An Open and Automated Android Behavior Monitor in Cloud

CHUN-YING HUANG, SHANG-PIN MA, MING-LUN CHANG, CHIN-HSIANG CHIU, TING-CHUN HUANG
Department of Computer Science and Engineering
National Taiwan Ocean University
TAIWAN
{chuang,albert,10257030,10257023,10257019}@ntou.edu.tw

Abstract

For security and privacy considerations, it is important for Android users to understand the behavior and the risk of an application. Although Google claims that new applications available on the official market have passed their security checks, the open design of the Android system still allows a user to install applications for third-party vendors. Therefore, there is still a demand for users to know more about an unknown application. In this paper, we discussed our experiences on setting up a scalable automated Android behavior monitor using virtualization techniques. Our contribution is two-fold: 1) We design and implement a scalable behavior monitor using both dynamic analysis and static analysis techniques; and 2) Based on parts of the analyzed results, we develop a preliminary filter to distinguish benign and malicious applications. The system is open to the public and we expect that the analyzed results can be fed back to the research community and further stimulate more studies on analyzing malicious Android applications.

Keywords: *Android, Behavior monitor, Classification, Dynamic analysis, Static analysis*

1 Introduction

The success of the Android mobile operating system has attracted a large number of users, developers, and researchers to explore new feasibilities to make a better mobile life. Due to its opened design, a large number of applications can be obtained from various different sources with no charge. However, one serious problem that every user must face is that – given a new application, is that application safe to use? Is an application been compromised by malicious attackers? Does that application steal personal confidential information from a mobile device?

There are many strategies for a user to judge whether an application is suspicious or not. One simplest solution to identify suspicious application is to check the requested permissions of an application before installing it. For example, a stand-alone jigsaw puzzle game should not request permission to send a short message. If the game requests such permission, it could be suspicious. However, permission-based determination is not so reliable. Sometimes it is even difficult to judge whether an application is suspicious by using permission. For example, an application may claim that it has a built-in automatic version check feature so that it requires Internet access. However, a user is infeasible to determine whether the application really uses the Internet to check for updates or it is engaged in some malicious activities such as stealing personal information or acting as a stepping stone. In such a case, the user has to make an in-depth observation so that the real purpose of the application can be revealed.

To solve the above issue, it would be better if there is more information provided to a user before he is going to install a new application. Therefore, we design and implemented an automated Android behavior monitor (ABM) as a cloud service in this paper. With the proposed solution, a user is able to submit an application package to our service and then a comprehensive report is sent back to the user to understand the behavior of the inspected application. Besides regular users, a researcher is also able to retrieve behavior reports for a bunch of benign and malicious applications and then attempts to find out distinguishable behavior and attributes to identify malicious applications. The proposed solution obtains application behavior by using both dynamic and static analysis techniques. For dynamic analysis, it monitors file system access, network access, and system call sequences. For static analysis, it retrieves information such as the request permissions, the required permissions, the built-in API calls, parameters, and control flows. The proposed solution also adopts virtualization techniques so that the service can be scaled out by simply cloning more virtual machines.

In addition to the proposed solution, a preliminary filter is built as an example to classify benign and malicious applications based on the analytical reports generated by the proposed solution. We expect that through an open and automated application behavior monitor service, we are able to stimulate more studies on analyzing malicious applications by providing

collected applications as well as thee the analyzed results to the research community.

The rest of this paper is organized as follows. In Section 2, we discuss related works that focus on analyzing Android applications. In Section 3, we introduce proposed solution including the design objectives and the components. In Section 4, we discuss the preliminary filter that is designed based on the analyzed results. Finally, a concluding remark is given in Section 5.

