Malware Detection in Android by Network Traffic Analysis

Mehedee Zaman*, Tazrian Siddiqui†, Mohammad Rakib Amin‡ and Md. Shohrab Hossain§
Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
Email: *devmhd@gmail.com, †rian.7590@gmail.com, ‡md.rakib.amin@gmail.com, §mshohrabhossain@cse.buet.ac.bd

***Abstract*—A common behavior of mobile malware is transferring sensitive information of the cell phone user to malicious remote servers. In this paper, we describe and demonstrate in full detail, a method for detecting malware based on this behavior. For this, we first create an App-URL table that logs all attempts made by all applications to communicate with remote servers. Each entry in this log preserves the application id and the URI that the application contacted. From this log, with the help of a reliable and comprehensive domain blacklist, we can detect rogue applications that communicate with malicious domains. We further propose a behavioral analysis method using syscall tracing. Our work can be integrated with be behavioral analysis to build an intelligent malware detection model.**

***Keywords*—Android, malware detection, netstat, pcap, ADB, Busybox**

I. Introduction

Smart phones and tablets are the most popular and widely used personal electronics devices today and roughly 70% of these devices run Android [1] operating system. Due to Android's vast user base, open nature, and relatively less restrictions on application distribution system, it has always been an attractive platform for malware. According to a recent report published jointly by Kaspersky Labs and INTERPOL [1], 20% of devices that uses their software were attacked at least once by malware.

Malware is a program which disrupts computer operation, gather sensitive personal and financial information, or gain access to private systems without user's consent. With the ever increasing use of mobile devices, mobile malware pose significant threats for users, because their mobile devices store contacts, bank account numbers, credit / debit numbers, private photos, messages and a lot of other sensitive information that can be leaked.

Given the recent tremendous growth of Android malware, there is a pressing need for effective malware detection methods. Existing detection methods can be classified into two major categories: static (code analysis) and dynamic (runtime/behavioral analysis). The sneakiest malware are almost impossible to detect using static analysis, because they often obfuscate the malicious code using random keys. Some malware download the malicious code at runtime and remove it after execution [2]. In these cases, a code analysis for known malware signature cannot detect the malware.

978-1-4799-8126-7/15/$31.00

These exists a few static and dynamic malware detection methods in the literature. Chandramohan et al. [3] has given a high-level overview of various detection methods. Zhou et al. [2] collected, classified and published a large collection of 1260 Android malware. We used malware samples from their collection to evaluate our detection method. Isohara et al. [4] demonstrated a system-call logging based method.

However, to the best of our knowledge, there exists no work that demonstrates network traffic-based malware detection method in details.

The objective of this paper is to demonstrate a detection method based on network traffic analysis. The method we described will be effective against malware that communicates with known malicious remote servers.

Our network traffic analysis is based on logging the URLs of all remote locations that are contacted by applications for a specific period of time. Given, we have a database of known malicious domains; the applications that contact any of those malicious domains can be flagged as malware.

We described our detection method in a detailed step-by-step manner, mentioning all the necessary tools and techniques used. Also, we briefly explained the purpose behind each step. This paper can be used as a technical guideline by researchers, who are trying to develop network traffic-based malware detection applications.

The rest of the paper is organized as follows. In Section II, the main strategy for malware detection is explained. The details of the steps followed for malware detection is presented in Section III. Section IV gives an overview of our work in progress and the direction we are heading towards. Finally, Section V has the concluding remarks.

II. Detection Strategy

We divided the malware detection procedure in two steps:

At first, we created log of URLs that are contacted by applications for a specific period of time. Then we tried to match each entry (URL) of the log with a list of known malicious domains. If a match is found, the application that contacted the malicious domain is a malware itself or has been affected by one.

A. Creating the App-URL table

App-URL table is a history/log of all attempts made by all applications to communicate with remote servers over HTTP. The table consists of **(url, app)** entries. Each HTTP request maps to a single entry, where **url** is the URL which is contacted, and **app** is the application that originated the HTTP request.

