An Empirical Study on Android malware behavior signature extraction

Thien-Phuc Doan, Long Nguyen-Vu, Huy-Hoang Nguyen, and Souhwan Jung*
{phucdt, longnv, hoangnh, souhwanj}@soongsil.ac.kr
Soongsil University, Seoul, South Korea

Abstract

Malicious applications, especially in mobile devices, constitute a serious threats to user data. Due to the openness, Android have become the most popular smart phone operating system in mobile market. Although fast and straightforward, Static analysis approach has many difficulties to detect stealthy malware. In this paper, we propose a behavior signature-based to classify whether malware or not. To achieve this, we first hook a number of sensitive APIs to collect all possible invocations of the app. We then extract the behaviors of malicious applications by comparing their flows and the value of parameters and results of each called APIs. In our study, we extracted behavior signatures from malware in each family. Our works help to improve the quality of analysis compared with static analysis-only approaches.

Keywords: Dynamic analysis, Android malware detection, Hooking, Behavior signature

1 Introduction

Malware have become very popular in Android platform. According to IDC report[13], Android OS accounts for 85.9% in the mobile market share 2019. Therefore, Android OS also the first attacker's target. As shown in statista.com, the total number of Android malware detection amounted to over 26.6 million programs[6].

Due to the platform nature for mobile devices, malware in Android follow these main types: Adware, Ransomware and Spyware. Adware is the most common and popular android malware that a smartphone gets infected with[16]. This kind of malware try to run as many as possible advertisements on the infected phone. You will receive continuous popups and ads on your screen. Also, another malicious or unwanted applications will be downloaded whenever the ads are clicked. Ransomware try to make problem to the smartphone such as change settings or lock personal data. Similar to computer malware, Ransomware demand the users that they have to pay the money to unlock their devices. Spyware on your Android can monitor, record and send all your information to the attackers. It may steal all the data stored in your device.

So far, many solutions have been proposed for android malware detection. These works are conducted as either static or dynamic analysis or their combination[15]. Static analysis techniques identify as much as possible the workflows of the application (e.g. FlowDroid[3]) through source code to figure out suspicious behaviors. On the other hand, dynamic analysis identifies the workflow by executing the application in an emulator or real device to detect any malicious behavior. Each of them has their own pros and cons. Static analysis works by just disassembling the APK without actually running it, therefore it does not infect the device[9]. This approach is undermined by the use of various code transformation techniques (i.e. string encryption, using reflection APIs). Dynamic analysis passes against the encrypted,

Research Briefs on Information & Communication Technology Evolution (ReBICTE), Vol. 5, Article No. 11 (November 30, 2019)
*Corresponding author: Souhwan Jung, School of Electronic Engineering at Soongsil University, Seoul, South Korea, tel: 82-2-820-0714, email: souhwanj@ssu.ac.kr, web: `http://cnsl.ssu.ac.kr/`

polymorphic and code transformed malware, but it is not quick as static approach. A serious drawback of dynamic approach is that some malicious execution path may get missed, if it is triggered according to some non-trivial event[9].

In this paper, we propose a novel approach for detecting whether Android malware using dynamic analysis. We use hooking technique from Frida[2] to extract the behavior flow of an application without modifying APK source. After that, we conduct a behavior signature-based analysis to detect malicious app. We then classify the malware into their families.

Our contributions can be described as follows:

1. We propose a new approach to get the dynamic flow of an application. This makes the tracking work in dynamic analysis easier than other works.

2. We design a framework named **BeeDroid** which performs dynamic analysis with low cost of overhead. This framework aims to extract behavior signature of each malware family in order to support the detection system.

The rest of this paper is organized as follows: Section 2 provides an introduction about Frida instrumentation toolkit. This section also provides and discusses related works and their limitations. Section 3 describes our **BeeDroid** design and approach. Section 4 shows the limitation of our approach and discussion about some solutions for these problems. Section 5 summarizes the contribution with conclusion and scope of our future works.