2 Related Work

A lot of researches have been devoted to analyze the (malicious) behavior on Android applications. Ju et al. [15] wrote a brief introduction on Android malware and discuss possible intentions of attackers. Enck et al. [6] wrote a good introduction on android's security design in 2009. Basically the android operating system provides a coarse-grained mandatory access control (MAC). It is able to enforce how applications access components based on permitted permissions. As a result, each android application must have a list of requested permissions and all these permissions must be granted at the install time. The requested permission list is often declared by an application developer manually. Hence, a number of interesting researches are devoted to review how permissions are declared in applications. Barrera et al. [1] analyzed how developers of android applications use the permissions. They explored and analyzed 1,100 applications using the Self-Organizing Map (SOM) algorithm. They found that although android has a rich set of permissions, only a small number of these permissions are actively used by developers. Felt et al. [7] studied android applications to determine whether android developers follow least privilege with their permission requests. They built a tool and applied it to 940 applications and found that about one-third of evaluated applications are over privileged. They also concluded that developers are trying to follow least privilege but failed due to insufficient API documentation. Johnson et al. [8] developed an architecture that automatically searches for and downloads android applications from android Market. With the application, they created a detailed mapping of android API calls to the required permissions. The idea is similar to [7] but they collected a large number (141,372) of applications to conduct the experiments. They found that the majority of developers are not using the correct permission set. The applications are either over-specify or under-specify their security requirements. Zhou and Jiang [13] systematically characterized 1,260 android malicious applications from various aspects, including their installation methods, activation mechanisms, and the carried malicious payloads. In addition, they also compared the permission requests of the 1,260 malicious applications against another top free 1,260 benign applications on android market. The comparison shows that the top 20 frequently requested permissions are similar for both benign and malicious applications.

In addition to analyze permissions, a number of researches tried to detect malicious application using static analysis or dynamic analysis techniques. These techniques are similar to those used to detect traditional malware on desktop personal computers. Besides many well-known signature-based virus scanners, androguard [5] is an open source project that dedicated to detect android malware. Androguard detect a malicious application or an injected malicious code based on control flow graph. A given application package is first disassembled and each identified method in assembly source codes is converted into a formatted string that represents the control flow graph [4] of the method. A number of predefined malware's control flow graphs are then compared against the obtained control flow graph strings to check if they are similar [10] to malware. Schmidt et al. [12] proposed a static analysis solution to detect malicious application based on the output of the readelf tool, which contains a list of symbols that involved with an executable. They then differentiate malicious applications from benign ones based on the combinations of system calls used in the executable. Burguera et al. [3] proposed to detect malware using dynamic analysis techniques. They developed a client named Crowdroid that is able to monitor Linux kernel system call and report them to a centralized server. Based on the collected dataset, they cluster each dataset using a partition clustering algorithm and hence differentiate between benign and malicious applications. Bläsing et al. [2] proposed AASandbox that performs both static analysis and dynamic analysis on android programs to automatically detect suspicious applications. Static analysis scans the software for malicious patterns at the source code level. Dynamic analysis executes the application in a sandbox which monitors and logs low-level accesses to the system for further analysis. It looks like a complete system but the authors did not show the performance on analyzing malware and therefore we do not how its performance. Due to the lack of malware samples, most existing works conduct experiments using self-made malware or a limited number of real malware. It still requires more evidence to prove the effectiveness of these solutions.

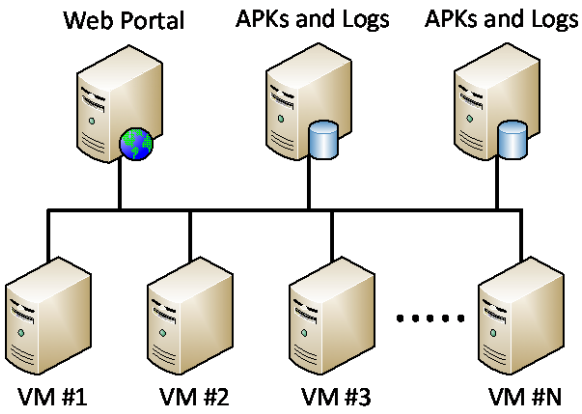


Figure 1 The overall architecture of the proposed solution.

It is of course that the current security model adopted by android could be not the best choice. Researchers also tried to propose alternatives to the existing model. Nauman et al. [9] proposed the Apex framework which allows a user to selectively grant permissions to applications. The framework includes an extended package installer and provided an easy-to-use interface for users to set the runtime constraints (allow, deny, or conditional allow) for each permission. Granting permissions explicitly by users may be not a good idea. Roesner et al. [11] proposed a user-driven access control where permission granting is built into existing user actions in the context of an application, rather than added as an afterthought via manifests or system prompts. Application developers have to modify their codes by embedding access control gadgets (ACGs) into the user interface (UI) of applications. Then, permissions are granted implicitly when a user takes actions via the UI. User-driven access control is able to grant permission in one time, in a session, in a period of time, or permanently depending on the corresponding action. It guarantees the least-privilege security model without losing its flexibility and ease-of-use property. Nevertheless, with existing design, android users now can simply make judgments based on the permissions, which is coarse-grained and still a difficult puzzle.

