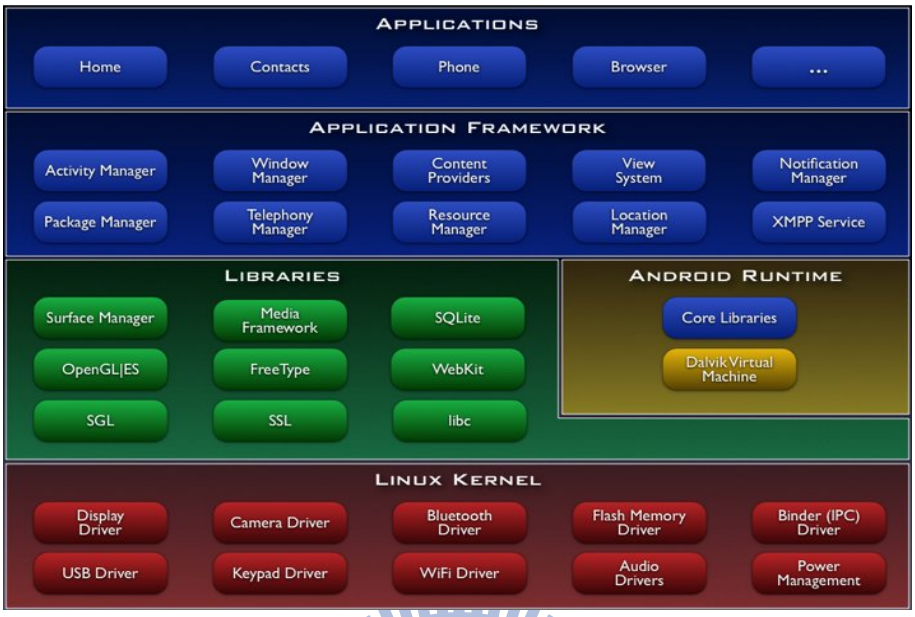


Figure 2: System architecture of Android

the *.dex* file format are also in the second layer.

The third layer is the application framework which is a set of Android core application framework APIs. The application framework can make the development easier and convenient to the application developer.

The final layer consist of JAVA applications that includes the file management, contacts, phone calling, etc.

All the applications on Android must be written in JAVA language. Next, Dalvik virtual machine convert the JAVA bytecode into the DX bytecode which can run on the Dalvik virtual machine. The development of the applications is similar with the traditional JAVA ME development, and the graphic user interface (GUI) is designed by the XML. However the operating

system layer still consists of C language program. As a result, there are still some methods to execute the program that is written in the C language.

We summarize two methods as follows:

1. The first way to execute the program written in C is using the Android Native Development toolkit (NDK) , with Android NDK you can directly call the functions in libraries which are written by C language instead of the libraries of the application layer.

2. The second method is to cross-compile the program written in C, so the execution file can work properly on Android, and then design a front end GUI to invoke the execution file.

In this thesis, we use the second method. We cross-compiled the Snort into the execution file which can run on the Android platform, also in the ARM architecture.

We put the Snort execution file into the */data* directory which is in the Android file system. We use the GUI to invoke the execution file, so that we can directly execute the Snort on the cell phone screen. The advantages of this method are that we can develope the program on the Linux operating system, and we only need to change the input arguments to the execution file in the JAVA application layer.

The Architecture of invoking the execution file in the Linux layer is shown in Figure 3.

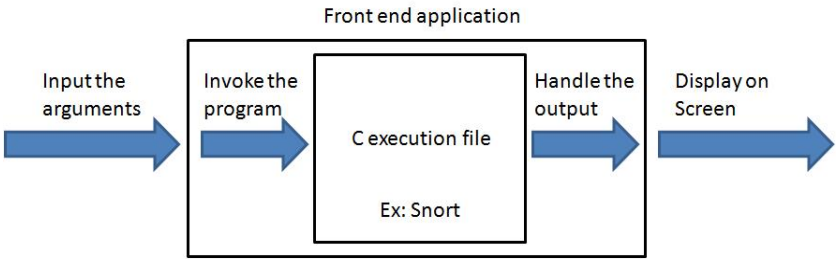


Figure 3: Flow diagram of how front end application works

1.2 Intrusion Detection System

When we are surfing on the internet, our computer may be under the threats of many attacks such as virus, worms, Trojan horses and cross-site scripting, etc. There are also many methods to protect our computers from attack. One of the methods is using the intrusion detection system (IDS) to compare the attack with our pre-defined signatures. When the hacker launches network attacks on our computers, some specific pattern exists. Those patterns are called "signature". Maybe the network flow will grow rapidly, or all the memory will be consumed by some programs, or the CPU will always run with the maximum speed.

We classify the IDS into two categories by the detection method: anomaly-based IDS and signature-based IDS as Figure 4 shows, anomaly-based IDS is to monitor the resources of the operating system like the CPU usage or memories. If some kind of resources usage is far beyond the threshold value, the IDS will alert the system administrator. Signature-based IDS uses the string-matching method by comparing the contents of packets with the signatures in the signature database, to determine whether there are malicious contents in the packets.

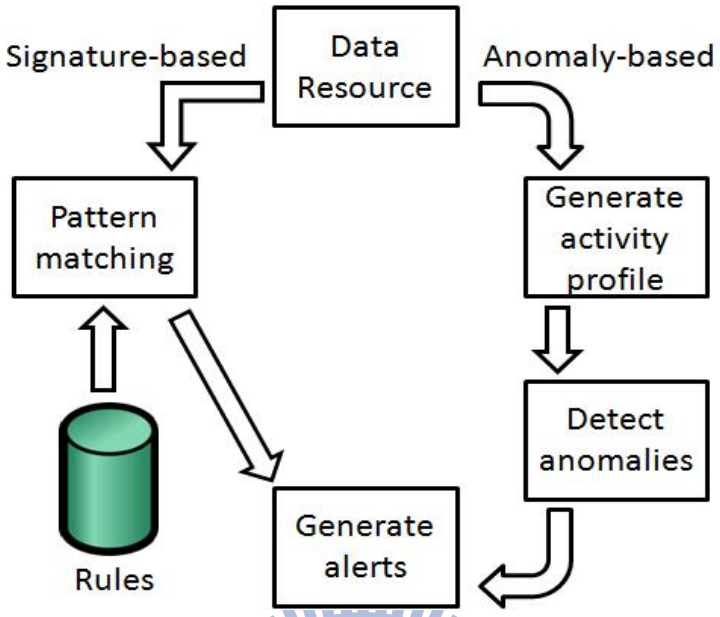


Figure 4: The architecture of IDS

By the detection target, we can divide the IDS into Host-based IDS(HIDS), which monitors the resources in the system, and Network-based IDS(NIDS), which scans the packets that flow into our computer. Most NIDSs use the signature-based detection method.