This process is further subdivided into four tasks:

1) Packet dumping: We have recorded all incoming and outgoing network packets to/from the android device for specific duration of time. This creates a packet dump file that contains information of which port number (of the mobile device) is accessing which URL.

2) Netstat Logging: To relate port numbers with applications, we periodically executed *netstat* [5] command throughout the duration of packet dumping and saved the outputs. Netstat gives information of which port number is being used by which application when the command is executed.

3) Extracting necessary information from packet dump: We do not take all packets into consideration. We are only interested in HTTP packets (and only requests, not responses). So we have filtered out all other packets from the packet dump we generated at the first step. We took only three fields from each packet: time, originating port and full request URI. This gives a time-sequenced log of port numbers and URIs that a port tried to connect to.

4) Aggregating packet dump and netstat logs: We have so far obtained two separate mappings: *application vs. port number* from netstat logs, and *port number vs. URL* from packet dump. We aggregate these two maps to create a time-sequenced log of applications and the URLs each application tried to contact (The App-URL table).

B. Matching the URLs with Domain-blacklists

We search the URLs in the App-URL table for known malicious domains. If an application tries to connect to a rogue domain (URL), we flag it as a malware. We can also enrich our blacklist by adding other domains contacted by a flagged application.

These steps are discussed in detail in the following section.

III. Details of Malware Detection steps

Our first step is to create an App-URL table. In this table, each row of the table indicates an attempt to make an HTTP connection by any application. We store the time, the application's unique identifier (package name), and the URL which was contacted.

A. Creating the App-URL table

1) Packet dumping: We need to use a software for recording all incoming or outgoing traffic (packets) of the android device. This can be done using *Wireshark* [6] in a computer which is connected to the same local network of the android device.

Alternatively, we can use a similar application in the mobile device. We have used *Shark for Root* [7] for this purpose. A rooted device is not required for this step. Non-rooted devices can use other applications, such as *tPacketCapture*, which captures packets by creating a VPN and directing all traffic through the VPN. We captured packets for a specific amount of time. This step produces a packet dump (*.pcap*) file.

2) Netstat Logging: The packet dump does not directly detect which packet is originated from/destined for which mobile application. The system differentiates packets of different applications by port numbers (source port for outgoing packets or destination port for incoming packets). Hence, we need to know which ports were being used by which applications when the packet was captured. We used the UNIX tool *netstat* [5] to get the mapping between applications and port numbers at a specific time.

Since the packets are recorded for some duration of time and netstat gives the *port number vs. application* mapping for an instance of time (just when the command is executed), a single netstat output will not suffice. Therefore, we executed netstat periodically, while the packets were being recorded.

We used *ADB* [8] to communicate with the android device. To access the interactive shell of the device, *adb shell* was used. In our experiment, we connected the android device with a UNIX computer. Then we executed the shell script shown in Fig. 1 in the computer.

```
for i in {1..100}
do
     adb shell "
     su -c 'busybox netstat -pnt | grep tcp'
     " > netstat
     adb shell "date +%s" > netdump$i
     awk '{print $4 ":" $7}' netstat > netstattemp
     awk -F":" '{print $5 " " $6}' netstattemp>>netdump$i
     echo finished: $i
     sleep 1
done
```

Fig. 1. Shell script used for netstat logging.

This script calls netstat 100 times, with 1 second interval in between. It filters just the necessary information (port numbers and corresponding pid/package names) from each netstat output, and saves them in separate files, along with the timestamp when the dump was taken. So after executing this script, we had 100 files (namely `netdump1, netdump2, ... netdump100`). A single netdump file is shown in Fig. 2.

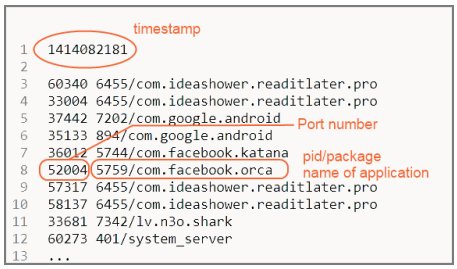

Fig. 2. A single netdump file

This step requires a rooted android device. Because, being a stripped down variant of linux, Android does not come with the *netstat* executable by default. So we used *Busybox*, a tool that allows execution of all standard UNIX commands in android. Busybox cannot be installed without super user permissions.