3 The Proposed Solution

The overall architecture of the proposed solution is shown in Figure 1. Except the web portal, all other components reside in a private network. Therefore, a user is only able to access the service via the web portal. On receipt of an application package (the so-called APK file), the hash value of that APK file is evaluated and compared against existing repository to ensure it is a fresh package. It a submitted package is now new, a previously analyzed report is sent back to the user immediately. On the contrary, if a fresh

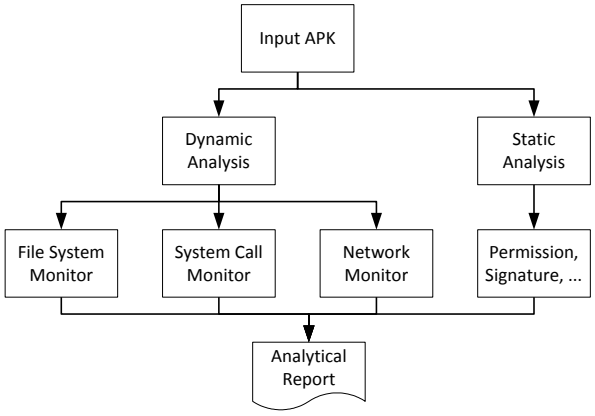


Figure 2 The components of an inspection instance.

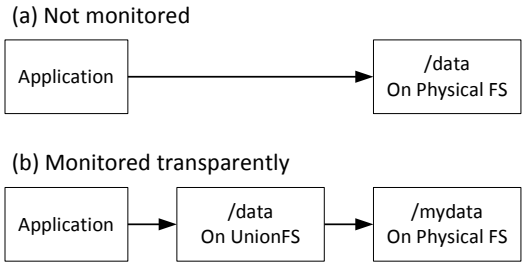


Figure 3 Access a real file system via a transparent intermediate layer.

APK file is received, it is placed into the "APKs and Logs" repository and then scheduled to be inspected later. The inspection of application behavior is done by virtual machines in the pool. Each virtual machine is able to run several concurrent inspection instances to monitor multiple applications at the same time. This would shorten the required time to analyze packages in the submission queue. However, if the number of virtual machines is not sufficient enough to process queued APK files in a reasonable time, more virtual machines can be cloned to share the workloads.

The components of an inspection instance are shown in Figure 2. Each instance has four major components, i.e., the file system monitor, the system call sequence monitor, the network traffic monitor, and the static analyzer. The detail of each component is introduced later in this Section.

3.1 File System Monitor

We develop our own file system monitor to log file access activities during the execution process of an application. As shown in Figure 3, we add a transparent intermediate layer between applications and the physical file system. At the beginning, we choose to use the FUSE (file system in user-space) to implement the intermediate layer. Although it is easier to code and debug, the FUSE sub-system

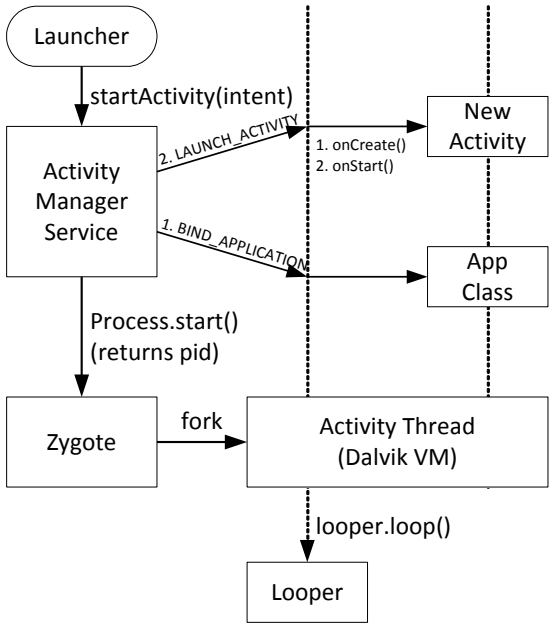


Figure 4 Android's application launch process.

cannot handle access permissions from different users properly. Therefore, we switched from FUSE to UnionFS, which is an in-kernel file system that is able to merge multiple mounts in a single view. To activate either FUSE or UnionFS, we have to re-compile the Linux kernel and modify the start-up scripts so that the monitored file system is mounted at a different location, e.g., from **`/data`** to **`/mydata`**, and then we can mount our transparent intermediate layer to intercept file system operations for a monitored mount point, e.g., **`/data`**. Currently we monitored only for **`/data`** and **`/cache`**, but more mount points can be easily monitored as well.