In our work, we have ported Snort to the Android platform. Snort is a popular, signature-based NIDS. It has powerful functionality and is a share-ware. Snort has a good software design architecture such that makes it can develop many modes and plug-ins separately by many users or engineers on the internet. The architecture of Snort is shown in Figure 5. A community keeps maintaining and updating Snort so far. There are many attacks against IDS, such as the fragmented packets attack, DOS attack, etc. Here we only consider the simplest problem: if the Snort will drop the packets when receiving large amounts of packets. In the chapter 6, we will have an

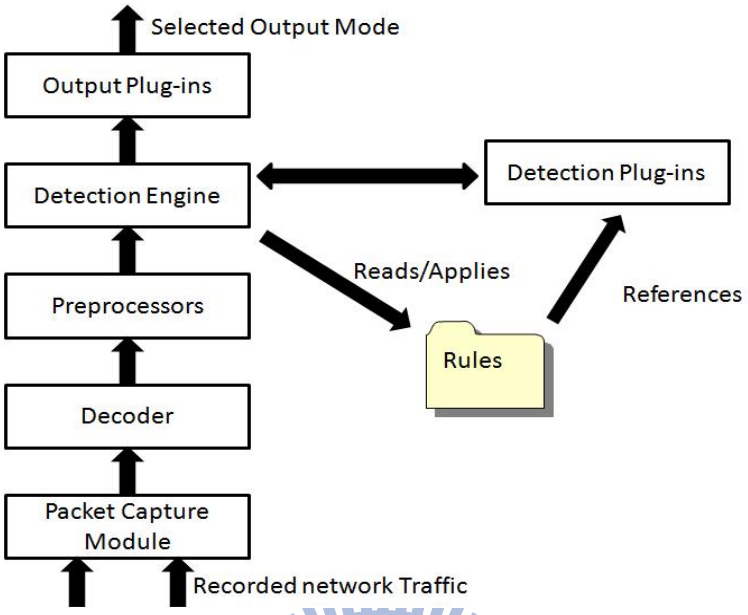


Figure 5: The architecture of Snort

experiment about the packet capturing.

1.3 Cell Phone Security

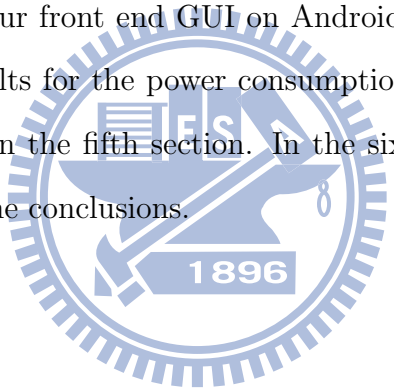
The security issues of smart phones can be classified into two types, one is whether we store private files securely or not and the other is if the cell phone is attacked by the attacker through the internet. Because the smart phone operating system like Symbian or Android are usually equipped with the wireless internet accessibility, it is much harder to prevent the cell phone from attacks. There are still some more attacks like the MMS attack and the Bluetooth attack.

Through our survey, we found that there are some anti-virus softwares on the Symbian and windows mobile. We also found iptables, one of the Linux

popular firewall package on Android, and tcpdump, the packet capturing package on Android. However, there is still no anti-virus software on Android till now, and there is no IDS on Android. So we decided to port the Snort on Android. As the first intrusion detection system on Android, we can provide our cell phone more security with Snort.

Roadmap:

In the following sections, we will list the software and hardware environments of our construction in the second section. We then show our main work, cross-compilation Snort on Android in the third section. In fourth Section, we introduce our front end GUI on Android and give some screenshots. Experiment results for the power consumption and packet capturing ability will be showed in the fifth section. In the sixth section we will give some discussions and the conclusions.



2 Related works

In 2008, Schmidt et al. introduce some important features of Android such as file right managements and the description of image partitions [16]. The authors also port some native Linux-based programs to Android platform such as Busybox, Bash, Chkrootkit, etc. According to their paper, iptables and Snort are not provided on Android at that time. So we start to survey about how to port the iptables and Snort on Android. When we started to port the Snort to Android, iptables is already provided on Android, which is a popular firewall application used on Linux system. We get the information and the idea about how to port Snort to the Android platform in this paper.

And in 2009, Batyuk et al. proposed a paper that describes the efficiency of moving the computing-intensive tasks to the native layer of Android [15]. The moving of the computing-intensive tasks will speed up 10 times to 30 times to do the computing-intensive task on the native layer instead of executing it on the JAVA layer. The method described is to use the Android native development toolkit (Android NDK), and call the function in JAVA virtual machine, in order to move the computing task to the native layer as the library. The application in JAVA layer only gives the arguments to the functions and receives the result from the functions. But we think that the disadvantage of the method is that the time of rewriting the program to Android will be massive.

3 Construct the Building Environment

The whole porting task can be divided into three main procedures.

1. Rooting the target Android system to get the super user authority.
2. Cross-compiling Snort to build the execution file that can be run on the Android system.
3. Implementing the front end GUI that is used to invoke the Snort execution file.

Since the packet capturing will need to directly control the network interface, so we need the root authority on Android. Otherwise, the security mechanism on Android will block the execution of Snort.

In this section, we introduce an application which is called CodeSourcery. We use CodeSourcery to build up the execution file. It is a compiler tool-chain that helps us do the cross-compilation.