2 Background and related works

Android framework provide developer many useful APIs to interact with the resources. Most of Android malicious behavior is trying to access the data or capabilities of the devices. Thus, to gather the information about an application's suspicious activities we need to track when and where the sensitive API is called.

Frida is a dynamic code instrumentation toolkit. It lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, GNU/Linux, iOS, Android[2]. With Frida, we can easily hook in any application run-time without any modification. In the other side, Frida has its own binding client in python and other scripting language which allow you to build your own automatic tool.

Mariconti et al. [14] proposed to build Markow Chains of behavioral model of a malicious malware with Soot[17] and Flowdroid [3] before analysis this behavior graph. Gilbert et al. [10] proposed AppInspector which can extract explicit and implicit flow by focusing on the return value of sensitive APIs. However, an application using sensitive data does not necessarily correspond to malware[5]. In this case, it is difficult to extract exactly the flow of what malware do because of obfuscation or others evasion techniques.

Most of current dynamic-based analysis techniques is trying to extract the sensitive information flow by executing the application and detecting anomaly behaviors. RunDroid[19] takes the source code of an application as input, instruments the source code and then intercepts the executions of the instrumented application to analyse message objects as set of log files. AndroTaint[15] represents an automatic tagging technique using DDI hooking which replace arbitrary methods in Dalvik code with native function call using JNI[7]. TaintDroid[8] builds their own system similar with android system by modifying the Dalvik VM Interpreter to marking all taint tags. By doing this way, their system extracts many unnecessary logs and spends the high cost in implementation work. In our work, the hooking becomes more efficient because it does not modify an application before its execution.

Many extraction data log solutions have been proposed. Xiao et al. [18] proposed AROW, a system call tracking tools using strace. Lin et al.[12] adopted the thread-grained system call sequences to identify

the malicious repackaged applications. Blokin et al.[4] proposed a method that incorporates sequence information into the features it uses to perform similarity analysis. Isohara et al.[11] proposed a kernel-based behavior analysis for Android malware inspection. These approaches have the limitation that their logs are too huge, and it may cause overhead when analysing them.

3 BeeDroid system design

To minimize the amount of API hooking workload of dynamic analysis, we design a framework named **BeeDroid**. The operation of BeeDroid goes through 3 phases, as depicted in Figure 1. First, we extract the flow named *call-flows* while malicious application runs. Then we compare all flow logs of each malware family to extract behavior signature of this family. Finally, we use the behavior signature to detect malicious application in Malware detection phase.