3) Extracting necessary information from packet dump: Packet dump (.pcap) contains comprehensive meta information about all packets, along with their contents. However, we are only interested in HTTP packets and only three fields of each packet. Pcap filtering can be accomplished by many different ways among which we used Wireshark.

We opened the pcap file in Wireshark. Then the following display filter was applied on the dump:

```
http && ip.src == X.X.X.X
```

Here, `X.X.X.X` is the IP address of the device. This was used to filter out the http responses. For now, we are only interested in requests.
We kept only the following columns in Wireshark:

- Time (in Seconds since epoch format)
- Src Port
- Full Request URI

Then we exported the displayed packets summary in a plain text file. In our experiment, we named the file `filtered.txt` (shown in Fig. 3).

4) Aggregating packet dump and netstat logs: Before this step, we had 100 files containing netstat outputs (*port-application* mapping at specific times). And we had a file filtered.txt, which contains the *port-URL* mapping for all HTTP request packets. We have written a script which processes all these files to produce the final App-URL table.

Since netdump files contains *port-app* mappings for specific moments (1 second apart), a packet's time will not necessarily match exactly with any of these moments. To assign such a packet to an application, we have made some assumptions.

Let t be the timestamp of a packet. Let t_1, t_2, $t_3, ..., t_{100}$ are the timestamps of the netstat outputs (they are stored in corresponding netdump files). Of course $t_1 < t_1 < t_3 < ... < t_{100}$. If $t < t_1$ or $t > t_{100}$, we discard the packet. We only consider packets with t such that $t_1 \leq t \leq t_{100}$.

Now for each of these packets, there is an i such that $t_i \leq t$ and $t_{i+1} > t$. We assign a packet to an application using the following rules:

1) If the same application A was using the packet's port at both t_i and t_{i+1}, then application A is the sender of the packet.
2) If application A was using the port at t_i, and the port was not in use at t_{i+1}, application A originated the packet.
3) If the port was not in use at t_i, and application A was holding it at t_{i+1}, application A originated the packet.
4) If the port was being used by application A at t_i and application B at t_{i+1} then,
if $t - t_i \leq t_{i+1} - t$, application A originated the packet. Otherwise application B originated it.
5) If no application was using the port at either t_i or t_{i+1}, We discard the packet.

Case 5 indicates that after t_i, some application opened the port, sent some packet(s) and then released the port before t_{i+1}. So this packet has gone untraced. We can lessen the frequency of such occurrences by decreasing the interval between t_i and t_{i+1}.

So for every packet (except the ones of case 5), we know the app which originated it. And `filtered.txt` contains Full request URI of all packets. So we now know the URL specified in the packet was contacted by this application. We have logged these (Application, URL) entries for each packet and the App-URL table is ready. A sample table is shown in Fig. 4 .

B. Matching the URLs with Domain-blacklists

When the App-URL table is ready, the table can be sent to a central server. The server can search the table for already known malicious domains, and notify the android device of any rogue application which might be trying to connect to a blacklisted domain. The server can also enhance its blacklist by adding new domains that are contacted by a malicious application.

	Timestamp	Port #	URL
1	1414082186.261850	57001	http://www.quora.com/api/do_action_POST
2	1414082186.531015	47612	http://www.quora.com/
3	1414082187.769571	47614	http://qsc.is.quoracdn.net/-28ce1f6c6095d6c5.css
4	1414082187.770059	47615	http://qsc.is.quoracdn.net/-aeeaea065aef57c7.js
5	1414082192.439645	47621	http://qph.is.quoracdn.net/main-thumb-t-4052-50-khhbtngfzevs...
6	1414082240.246866	45830	http://api.duolingo.com/api/1/version_info
7	1414082240.286386	54574	http://api.duolingo.com/api/1/store/get_inventory
8	1414082240.287393	55690	http://api.duolingo.com/api/1/store/get_inventory
9	1414082277.182687	47634	http://www.memrise.com/api/auth/facebook/
10	1414082279.105752	47635	http://www.memrise.com/api/app/settings/
11	1414082279.671243	47636	http://www.memrise.com/api/level/get/?with_content=true&lev...
12	1414082280.704813	47637	http://www.memrise.com/api/user/courses_learning/?user%5Fid...
13	1414082284.491800	47275	http://static.memrise.com/uploads/things/audio/14218347_136...
14	1414082284.491922	47276	http://static.memrise.com/uploads/things/audio/14218346_136...
15	1414082298.626474	54348	http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js
16	1414082302.333963	39488	http://data.flurry.com/aap.do
17	...		