3.1 Cross-Compiler Installation

We have installed the cross-compiler on Ubuntu 8.10, and we used the Sourcery G++ lite as our cross-compiler tool-chain, The compressed file can be downloaded from [4], double clicking on the file to uncompress and install it. After installation, we set the CodeSourcery directory path as the pre-defined instruction search path for convenience. The software and the hardware editions we used are summarized in Table 1.

Operating system on host machine	Ubuntu 8.10
Cell phone hardware	HTC Magic 32A
Operating system on cell phone	CyanogenMod
Cross-compiler	Sourcery G++ Lite
Back up ROM	Nandroid

Table 1: The edition of hardware and software we used

3.2 Rooting Android

There are many ways to get the root authority on Android, and there are many users who have made their own rooted Android image. To get the root authority, we overwrite one of the users' rooted Android image on our cell phone. The rooting procedure can be done on both windows and Linux. We choose the rooted Android image which is made by a hacker called Cyanogen. The reason we choose the CyanogenMod image is, in this image there are not many applications in it, since we maybe need more memory and storage spaces to ensure that the Snort can properly put into the system. After rooting procedure, we can access the */data* directory in the Android file system. The cell phone we use is the HTC magic (You can also call it G2 phone or sapphire which is a more common name on the internet), since the G2 phone have the most information about rooting, we can get all the information we want on the internet very quickly if we have the problem in every procedure.

3.2.1 Backing up the system

The rooting procedure may fail. Therefore, we backup the data and the system setting before we proceed the rooting procedure, so that we can recover the system if we fail to root the Android.

1. First we boot the cell phone in the fastboot mode by pushing the power and the return button together. When the cell phone is booted in the fast-boot mode, the screen displays three androids on a skating board.

2. Second, we change the directory to the folder where the fastboot instruction is placed and key in the following instruction. The recovery image is placed in the same folder.

```
$./fastboot boot recovery-RA-magic-v1.2.x.img
```

3. When the booting sequence is finished, a menu list would be displayed, We choose the Nandroid v2.x Backup to back up the system. When the back up procedure is finished, the SDcard would contain back up files. If the rooting action failed, we can use the menu list to recover the system by choosing the recover option.

3.2.2 Rooting the system

We can use the recovery image above or by the adb instructions to overwrite the image into the read-only memory of the Android.

We enter the fastboot mode again and choose "apply update.zip" option, and sequentially flash the HTCP_ADP_1.6_DRC83_rooted_base.zip image and update-cm-4.2.4-signed.zip, bc-4.xxxx.zip. We enter the main screen of the operating system after about 10 minutes, because the system need to be reloaded into the memory of the cell phone.

Another way to flash the image is to use the adb shell. Here are steps. First the image must be stored in the SDcard. A suggested instruction is as follows:

```
$adb push boot.img /sdcard/boot-new.img
```

Second, we use the flash_image instruction to flash the system image.

```
$adb shell flash_image boot /sdcard/boot-new.img
```

After the system.img, recovery.img and boot.img are flashed, the rooting procedure is completed.

Sometimes you only need to flash the system.img and then reboot if you do not want to update automatically, and in this way the rooting action will be a little faster. We also supply the rooting ROM for convenience, so we only need to use the recovery-RA-magic.img above to complete the rooting action.

4 Cross Compiling Snort On Android

Here we progressively do the cross-compilation. We cross-compile the BusyBox to ensure that our cross-compiler can work correctly. Then we cross-compile the Tcpdump to check the libpcap can work on Android and the compatibility to Snort. Alternatively, if we start the cross-compilation on Snort first, it will be hard to define where the compilation problem is.

Through our testing, we use libpcap 2.x and libpcre 8.0 to constitute the Snort. Both of the editions are the newest. This is good for upgrading of Snort in the future, since the newer version of library means that we can have more compatibility.

In the following, we have two environments, Ubuntu terminal and the adb shell, We use $ and # to indicate them, repectively.

Figure 6 displays the compilation procedure of a program. We use the cc arguments to specify the compiler we used to compile the source file, and use the ar arguments to specify the tool we want to use to make the static library and then use the linker to link the object file and the static library. We will get the execution file. It is called the a.out by default.

4.1 Cross Compiling Busybox

We compile the Busybox to check if the cross-compiler can work properly. Our steps are as follows, we change the folder to the BusyBox, and execute the following instruction.

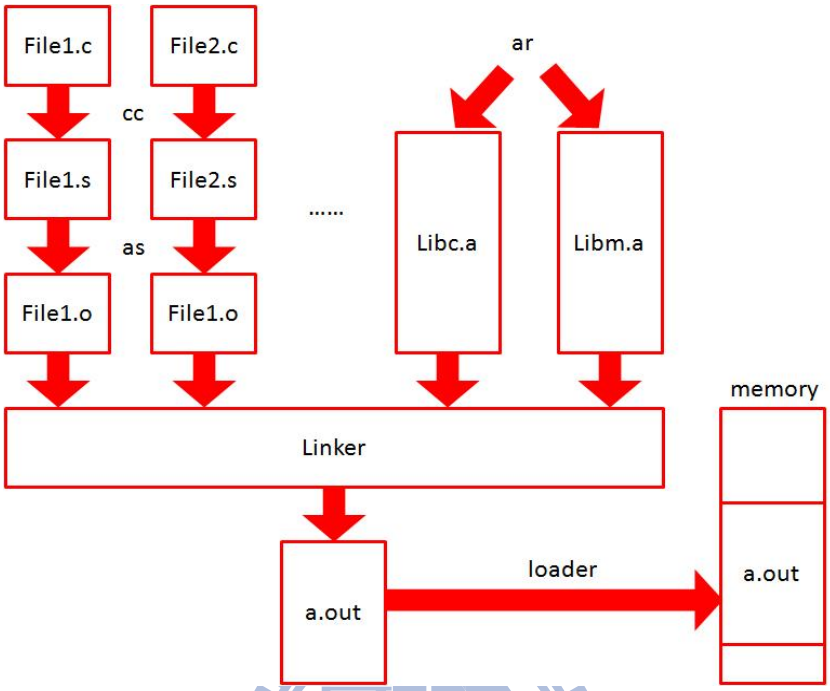


Figure 6: The compiling procedure

```
$make menuconfig
```

The Busybox will enter the configuration menu. We choose the building options in BusyBox setting, and key in the path where we put the cross-compiler in the Cross Compiler prefix column.

```
.../Sourcery_G++_Lite/bin/armnonelinuxgnueabi-
```

We save the setting and exit the configuration menu. To apply the new setting, we make the file again.

```
$make
```

```
$make install
```

There are two ways to verify the result. One is to execute the execution file, the other is to see the file information with the instruction "file" from Linux. If the execution file cannot run on our building computer or it is statically compiled, then our experiences shows that it can run on the Android platform.