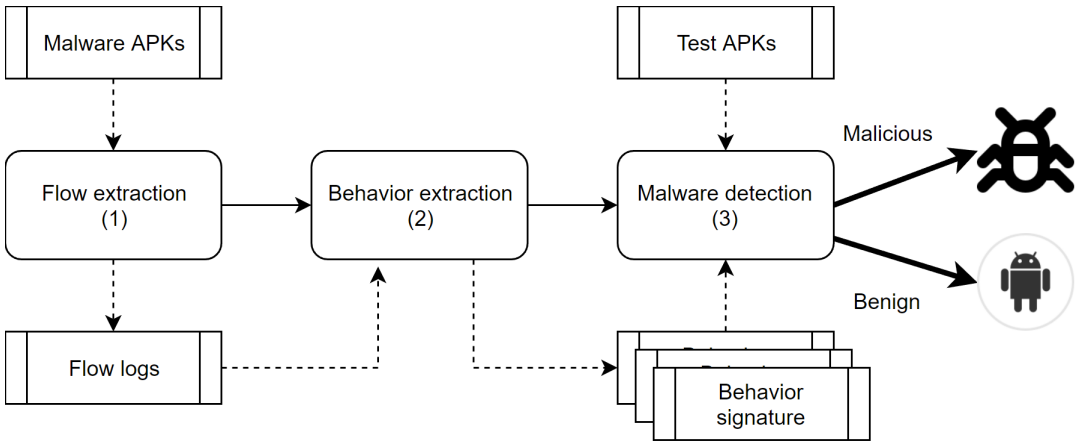

Figure 1: BeeDroid workflow overview

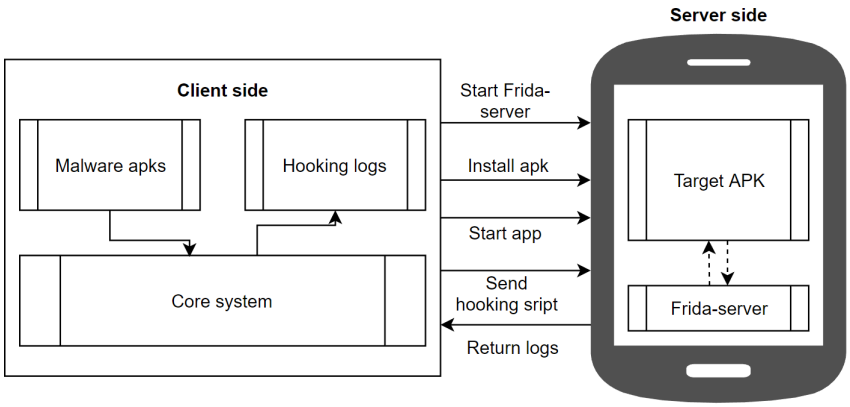

Figure 2: Hooking model

3.1 Flow extraction

We focus on monitoring the flow to sensitive APIs which are used to support application accessing to the resources of android device. These APIs existing in malware often incur unusual behaviors.

3.1.1 Hooking

The hooking system has two components: client side and server side. Client side works as an automation tool in order to automatically install malicious application in server side and retrieve analysis data. As shown in Figure 2, the workflow of hooking has these steps:

1. Client sends a request to start Frida-server.
2. Client installs malware APK using adb command.
3. Client sends a request to start the malware.
4. Server starts malware app and response process information to client.
5. Client sends hooking script.
6. Frida-server instruments the hooking script into the memory of targeted application then send back the hooking logs to client.
7. Client receives analysis log from server.

3.1.2 Flow logs collection

getStackTrace() is a powerful method to get the tracing flow (from source to sink) at any where this method is call. Figure 3 shows the stack trace log if an error occurs at *SmsManager.getSubcriptionId* API. Base on the log, we easily collect the source (*html.app.o.doPayment*) to the sink (*getSubscriptionId*) without any complex algorithms. Thus, we instrument the hooked API at runtime to add a call to *getStackTrace()* before return the results of instrumentation API. Then we collect the stack trace logs as the taint flow for the sensitive API when it's called.

```
[*]Stack trace:
dalvik.system.VMStack.getThreadStackTrace at VMStack.java line -2
java.lang.Thread.getStackTrace at Thread.java line 1566
android.telephony.SmsManager.getSubscriptionId at SmsManager.java line -2
android.telephony.SmsManager.sendTextMessageInternal at SmsManager.java line 318
android.telephony.SmsManager.sendTextMessage at SmsManager.java line 301
android.telephony.SmsManager.sendTextMessage at SmsManager.java line -2
html.app.c.a at null line -1
html.app.h.a at null line -1
html.app.h.a at null line -1
html.app.o.doPayment at null line -1
org.chromium.base.SystemMessageHandler.nativeDoRunLoopOnce at SystemMessageHandler.java line -2
org.chromium.base.SystemMessageHandler.handleMessage at SystemMessageHandler.java line 39
android.os.Handler.dispatchMessage at Handler.java line 102
android.os.Looper.loop at Looper.java line 154
android.os.HandlerThread.run at HandlerThread.java line 61
```

Figure 3: Stack trace log at SmsManager.getSubscriptionId

Both benign apps and malware apps use sensitive APIs, one is for legitimate reason and the other is not. Some APIs are used in the same way between malware and benign apps, so the taint flows are similar. For example, putExtra API adds extended data to the intent. This API is used to transfer data between activities or services. Both benign app use putExtra() to share data locally in application. Malware app use putExtra() as the means to call an implicit activity. As shown in Figure 4, Backdoor.AndroRAT.1 try to connect to a Command and Control server (C&C) "fatalxerror007.no-ip.biz".