Fig. 3. Extracted information from the packet dump in `filtered.txt` file

	Timestamp	Port#	App identifier	URL
1	1.41408219400620 4E9	52791	com.quora.android	http://www.quora.com/ajax/action_log_POST
2	1.41408219571637 9E9	42998	com.quora.android	http://www.quora.com/webnode2/server_call_POST
3	1.41408219660355 5E9	47619	com.quora.android	http://qph.is.quoracdn.net/main-thumb-9715372-5...
4	1.41408220127988 6E9	52225	com.quora.android	http://www.quora.com/webnode2/server_call_POST
5	1.41408224024686 6E9	45830	com.duolingo	http://api.duolingo.com/api/1/version_info
6	1.41408224028638 6E9	54574	com.duolingo	http://api.duolingo.com/api/1/store/get_invento...
7	1.41408225598758 8E9	45830	com.duolingo	http://api.duolingo.com/api/1/users/show?userna...
8	1.41408225645597 2E9	59259	com.duolingo	http://api.duolingo.com/api/1/store/get_invento...
9	1.41408226980228 6E9	39860	com.memrise.android	http://data.flurry.com/aap.do
10	...			

Fig. 4. Final output: App vs. URL table

We analyzed two known malwares using this method: *DroidKungFu* and *AnserverBot*, both known for contacting remote C&C servers [2]. Within minutes of installation DroidKungFu accessed *www.waps.cn*, which was listed as a malicious domain in *virustotal.com*. Anserverbot did not contact any blacklisted domain within the first 10 minutes when we recorded packets. The reason might be using an unreliable and freely available domain-blacklist from the internet. Or worse, maybe it communicates over protocol(s) other than HTTP.

IV. Work in progress

We are trying to develop additional behavioral analysis methods which will rely on behaviors other than network traffic. For example, if we can log all system calls made by an application, we can use it on known malware to find patterns in sequence of system calls. These signatures can be used to detect new applications infected by known malware.

Our final objective is to propose an intelligent detection model, which will analyze multiple behaviors and integrate the results for detecting malicious behavior.

For logging system calls, we have used *strace* [9]. But Busybox is required to run strace in Android.

V. Conclusion

In this paper, we have discussed briefly about different types of malware detection techniques and their effectiveness with specific types of malware. We have thoroughly demonstrated a behavioral detection method for detecting mobile malware that can communicate with blacklisted domains and pass sensitive personal / financial information. We have also discussed our direction of research toward devising an intelligent malware detection model. We hope our demonstration presented in this paper will help researchers develop malware detection applications in order to protect users of mobile devices.

References

[1] Kaspersky Lab and INTERPOL Survey Reports, “Mobile cyber threats.”

[2] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *IEEE Symposium on Security and Privacy*, San Francisco, CA, May 20-23, 2012, pp. 95–109.

[3] M. Chandramohan and H. B. K. Tan, “Detection of mobile malware in the wild,” *IEEE Computer*, vol. 45, no. 9, pp. 65–71, Sep 2012.

[4] T. Isohara, K. Takemori, and A. Kubota, “Kernel-based behavior analysis for android malware detection,” in *Internation Conference on Computational Intelligence and Security*, Hainan, Dec 3-4 2011, pp. 1011–1015.

[5] Netstat command.

[6] Wireshark, *A network protocol analyzer for Unix and Windows.*

[7] Shark for Root, *an android application to capture incoming and outgoing packets*.

[8] Android Debug Bridge.

[9] Strace, *a diagnostic userspace utility for Linux.*