We push the execution file of Busybox into the Android file system with the following instructions.

```
$adb push BusyBox /data
```

```
$adb shell
```

```
#cd /data
```

Now we can execute the BusyBox under the */data* directory.

4.2 Cross Compiling tcpdump

In order to check whether we have correctly compiled the static library of libpcap, we compile tcpdump on Android. If the tcpdump can not run on Android, we can narrow down the problem to the library. If the tcpdump can run on Android, we get the right static-version of libpcap. We use the following configuration to configure the makefile and then make it.

```
$CC=arm-none-linux-gnueabi-gcc
./configure --host=arm-linux --prefix=/TARGET LDFLAGS="-static"
```

```
$make
```

```
$make install
```

For the compiling arguments, we compile the .c file into the .o file with CC, and we specify the host machine is arm-Linux, and set the LDFLAGS to be "-static" to make it statically-compiled.

After the libpcap.a is statically-compiled, we can use the following configuration to compile the tcpdump file. We do not need to specify the path of libpcap.a in the configuration of tcpdump. After the configuration, we use make instruction to compile the static-compiled version of tcpdump.

```
./configure --host=arm-linux --prefix=/TARGET LDFLAGS="-static"
```

```
$make
```

4.3 Cross Compiling Snort

Finally, we can start the cross-compilation on Snort. We statically compile the two libraries: libpcap and libpcre. These two libraries will be used in the compilation of Snort.

4.3.1 Libpcap

We have done the cross-compilation on libpcap in Section 4.2. Here we summarize the configuration and instructions.

```
$CC=arm-none-linux-gnueabi-gcc
./configure --host=arm-linux --prefix=/TARGET LDFLAGS="-static"
```

$make

$make install

4.3.2 Libpcre

Here we set the CXX argument for the libpcre, the CXX argument is to notify the compiler with which CXX compiler we want to use.

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

$make

$make install

The two libraries will be installed into the directory specified in the –prefix argument. We can compile the Snort with the same prefix setting. By this way, the arguments will be shorter.

4.3.3 Snort

Since we have to link two libraries statically, so the configuration requires two more arguments, –with-libpap and –with=libpcre. With the argument we just specified above as –prefix=/TARGET.

```
$CC=arm-none-linux-gnueabi-gcc AR=arm-none-linux-gnueabi-ar
RANLIB=arm-none-linux-gnueabi-ranlib
./configure --host=arm-linux --disable-shared --prefix=/TARGET
LDFLAGS="-static" CXX=arm-none-linux-gnueabi-g++
--with-libpcap-libraries=/TARGET /lib/
--with-libpcre-libraries=/TARGET /lib/
```

$make

$make install

Due to the makefile problem, we use Snort 1.7 to do the cross-compilation. The haderfile of pcap-bpf.h has renaming problem. We use the following instruction to link the name of the header file to solve the renaming problem. As a result, the compiler can find the header file in the compiling process.

```
\$ln -sf /usr/loca/include/pcap-bpf.h /usr/local/include/net/bpf.h
```

The whole cross-compilation procedures are completed. We can use the same instructions to execute the Snort through adb shell instructions. We use the following instructions to observe the packets flow of the Android.

$adb push Snort /data

$adb shell

#cd /data

Figure 7: The result of executing Snort on computer

#./Snort -v

Our sample result is shown in Figure 7.

5 Front End Application Design

5.1 Architecture

The statically-compiled version of Snort is already an executable program. We can execute it right after we place it into the file system of Android. However, the default interface of the cell phone does not support direct access to the file system. Thus, a user can not execute Snort by default. We provide a front end to solve this problem. Our front end is an interface for invoking the Snort from the cell phone. The front end application will execute Snort and set the arguments of Snort, and display the result or the record on the screen of the cell phone. The size of the front end is about 30KB, this size will not grows rapidly even when we add more features to it in the future, since the main functionalities are all done in the native layer.

The architecture of the front end application consists of three parts. It is shown in Figure 8.

1. Snort_frontend.java: The program entry of the front end application, and this file defines the graphic user interface (GUI) and the function of the units such as buttons and menu list.

2. Api.java: The main function used to invoke Snort, and with many pre-setting arguments that can execute Snort in different modes.