The Android emulator emulates an ARM-based CPU. Therefore, we use the official development toolkit NDK to conduct cross-compile on x86 Linux hosts. Kernel source codes are retrieved from Google source [14]. Since UnionFS is not part of the standard kernel, we manually merge the UnionFS into the kernel source tree. When the cross-compiling of the Linux kernel is done, we then modify the **`init.rc`** file stored under the root directory. This file defined fundamental start-up scripts and services. For each monitored file system, we alter the default mount point defined in **`init.rc`** and then mount UnionFS at its original path. As a result, we can then intercept and log all interested file system activities.

3.2 System Call Monitor

To implement the system call monitor, we use the open source **`strace`** tool. The **`strace`** tool is a command-line tool that intercepts and records the system calls that called by a process and the signals

```
IP 122.225.x.y.80 > 10.0.2.15.41762: Flags
[S.], seq 164288001, ack 669385195, win 8192,
options [mss 1460], length 0

0x0000: 0000 0001 0006 5254 0012 3502 0000 0800
0x0010: 4508 002c 05ec 0000 4006 d040 7ae1 xxyy
0x0020: 0a00 020f 0050 a322 09ca d601 27e6 01eb
0x0030: 6012 20
```

Figure 5 A sample network flow sent from a malicious application.

that are received by a process. The name of each system call, its arguments, and its return value are printed on standard error or to a specified file. Since the Android OS is a Linux-based operating system, it is not difficult to port the tool onto Android using the NDK. The problem of monitor an Android application is that: How to attach the **`strace`** tool to an Android application process, especially when the process is launched in a virtual machine?

Readers may have launched an Android application in the shell prompt using the **`am`** script. However, if you use the **`strace`** tool to monitor the `am` script, you are not able to monitor any system calls called by the application process. To make a successful system call monitor, we have to understand the process that the Android OS launches an application. As shown in Figure 4, a user may launch an application by clicking the application icon in the launcher or using a shell command such as **`am`** to start an application. The launch request is sent to the activity manager service and then the application process is forked by the **`zygote`** service, which creates a Dalvik virtual machine and then run the Java-based application in the virtual machine. Therefore, the **`strace`** tool should be attached to the **`zygote`** service instead of the launcher or the **`am`** script. There are two possible strategies to achieve the goal. First, a user is able to modify the **`init.rc`** file, which also defines the command line to launch the **`zygote`** service, and append the **`strace`** tool before that command line. Alternatively, a user is able to attach the **`strace`** tool to the **`zygote`** process on demand and then launch the monitored Android application.

We recommend the second strategy because it is much simpler and the output of the **`strace`** tool contains less noise.

3.3 Network Monitor

Implement the network monitor is straightforward. There are two different solutions. If an application is monitored in an Android emulator, the emulator itself has an option to save all the network traffic of the emulator into a single file using the **`pcap`** format,

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

Figure 6 The syntax of the control flog graph string represtation.

```
1: mov X, 4
2: mov Z, 5          →      B
3: add X, Z
4: goto +50          →      G
5: add X, Z          →      B
6: goto -100         →      G
```

Figure 7 An example of generating CFG string from a short piece of code.

which is the same as the well-known **`tcpdump`** tool. In case that an application is not monitored in an emulator, we can still cross-compile the **`tcpdump`** tool and then capture the traffic using that tool. When the monitoring of an application is finished, the captured network traces is then used for further analysis.

A simple screen shot for a captured network flow is shown in Figure 5. It is a request sent by a malicious application. The application connects to a suspicious IP address 122.225.x.y in China using the HTTP protocol. For the ease of understanding the application behavior, we also write several scripts to parse the **`pcap`** file and generate comprehensive reports such as lists of IP addresses, lists of domain names, and data exchanged in unencrypted popular application protocols such as HTTP.

3.4 Static Analysis

Our static analysis implementation majorly relies on the androguard tool. Androguard is a reverse engineering tool for Android packages. It is written in **`python`**. It is able to retrieve meta-information from an APK file as well as disassemble the package into assembly language using Jasmin's (dedexer's) syntax. Some primitive meta-information of an application includes its package ID, API level, activities, services, and requested permissions. It is also to statically

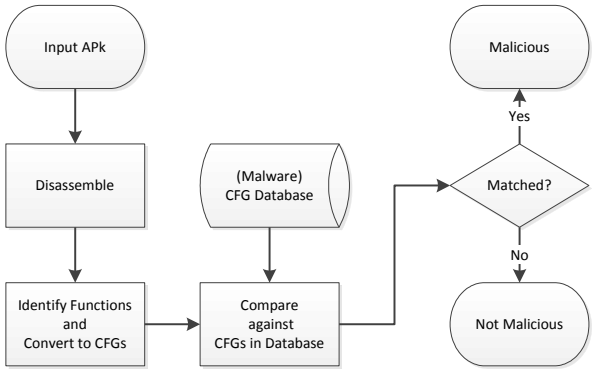


Figure 8 The malicious application detectioin flow of androguard.