In this paper, we designate *call-flow* of an application is the set of *Caller, Callee, Param_values* and *Return_value*. Caller is a correctly ordering set of methods for expressing call functions to sensitive APIs.

```
if (this.myIp == "") {
    this.ipfield.setText("fatalxerror007.no-ip.biz");
    this.portfield.setText("9999");
    this.Client.putExtra("IP", this.ipfield.getText().toString());
    this.Client.putExtra("PORT", Integer.parseInt(this.portfield.getText().toString()));
```

Figure 4: putExtra() in malware source code

In our work, we do not consider the APIs in Android framework layer. Callee is the sensitive APIs that we consider in this behavior. Params_value is a dictionary of type of parameter and its value which is the input of Callee. Return_value is the output of Callee after doing the action.

3.2 Behavior extraction

An application has many *call-flows* followed by order. Each call-flow is shown in one line of log file after Flow extraction. As shown in Figure 6, line 1 is a call-flow that represent the behavior from *onSms-ButtonClick()* raised to *File.exists()* API. Note that, not all call-flows represent for malicious behaviors. In this process, we try to extract the behavior signature of a malware family.

In a malware family, apps usually have different activities, functions and behaviors. However, they usually aim at the same goal (sink). For example, a malware belong to **AndroRAT**[1] family always connects to a C&C server in an implicit way by silently starting new activity with IP address and Port number of C&C server added earlier using *putExtra()* API. In this paper, a set of call-flows that almost all malware in a family have is called behavior signature. In simple malware families, their behaviors are just a sequence of call-flows. However, complicated malware families have additional junk-behaviors between their malicious call-flows.

Considering the behavior of an entity is not only assessing the number of actions but also evaluating the order in which these actions are performed. Hence, to get the most accurate view of the behavior of a malicious app, we should consider the correlation of their call-flows.

The first step of our method is generating Callee vector of each malware behavior. Then we extract the Callee matching blocks of all malware behavior in the family. Note that Callee must be exactly match because API name is pre-defined by Android SDK and represents a specific function. After getting Callee matching blocks, we consider about the similarity of Caller, Param_values and Return_value. These part of call-flow no need to be exactly match. For example, Erop family try to send SMS to premium number for money. Each Erop malware have different list of premium number. Thus they pass different parameter value to sendMessage() API, but still have the similarity between them such as length of premium number. In our work, we use difflib implemented by python to calculate the similar ratio of Caller, Param_values and Return_value between each malware in the family. If the ratio exceeds a threshold θ we accept this call-flow and append it into behavior signature. The method to extract behavior signature is described in detail in Algorithm 1.

The *get_matching_block()* method, which implemented by python, returns a list of matching block. Each match describes matching subsequences as a form (a, b, size) with a, b is starting index of object rootsig and familyls[i] and n represents the length of matching block. The Similar() method using *difflib* (a python library) to calculate the ratio of similarity between two lists or strings in sequence correlation.

```
1 familyls = all_apks_call_flows
2 rootsig  = familyls[0]
3 for i=1 to len(familyls) do
4     matches = SequenceMatcher(rootsig.CalleeVector,
5                               familyls[i].CalleeVector).get_matching_block()
6     tmp = []
```

```
     for match in matches do
         flowsA = rootsig[match.a:match.a + match.size]
         flowsB = familyls[i][match.b:match.b + match.size]
         matchCaller = similar(flowsA.Caller, flowsB.Caller)
         matchParam = similar(flowsA.Param, flowsB.Param)
         matchRes = similar(flowsA.Ret, flowsB.Ret)
         if (matchCaller > CallerThreshold and
               matchParam  > ParamThreshold  and
               matchRet    > RetThreshold) do
               tmp.append(match)
    end for
    if(len(tmp)>1)
          rootsig = tmp
end for
Behavior_Signature = rootsig
```