3. Openfile.java: The functions that we used to open the result file in the file system.

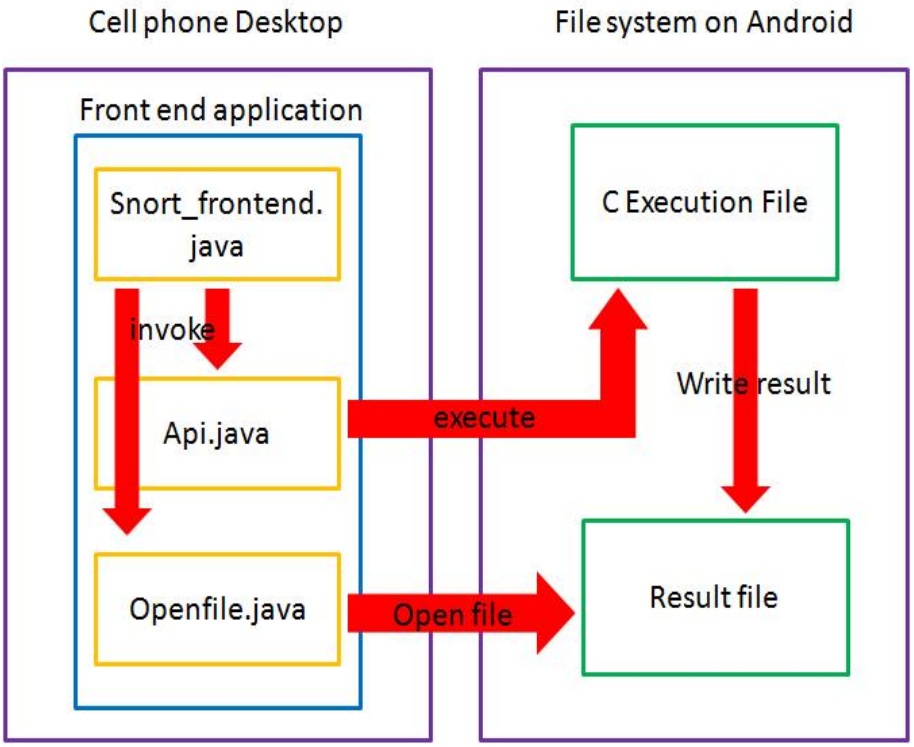


Figure 8: The architecture of front end application

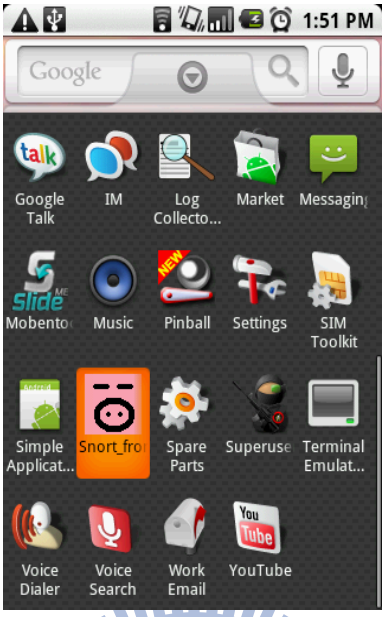


Figure 9: The Snort icon in the main menu

Here we reference some source codes in the droidwall open project in the google project platform [8], which is an open project for the front end application of iptables on Android.

Figure 9 displays a screenshot of the front end application. It shows the icon of the Snort front end application in the main menu, we use the pig to be the icon as the Snort's mascot.

Figure 10 is the main screen that will be displayed when we click the icon. When the first button is clicked, we will invoke the openfile.java to turn to the activity that displays the alert file. In Android, we often call a screen as an "activity". The second and the third button are the buttons that will execute Snort with different configurations, one is for the sniffing mode, and the other is for the execution with the configuration file.

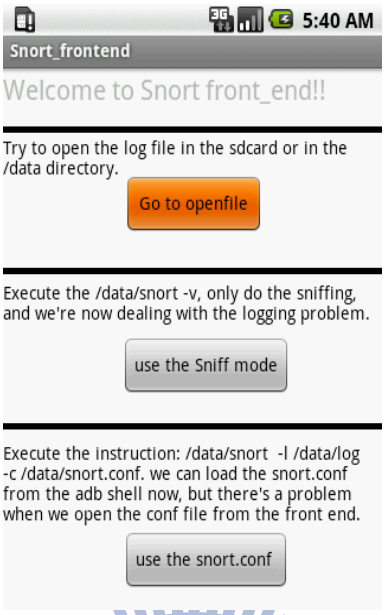


Figure 10: The main screen of front end application

In Figure 11, we click the button "use the snort.conf", then Snort will execute with the snort.conf with the signatures in it.

Figure 12 is the menu list of the application. When the START button is clicked, the application will be invoked to start the service of Snort, and the second button can stop the service of the snort.

Figure 13 shows the result of packet sniffing, we can see the initialization of Snort, and the network interface we used. We can also see the content of the packet, including the length of the packets, the binary content of the packets, and the readable content of the packets in Figure 14. Figure 15 is the activity that displays the alert file.