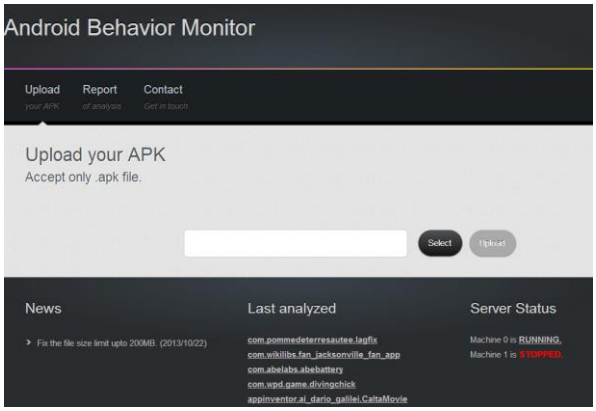


Figure 9 The web portal page of the proposed solution.

analyze the package and retrieve advanced information such as the required (actual) permissions. In addition, androguard has several built-in signatures that are able to detect some existing malicious applications.

One important feature of androguard is to create control flow graph (CFG) [15] of each function. A CFG is a string-represented function structure. The syntax of CFG strings is shown in Figure 6. Therefore, given a short piece of code in Figure 7, it can be converted into a CFG string represented by `BGBG`. Since most android malicious software is repackaged software, androguard detects malicious software by comparing a CFG string against previously collected CFG strings signatures from malicious applications, as shown in Figure 8. The comparison of CFG string signatures is done by using various string similarity algorithms such as Kolmogorov complexity, NCD, NCS, and entropy.

Although androguard is able to effectively detect malicious applications, most of the CFG string signatures used by androguard are generated manually. It would better if the generation of malicious signatures can be done automatically. Therefore, our analytical reports also attempt to generate the CFG string for all analyzed applications