Listing 1: Algorithm 1 - Procedure for the behavior signature extraction

3.3 Malware detection

Base on the behavior signatures that extracted by Behavior detection phase, we compare the call-flows of unknown Android application with all the signature we have. The detail for this phase is show as Algorithm 2.

We do the same way to extract the matching block between the behavior of unknown app and each family behavior signature. BY asserting the malware database, we came up with the optimal α value of 0.87

```
signatures = all_families_behavior_signature
testFlows  = all_callflows_of_unknown_apk
for i=0 to len(signatures) do
    matches = SequenceMatcher(testFlows.CalleeVector,
                          signatures[i].CalleeVector).get_matching_block()
    tmp[]
    for match in matches do
        flowsA = testFlows[match.a:match.a + match.size]
        flowsB = signatures[i][match.b:match.b + match.size]
        matchCaller = similar(flowsA.Caller, flowsB.Caller)
        matchParam = similar(flowsA.Param, flowsB.Param)
        matchRes = similar(flowsA.Ret, flowsB.Ret)
        if (matchCaller > CallerThreshold and
               matchParam  > ParamThreshold  and
               matchRet    > RetThreshold) do
                tmp.append(match)
    end for
    testFlows = tmp
    ratio = similar(testFlows,signatures[i])
    if (ratio > α) do
        alert("Malware detected!")
end for
```

Listing 2: Algorithm 2 - Procedure for malware detection

4 Limitation and discussion

While conducting **BeeDroid** to construct call-flow behaviors, we observe some limitations of this tool. First, if the environment is a well-known emulator, malware apps which have anti-debug, anti-emulator techniques may refuse to run. This problem can be solved by using customized emulator which fakes device information. Second, Frida-server process can be easily detected. However, we can rebuild Frida-server with another name and bind it to another port number. Finally, some sensitive APIs that interact with same resource have different names and even different ways to use (e.g. parameters). But these APIs account for a small percentage in SDK sensitive APIs. In other words, in a malware family, malicious apps use the same APIs. If we divide the malware in the family as detail as possible, it will be easier to apply our techniques.

5 Conclusion

In this paper, we have proposed a framework to construct Android malware behaviors and define their signatures for further analyses. **BeeDroid** is a novel approach that reduces the overhead for analysis process while avoiding sacrificing analysis data. For future work, we will try to improve our framework to make the similarity calculation more precise. However, preparing for the analysis environment as like real phone as possible and equip more functionalities to our analysis environment to trigger more evasive apps.

Acknowledgement

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP), a grant funded by Korea government (Ministry of Science and ICT) (no. 2016-0-00078, Cloud Based Security Intelligence Technology Development for the Customized Security Service Provisioning).

References

[1] Android malware dataset. `http://amd.arguslab.org/` [Online; accessed on September 15, 2019].

[2] Frida. `https://www.frida.re/docs/home` [Online; accessed on September 15, 2019], 2019.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. Mcdaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not*, 49(6):259–269, June 2014.

[4] K. Blokhin, J. Saxe, and D. Mentis. Malware similarity identification using call graph based system call subsequence features. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops (ICDCSW'13), Surat, India*, pages 6–10. IEEE, July 2013.

[5] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proc. of the first ACM workshop on Security and privacy in smartphones and mobile devices (SPSM'11), Chicago, Illinois, USA*, pages 15–26. ACM, October 2011.

[6] J. Clement. Global android malware volume 2018. `https://www.statista.com/statistics/680705/global-android-malware-volume` [Online; accessed on September 15, 2019], 2019.