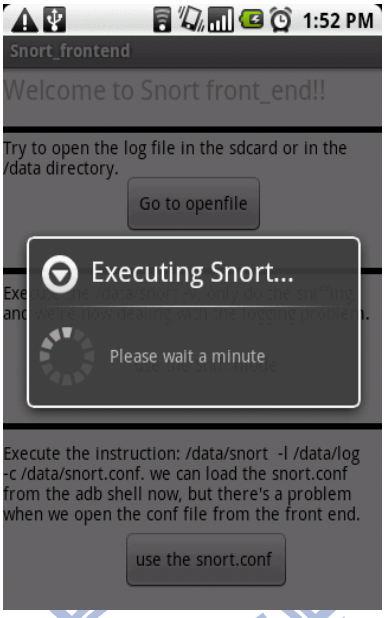


Figure 11: Executing Snort

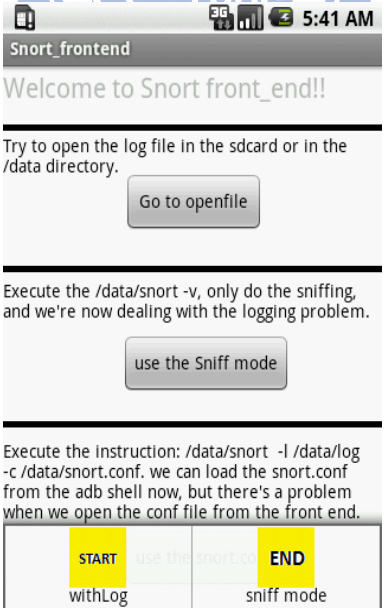


Figure 12: The menu list of the front end application

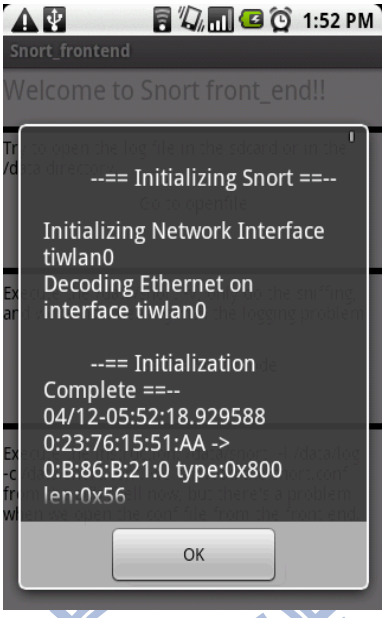


Figure 13: The result of packet capturing

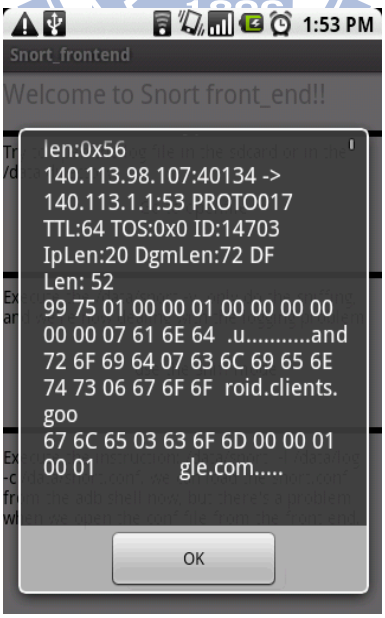


Figure 14: The result of packet capturing II

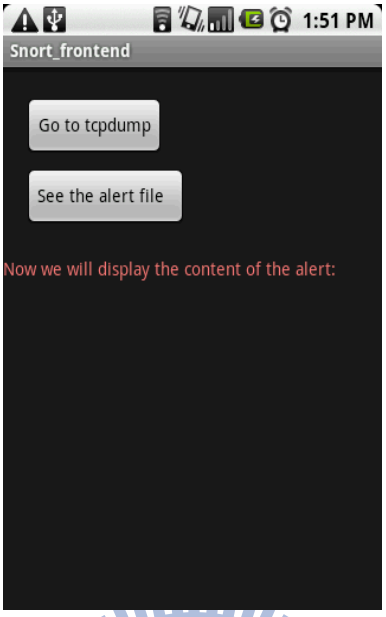


Figure 15: Alert file of Snort

Snort is now executing. We can now press the home button to jump to other works. As Figure 16 shows.

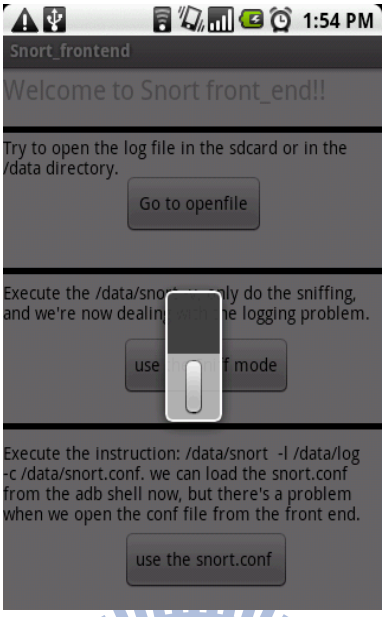


Figure 16: Snort is executing in background

6 Functionality Testing

After porting Snort to Android, we need to test if Snort can properly execute the tasks we need. Through our testing, we found that Snort can correctly read the signatures in the SDcard and record the packets into the cell phone storage, and we can also do the same thing through the front end graphic user interface now.

6.1 Packet Capturing

We tested the functionality of packet capturing, the first part is to test if Snort will drop the packets when we are surfing on the internet and downloading applications. We run the Snort in the background, and keep processing the action of watching web pages and installing applications at the same time, after 5 minutes, we get the first result. The result shows that we have

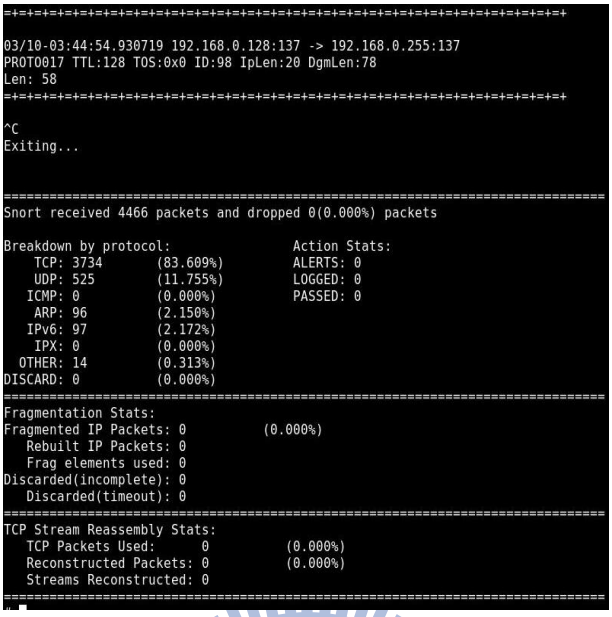


Figure 17: Packet capturing when downloading and surfing on Internet

received 4466 packets in 5 minutes, with 3734 packets of TCP protocol and 525 packets of UDP. Most important of all, 0 packets dropped. Thus by this experiment we can conclude that Snort will not drop the packets in the normal use like surfing or downloading the applications from the Android Market.

In the second experiment, we increase the packets flow. We keep watching the video on the internet for 6 minutes, and we totally receive 53234 packets of TCP. On average, we receive 148 packets per second of TCP. In this experiment, Snort dropped only 3% packets(1527 packets). Because the real-time streaming is one of the heaviest tasks for the network usage, we can conclude that Snort can protect the cell phone even when we receive a large scale of packet flow.