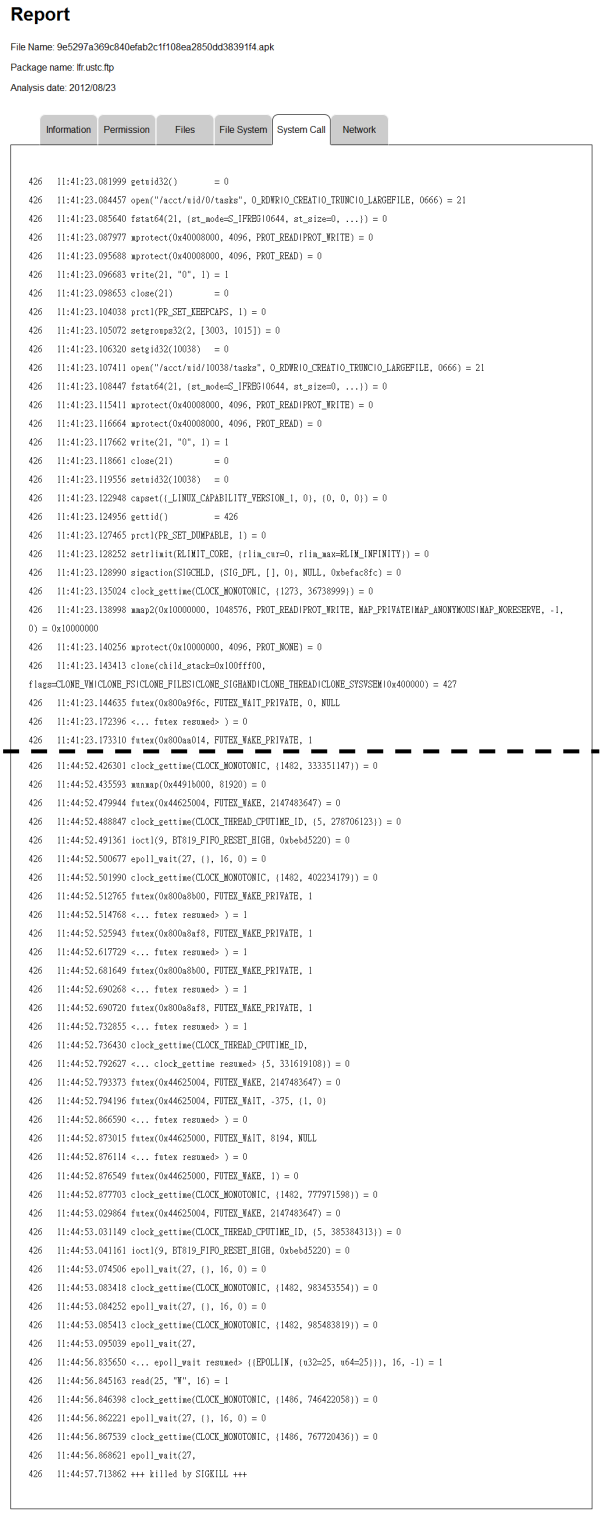


Figure 10 A sample screenshot that shows the system call traces monitored by the propsoed system.

and we expect that by mining the CFG strings for both benign and malicious applications would help the research on automatic generation of CFG string signatures.

In addition to androguard, we also use anti-virus software to check whether an examined application is a malicious application. This information is stored along with the report so that a user is able to determine whether a submitted APK file is a known malicious malware or not. Currently we use the open source ClamAV virus scanner and the commercial Avira virus scanner in the Linux host. Note that we did not scan for virus inside a virtual machine instance. Instead, it is performed outside of the VM to simplify the implementation of virtual machine images.

```
com.rovio.angrybirds:
0,0,0,1,1,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,1,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,benign
```

Figure 11 An example of the AngryBird bit-vector for the mix of requested and required permissions retrieve from the analytical report.

3.5 The Implementation

We implement all the inspection instances in a single virtual machine image. For the ease of expanding the system, we use `qemu-kvm` as the core of our virtualization technique and host virtual machines on an Ubuntu Linux operating system. Each virtual machine image contains all the introduced components. The job dispatching from the web portal is done by using HTTP requests as well. On receipt of a fresh APK, an inspection request is sent from the web portal to a selected VM instance. When an inspection is done, the results are then stored into the "APKs and Logs" repository.

To simplify the design, the web portal provides a single file chooser for users to choose a local file and upload it to our system. To prevent a malicious file from triggering alerts from local anti-virus scanners, we recommend a user to temporarily disable the real-time virus scanner and encrypt a malicious sample APK files as a password protected ZIP file with a password of "infected." Figure 9 shows the screenshot for the web portal. Figure 10 shows the system call traces obtained from an uploaded application for three minutes of monitoring. It has made 1,623 system calls but due to the limit of the page length, we only leave the head and the tail part.

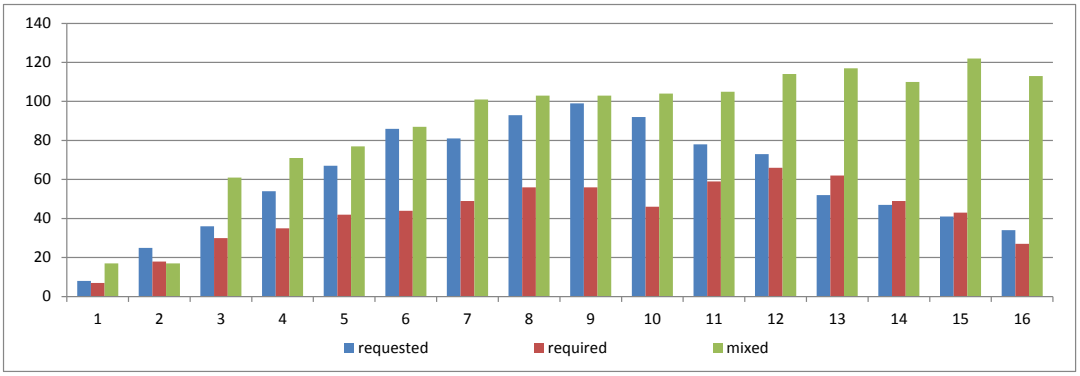


Figure 12 The number of bit vectors collected for different value of N (ranging from 1 to 16).