[7] V. Costamagna and C. Zheng. Artdroid: A virtual-method hooking framework on android art runtime. In *Proc. of the First International Workshop on Innovations in Mobile Privacy and Security & Engineering Secure Software and Systems (IMPS@ESSoS'16), London, UK*, pages 20–28. ceur-ws, February 2016.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. Mcdaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the*

9th USENIX conference on Operating Systems design and implementation (OSDI'10), Berkeley, California, USA, page 1–29. USENIX Association, January 2014.

[9] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, December 2015.

[10] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proc. of the second international workshop on Mobile cloud computing and services (MCS'11), Bethesda, Maryland, USA*, pages 21–26. ACM, January 2011.

[11] T. Isohara, K. Takemori, and A. Kubota. Kernel-based behavior analysis for android malware detection. In *Proc. of the 2011 Seventh International Conference on Computational Intelligence and Security (CIS'11), Hainan, China*, pages 1011–1015. IEEE, December 2011.

[12] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39:340–350, November 2013.

[13] R. M. Chau. Smartphone market share - os. `https://www.idc.com/promo/smartphone-market-share` [Online; accessed on September 15, 2019], 2019.

[14] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *ACM Transactions on Privacy and Security*, 22(2):14:2–14:34, April 2017.

[15] V. G. Shankar, G. Somani, M. S. Gaur, V. Laxmi, and M. Conti. Androtaint: An efficient android malware detection framework using dynamic taint analysis. In *Proc. of the ISEA Asia Security and Privacy (ISEASP'17), Surat, India*, pages 1–13. IEEE, January 2017.

[16] T. L. TEAM. Some common and popular types of android mobile malware. `https://tweaklibrary.com/some-common-and-popular-types-of-android-mobile-malware` [Online; accessed on September 15, 2019], 2018.

[17] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *Proc. of the Centre for Advanced Studies on Collaborative research (CASCON'99), Mississauga, Ontario, Canada*, page 13. IBM Press, November 1999.

[18] X. Xiao, X. Xiao, Y. Jiang, X. Liu, and R. Ye. Identifying android malware with system call co-occurrence matrices. *Transactions on Emerging Telecommunications Technologies*, 27(5):675–684, March 2016.

[19] Y. Yuan, L. Xu, X. Xiao, A. Podgurski, and H. Zhu. Rundroid: recovering execution call graphs for android applications. In *Proc. of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17), Paderborn, Germany*, pages 949–953. ACM, August 2017.

Author Biography

Thien-Phuc Doan is currently a Master student at Soongsil University, Seoul, Korea. He received his B.S. in Cyber Security in Vietnam National University, Ho Chi Minh city, University of Information Technology in 2018. His research interests are in the areas of Web Security, Mobile Security and Malware Analysis.

Long Nguyen-Vu is currently a PhD student at Soongsil University, Seoul, Korea. He received his B.S. in Computer Science in Vietnam National University of Information and Technology in 2012, and his M.S. degree in Electronic Engineering from Soongsil University in 2016. His research interests are in the areas of Big Data, Mobile Security, Cloud Security with emphasis on Malware Analysis and System Design.

Huy-Hoang Nguyen is currently a Master student at Soongsil University, Seoul, Korea. He received his B.S. in Information Technology in National University of Civil Engineering, Hanoi, Vietnam in 2012. His research interests are in the areas of Artificial Intelligent apply to Mobile Security and Cloud Security area with emphasis on Malware Analysis and System Design.

Souhwan Jung is a professor of the School of Electronic Engineering at Soongsil University, Seoul, Korea since 1997. He has spent about 20 years on designing network security protocols. He also served as a government R&D program director for Information Security of MKE during 2009–2010. He has led a security research center funded byMSIP during 6 years since 2012. His current research interests include Mobile Security and Android Malware Analysis.