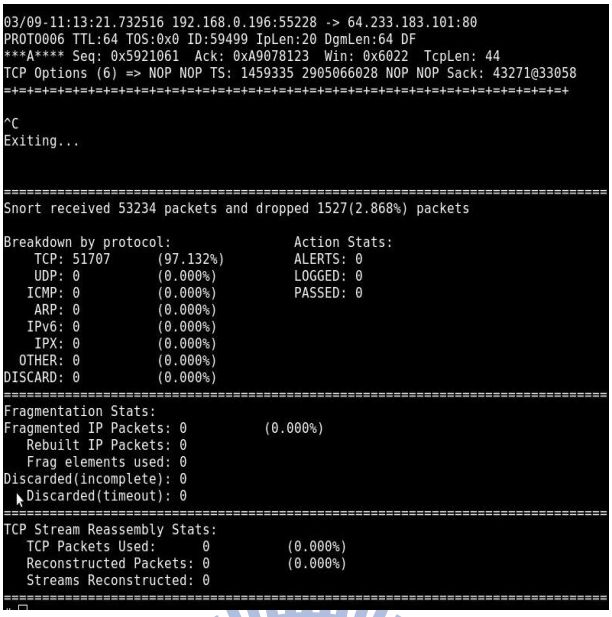


Figure 18: Packet capturing when online real-time streaming

Through these two experiments, we can conclude that the capability of packet capturing of Snort on Android is fairly good, The packet capturing functionality is fundamental in Snort, we are planning to design more experiments to test the capabilities of Snort.

Figure 19 exhibits the alerts generated by snort. The file of snort.conf determines when an alert should be generated. In our experiment, we use one signature to define this alert. The only signature we used in snort.conf is displayed in the following.

```
alert tcp any any - > any any (msg:"phishing site";content:"yahoo";nocase;)
```

Other signatures provided by us can be found in appendix.

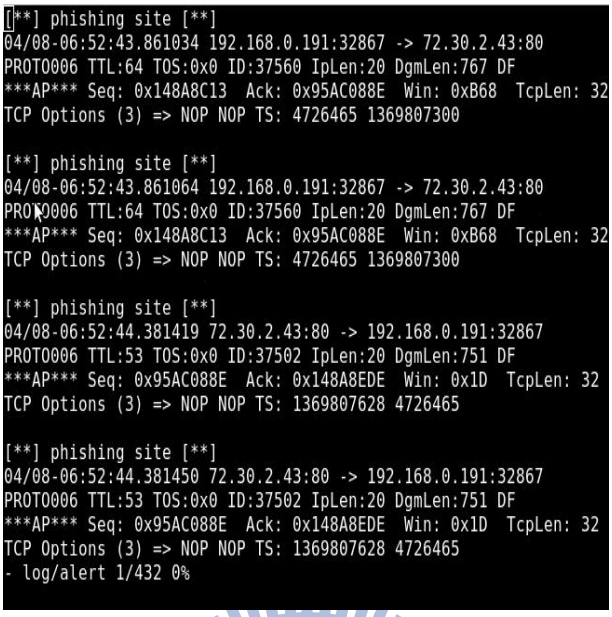


Figure 19: Alert file

6.2 Power Consumption

Since the battery power is very limited to the cell phone. We have also done some power consumption testings to see how much power consumption Snort will make.

The first experiment is to test the power consumption with or without executing Snort in the background when the cell phone entering the energy-saving mode. We record the power consumption of the cell phone when it is in the energy-saving mode in one hour. One time with Snort executed, and the other time without Snort executed in the background. The result of the power consumption is shown in Table 2 with the Battery lasting time.

The second experiment is testing the power consumption when watching the video over the internet. We compare the consumption of watching video

	Time	Power Consumption	Battery lasting time
Without Snort	60 mins	2.8 %	35.7 hours
With Snort	60 mins	4.6 %	21.7 hours

Table 2: Power consumption when entering energy saving mode

	Time	Power Consumption	Battery lasting time
Without Snort	60 mins	7.2 %	13.9 hours
With Snort	60 mins	9.6 %	10.4 hours

Table 3: Power consumption when doing online streaming

and running Snort in the background with the consumption of watching video only. We can see the result in Table 3.

6.3 Memory Consumption

The number of signatures will affect the memory consumption of Snort. The more signatures, the more memory will be consumed. So we need to know how much memory will a signature consume.

First, we test the memory consumption of Snort with only one signature in the snort.conf file. Bacause we know that Snort consists of the C execution file part and the JAVA front end application part, we have to add these two memory consumptions together to get the result which is the exact memory consumption of Snort. We found that C execution file part of Snort consumed 3352KBs. The JAVA front end application part consumed 15592KBs. The total memory consumption is 18944KBs. We pick up an application: alarmclock as the comparing application, the Snort totally consumes 18944KBs and the alarmclock consumes 17920KBs, Snort only uses additional 1024KBs of memory. We can conclude that Snort doesn't con-

Memory use	JAVA part	C part	total
Snort	15592KB	3352KB	18944KB
Alarmclock	17920KB		17920KB
Difference			1024KB

Table 4: The comparison of memory use between Snort and Alarmclock

Number of signature	Memory use of C part	Memory used per signature
1	3352KB	
100	4004KB	6.58KB
200	4532KB	5.92KB

Table 5: Relationship between memory consumption and the number of signature used

sume much more memory than the alarmclock. The observation is shown in Table 4.

Second, we test the memory consumption of Snort with 200 signatures, we found that the C execution file part of Snort consume 4532KBs, so we know that one signature will consume 6KBs. In other words, we can use about 2816 signatures in the snort.conf file, and the total memory consumption of Snort will be equal to the memory consumption of two alarmclock applications. We summarize the result in Table 5. However, there are not that many signatures we can get for Android at this time. The number of signatures we have now is 250 and it is far less than 2816. The more discussions about signature will be given in section 6.4.

6.4 Signature Testing

Since cell phones are not like desktops, the services on cell phones are still less than the services on desktops. Many applications only exist on desktops.

Hence, some corresponding signatures are useless for the use of Snort on Android, we only keep the signatures that corresponds to the applications which have services on cell phones. We make an experiment to ensure that the signatures we used can properly detect the malicious behaviors on the cell phone.

Through our estimations and observations, we now have only 250 signatures that fit to the usage of Android. By the result of the memory testing, we can have about 2816 signatures in the snort.conf and the memory usage of Snort will be equal to two alarmclock applications, so we still have plenty of space to strengthen the security of our cell phones.

We will provide a snort.conf with some fixed signatures that can properly detect the malicious behaviors, and we will keep fixing more signatures for the use of detections on cell phones.