4 Application

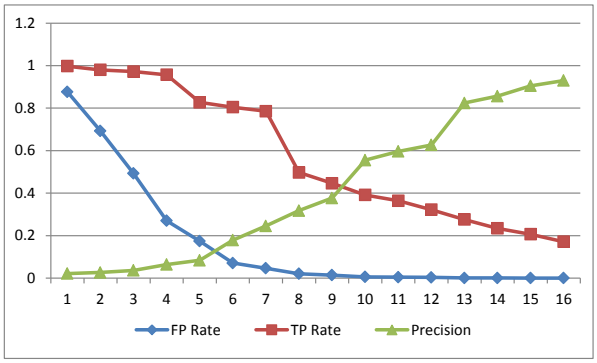
Based on our analytical report for 120 thousands android applications, we have designed a preliminary quick filter to exclude benign samples from being further inspected. The filter is based on a simple assumption: Can we identify a suspicious application by simply checking the requested and required permissions of an application?

To answer the question, we retrieve the requested and the required permissions reported by the system. We then create three data sets: #1) the requested permissions; #2) the required permissions; and #3) the mix of the requested and the required permissions. We translate each data set into a bit vector to indicate what permissions are needed by an application. Since Android has 139 built-in permissions, both dataset #1 and dataset #2 have exact 139 bits but dataset #3 has 278 bits. Figure 11 shows an example for the AngryBird game for dataset #3. In addition to the bit values, the virus scanning tag is attached at the end of the vector.

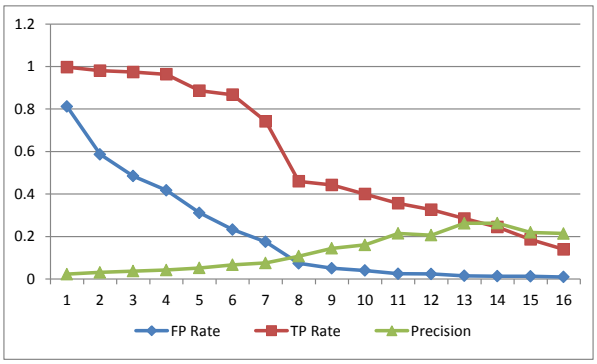
We then create the malicious application filter by collecting bit vectors of malicious samples. To reduce false positives, a bit vector with more than or equal to N bit-1 is preserved. Otherwise, it is dropped. The set of all the collected bit vectors is used as the quick filter. The number of bit vectors within a filter created by using different N is shown in Figure 12. Note that duplicated bit vectors are eliminated so the bit vectors with in a filter are all distinct.

To quickly filter out benign applications, for each suspicious application, we retrieve its requested and required permissions, create the bit vector V for the application, and then make a bitwise AND operation iteratively for V and each bit vector V' in the filter. If a V AND V' operation is equivalent to V', we believe that the application is a possible malicious candidate.

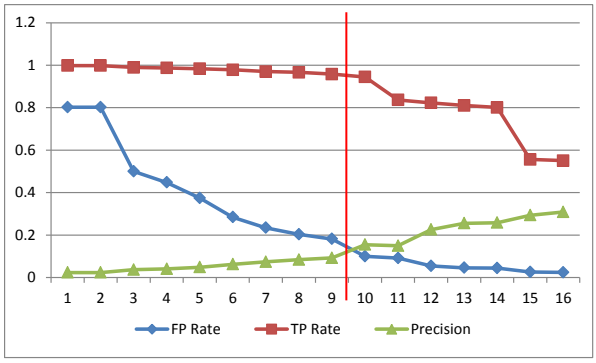
Based on the above strategy, we evaluate filters created for the three data sets. The results are shown in Figure 13. We found that the filters created from



(a) Based on requested permissions.



(b) Based on required permissions.



(c) Based on the mixed permissions.

Figure 13 The performace of the fiilters created using different datasets.

either requested permissions or required permissions alone did not perform well. But the filter created from the mix of requested and required permissions has a good performance if the number N is well-selected. When N is set to 9, there are 103 bit vectors within the filter. The resulted filter is able to detect approximately 96% of malware while the false positive rate is only 18%. This also means that the filter is able to detect most malicious applications and is able to reduce more than 80% workload of the backend fully featured malicious software analyzer.

5 Conclusion

The open design of the Android operating system provides its users a great flexibility on selecting the source of applications. However, a user still has to take the risk of installing a malicious application since not all applications have passed security checks. We would like to reduce the risk by providing more behavioral information about an application to both users and researchers. In addition to understand an individual application's behavior, our preliminary application also shows that by observing behavior from a bunch of application, it is possible to find out good solutions to classify applications. We expect the analytical reports provided by the proposed service can be good materials for researchers to have complete pictures of applications' behavior and therefore stimulates studies on the detection of malicious applications.