7 Discussions and Conclusions

We have completed the Snort 1.7 porting, Snort uses signatures to detect abnormal events. However, the signatures fo Snort are all designed for the attacks on PCs. Signatures for attacks to Android are required in order to enable the true value of Snort.

In addition, the number of signatures may affect the power consumption. In our experiment we only use one signature and the power consumptionis about 2% of the total battery power per hour. We sniff all traffic by the signature. When the signature defines a rarely event, the power consumption may be much lower.

For higher edition of Snort, the statical compilation fails.We specify the compile option of LDFLAGS to be "-static", we still get the dynamic-linking version of Snort, and we are now still trying to solve this problem.

We introduce the method that shows how to port Snort on Android, and we use its powerful functionalities to provide the security protection on Android. We can scan all the packets to determine whether there are malicious contents in the packets and monitor our cell phone packets flow. Many users use Android cell phones as wireless access points to their laptops, Snort can protect the laptops by monitoring the packets flow through the Android cell phones. Snort is very useful in this scenario.

In our implementation, the front end application can be easily maintained, and the upgrading can be divided into three parts.

1. When upgrading Snort, we only need to choose the newest version of Snort to do the porting in order to upgrade the functionalities of Snort.

2. If the Snort has the new modes to be executed in, we then have to rewrite the front end application.

3. When updating the signature used by Snort, we only need to update the snort.conf file.

In our real-world experiments, we have showed the power consumption of Snort does not cause high-overhead to the cell phone battery. When the cell phone enters the energy-saving mode, the Snort will only consume 1.8% of the total battery power per hour. When Snort handles video streaming data, it only spends 2.4% of total power in 20 minutes. The experiments also show that the base power-consumption of Snort could not be lower than 2% per hour even when the cell phone is in the energy-saving mode. We conclude this problem is caused by the busy-waiting of Snort. We will further study the source code of Snort in order to solve this problem.

In the other hand, memory consumption testing also shows that the Snort didn't bring too much memory overhead. Normally, a java application will consume at least 15000KB memory, so does the front end application of Snort, but Snort will only consume 1000KBs to 3000KBs memory in addition. The number of signatures will also affect the memory consumption. add one more signature to the snort.conf file will make Snort consume 6KB in addition.

In the experiment of system reliability, we open and close 15 applications with Snort executing in the background, and the system remains stable. We can conclude that Snort will not take too much overload to the cell phone.

In the experiment of packet capturing, the better way is to feed the more standard traffic to the Snort and see if the performance of Snort will decrease or not. We can further analyze which kind of attacks will work in the architecture of Android, and which kind will not work.

8 Reference

1. Android Open Source Project (2010/3/11),

   http://source.android.com/

2. Android - An Open Handset Alliance Project (2010/3/11) ,

   http://developer.android.com/guide/basics/what-is-android.html

3. Eclipse Integrated Development Environment (2010/3/11),

   http://www.eclipse.org/

4. Codesourcery (2010/3/11),

   http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite

5. Android Market (2010/3/11),

   http://www.android.com/market/

6. Android resource forum (2010/3/11)

   http://android.cool3c.com/

7. Android internals (2010/3/11),

   http://groups.google.com/group/android-internals?pli=1

8. Droid wall (2010/3/11),

   http://code.google.com/p/droidwall/

9. Snort (2010/3/11),

   http://www.snort.org/

10. XDA developers (2010/3/11)

http://forum.xda-developers.com/index.php

11. INSECURE.ORG, "Top 100 network security tools,"2006.[Online]

.Available: (2010/3/11)

http://sectools.org/

12. CyanogenMod (2010/3/11)

http://www.cyanogenmod.com/

13. CyanogenModWiki (2010/3/11)

http://wiki.cyanogenmod.com/index.php/Full_Update_Guide_-_G1/Dream/Magic32A_Firmware_to_CyanogenMod

14. System and Internet Infrastructure Security Lab: Understanding Android's Security Framework.

15. Leonid Batyuk, Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Ahmet Ahmet Camtepe, Sahin Albayrak: Developing and Benchmarking Native Linux Applications on Android. MOBILWARE 2009:381-392

16. A.-D. Schmidt, H.-G. Schmidt, J. Clausen, A. Camtepe, and S. Albayrak: Enhancing Security of Linux-based Android Devices. In: Proceedings of 15th International Linux Kongress. Lehman Verlag, Hamburg (2008)

17. J. Cheng, S. H. Y. Wong, H. Yang, and S. Lu, "Smartsiren: virus detection and alert for smartphones, "in International Conference on

Mobile Systems, Applications, and Services (Mobisys2007), 2007, pp. 258-271.

18. D. Samfat and R. Molva, "IDAMN: An Intrusion Detection Architecture for Mobile Networks, "IEEE Journal on Selected Areas in Communications, vol. 15, no. 7, pp. 1373-1380, Sep. 1997.

19. M. Miettinen, P. Halonen, and K. Kimmo Hätönen, "Host-Based Intrusion Detection for Advanced Mobile Devices," in AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 2 (AINA'06). Washington, DC, USA: IEEE Computer Society, 2006, pp. 72-76.

20. A.-D. Schmidt, F. Peters, F. Lamour, and S. Albayrak, "Monitoring smartphones for anomaly detection," in MOBILWARE 2008, International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications, Innsbruck, Austria, 2008.

21. Dorothy E. Denning: An Intrusion-Detection Model. IEEE Trans. Software Eng. (TSE) 13(2):222-232 (1987)

22. Uwe Aickelin, Jamie Twycross, Thomas Hesketh-Roberts: Rule Generalisation in Intrusion Detection Systems using Snort CoRR abs/0803.2973 (2008)

23. Alok Tongaonkar, Sreenaath Vasudevan, R. Sekar: Fast Packet Classification for Snort by Native Compilation of Rules. LISA 2008:159-165

24. Arman Tajbakhsh, Mohammad Rahmati, Abdolreza Mirzaei: Intrusion

detection using fuzzy association rules. Appl. Soft Comput. (ASC) 9(2):462-469 (2009)