Acknowledgment

This work was support in part by National Science Council by the grant NSC 102-2219-E-019-001. We thank the anonymous reviewers for their insightful and helpful comments.

References

[1] David Barrera, H. Güne s Kayacik, Paul C. van Oorschot, and Anil Somayaji, A methodology for empirical analysis of permission-based security models and its application to android, Proc. ACM conference on Computer and Communications Security (CCS), Chicago, IL, USA, 2010, pp. 73-84.

[2] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak, An android application sandbox system for suspicious software detection, Proc. IEEE International Conference on Malicious and Unwanted Software (MALWARE), Nancy, France, 2010, pp. 55-62.

[3] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani, Crowdroid: behavior-based malware detection system for android, Proc. ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Chicago, IL, USA, 2011, pp. 15-26.

[4] Silvio Cesare and Yang Xiang, Classification of malware using structured control flow, Proc. Australasian Symposium on Parallel and Distributed Computing (AusPDC), Brisbane, Australia, 2010, pp. 61-70.

[5] Anthony Desnos, androguard - reverse engineering, malware and goodware analysis of android applications ... and more (ninja !), Google Project Hosting, [online] http://code.google.com/p/androguard/.

[6] William Enck, Machigar Ongtang, and Patrick McDaniel, Understanding android security, IEEE Security and Privacy, Vol. 7, No. 1, 2009, pp. 50-57.

[7] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner, Android permissions demystified, Proc. ACM conference on Computer and Communications Security, Chicago, IL, USA, 2011, pp. 627-638.

[8] Ryan Johnson, Zhaohui Wang, Corey Gagnon, and Angelos Stavrou, Analysis android applications' permissions, Proc. IEEE International Conference on Software Security and Reliability Companion (SERE-C), Gaithersburg, Maryland, USA, 2012, pp. 45-46.

[9] Mohammad Nauman, Sohail Khan, and Xinwen Zhang, Apex: extending android permission model and enforcement with user-defined runtime constraints, Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS), Beijing, China, 2010, pp. 328-332.

[10] Pouik and G0rfi3ld, Similarities for fun & profit, Phrack #68, April 2012, [online] http://www.phrack.org/issues.html?issue=68&id=15\#article.

[11] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan, User-driven access control: Rethinking permission granting in modern operating systems, Proc. IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 2012, pp. 224-238.

[12] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Clausen, Osman Kiraz, Kamer A. Yuksel, Seyit A. Camtepe, and Sahin Albayrak, Static analysis of executables for collaborative malware detection on android, Proc. IEEE International Conference on Communications (ICC), Dresden, Germany, 2009, pp. 631-635.

[13] Yajin Zhou and Xuxian Jiang, Dissecting android malware: Characterization and evolution, Proc. IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 2012, pp. 95-109.
[14] Googlesource, Android git repositories, [online] https://android.googlesource.com/
[15] Seung-Hwan Ju, Hee-Suk Seo, and Jin Kwak, Study on Analysis Methodology for Android Applications, Journal of Internet Technology, Vol. 14, No. 5, 2013, pp. 851-857.

Biographies

Chun-Ying Huang received his Ph. D. degree in Electrical Engineering from National Taiwan University, Taiwan in 2007. He joined Computer Science and Engineering Department at National Taiwan Ocean University in 2008, where he is currently an associate professor. Dr. Huang's research interests include network security, multimedia networking, and cloud computing.

Shang-Pin Ma received his Ph.D. in Computer Science and Information Engineering from National Central University, Taiwan, in 2007. He has been an assistant professor of Computer Science and Engineering Department, National Taiwan Ocean University, Taiwan, since 2008. His research interests include service-oriented computing, software engineering, and semantic web.

Ming-Lun Chang received his B.Sc. degree in Computer Science and Engineering from National Taiwan Ocean University, Taiwan in 2013, where he is now pursuing his master degree. His current researches focus on network transport protocols and cloud computing.

Chin-Hsiang Chiu received his B.Sc. degree in Computer Science and Engineering from National Taiwan Ocean University, Taiwan in 2013, where he is now pursuing his master degree. His current researches focus on mobile security and software evaluation.

Ting-Chun Huang received his B.Sc. degree in Computer Science and Engineering from National Taiwan Ocean University, Taiwan in 2013, where he is now pursuing his master degree. His current researches focus on computer networks and cloud security.