Capturing Android Malware Behaviour using System Flow Graph

Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong

Capturing Android Malware Behaviour using System Flow Graph

Radoniaina Andriatsimandefitra[1] and Valérie Viet Triem Tong[1]

CIDRE research group,
SUPELEC
Avenue de la Boulaie, 35510 Cesson-Sévigné, France
[1] firstname.lastname@supelec.fr

Abstract. This article uses a new data structure namely System Flow Graph (SFG) that offers a compact representation of information dissemination induced by an execution of an application to characterize malicious application behavior and lead some experiments on 4 malware families DroidKungFu1, DroidKungFu2, jSMSHider, BadNews. We show how SFG are relevant to exhibit malware behavior.

1 Introduction

Android is an operating system dedicated to mobile devices. Due to its widespread adoption and the sensitive nature of data such devices may contain, Android became the target of malicious applications. As pointed out in different studies [1, 2], Android security mechanism is not efficient enough to perfectly protect users and sensitive data on their device from malware. Security groups have thus worked on security extensions for Android to improve its security level.

Monitoring applications to detect misbehaviours appears to us as a promising idea but we also believe that the analysis should not be restricted to the application itself but to its impact, direct or not, on the entire system. Indeed, traditional approaches limit themselves to the application itself and its direct interaction with its environment [3–6]. By doing so, some useful information about the action of the application may remain unnoticed. For instance, monitoring only a running sample of DroidKungFu1[7] and its child processes on Android, does not allow to observe that once it drops one of its payload in `/system/app`, the content of the payload is automatically used by two main system processes and propagated to other files in the system.

To overcome this shortcoming of traditional profiles, we choose to monitor information flows caused by an application in the whole system. Therefore we propose to define an application profile as a representation of how the application disseminates its data in the whole system and how these pieces of data are processed by the other applications. For that purpose, we use a data structure previously introduced in [8]. This data structure, called System Flow Graph (SFG), represents how an application disseminates its own pieces of data in an operating system during an execution. In this article we present how this structure helps to understand and classify malware.

As stated in [9], malware authors are used to repackage applications with their malicious code and submit these repackaged applications on distribution platforms to infect users. These repackaged applications represent 86% of the malware samples in Android genome project according to the same source. As a malware author infects different applications with the same malicious code, these applications share a common behaviour and consequently we claim that they should share a common sub-SFG. This will be able to cope even with code that is obfuscated or ciphered. In the following we present related works in application analysis and tainting techniques for malware detection (section 2). Then, we present the underlying model of the information flow monitor we choose to use and a compact format to describe information flows (section 3). We then explain how a SFG is used to build a malware signature based on its behavior (section 4) and experiment our proposition (section 5).

2 Related works

Tainting consists in marking pieces of information to monitor how they are disseminated in a program or in a system. They have been used to analyse how applications access sensitive data and how they process it. In [10], Yin et al. proposed to monitor information flow at hardware level to detect if suspicious applications accessed sensitive data and how they processed it. In subsequent work [11], Yin et al. used a similar approach and presented DroidScope, an Android analysis environment. Like Panorama, DroidScope monitors information flow at hardware level and aimed at detecting if a monitored application accesses to data considered as sensitive, when it is the case the application is considered as malicious. Tainting techniques have also been used on real devices to identify Android applications that leak confidential data. In [1], Enck et al. present TaintDroid, a modified version of Android that monitors information flow at application level. They selected pieces of sensitive data (e.g: IMEI and location data) that should be considered as confidential and monitored how they spread thanks to tainting techniques. When an application accesses a piece of tainted data, TaintDroid considers that the piece of data is flowing and propagates the corresponding taint to the destination container. In TaintDroid, destination containers that can be tainted are Java application variables, IPC messages and files. TaintDroid raises an alert whenever a piece of tainted data leaves the device. Their study showed that more than a third of thirty popular applications are responsible of sensitive data leakage to remote entities. If TaintDroid detects information leaks, it does not however give enough information to diagnose how the leaks happened. Furthermore, it only monitors information flow at Java application level: information flows involving native applications are not detected by TaintDroid because they do not run inside the Dalvik virtual machine which has been modified to monitor information flow.

As tainting techniques permit to understand how pieces of sensitive data are used, they give a better insight about the intent of an application compared to its use of functions. We therefore claim that such techniques are a better candidate

to classify and detect pieces of malware. Unlike the approach used in DroidScope, Panorama and TaintDroid in which their authors focus on information they consider as sensitive, we think that the classification and detection should first be focused on the pieces of data owned by the application under analysis. To classify malware samples and detect their execution, we mark their origin container, `apk` file on Android and monitor how its content is disseminated in the system. From this dissemination profile, we regroup samples that propagate their own data in the same way and use this profile to detect execution of samples from the same malware family.

3 Capturing external behaviours of applications

In this work, we use Blare[1] [12], an intrusion detection system, to monitor information flows at system level. Blare is aware of information flows occurring between files, processes and sockets. Blare implementation relies on the Linux Security Module framework [13] that introduces hooks in the kernel to intercept syscalls. Blare uses these hooks to intercept syscalls that induce information flow between system objects and maintains tags on these objects. In particular, Blare maintains a tag called *itag* on each container of information at system level. This tag permits to know if a container has been contaminated by a marked piece of data. Besides intercepting syscalls, Blare also performs a finer monitoring of information flows occurring through the binder driver on Android. When Blare observes an interaction between system objects, it computes the corresponding information flow and performs the appropriate tag update. For example, when a process P reads a file F, it performs a `read` syscall. Blare intercepts this syscall and deduces that information flows from F to P. It then updates the tag associated to P to take into account its new content. If the file is later read by an other process then this process will have an *itag* indicating that its content has also been contaminated. In general, Blare updates the value of the *itag* attached to an object each time it considers that the content of this object has been changed. In this work, we use Blare to keep under surveillance a newly installed application that we do not trust. We assign a unique identifier i to information originating from the application and monitor how the pieces of information identified with i are disseminated in the whole system. Technically speaking this is done by assigning $\{i\}$ as the *itag* value of the `apk` file of the application we want to monitor. The `apk` file is the archive containing all the code and resources of Android applications. As we only focus on information from the application that we do not trust, there is no other identifier used in the system and the only possible values of *itag* are therefore $\varnothing$ and $\{i\}$. During an execution, if container c has an *itag* value equal to $\varnothing$, it means that c has not been contaminated by the marked application. Otherwise it means that the content of c may have been contaminated by the application. Each time Blare observes an information flow involving a content associated to an non-null *itag*, it adds an entry in its log to describe the observed flow. Each log entry describes the source container of

[1] `http://blare-ids.org/`

the flow, its destination and the information identifiers associated to the pieces of information that are propagating. It also contains a timestamp corresponding to the moment at which the flow was observed. Source and destination are described by their type, their name and their system identifier.

A log produced by Blare when monitoring an application exactly depicts how pieces of data owned by this application are disseminated within the system. However, the longer the monitoring of an application lasts, the bigger its log size grows. Some of the log entries may be repeated: for instance when a process reads a huge file it will repeat several times the same read syscall. We then propose in a data structure named System Flow Graph to compact the representation of a Blare log. A System Flow Graph (SFG) [8] describes how pieces of information are disseminated within the system during one execution. A SFG is a graph representation of a Blare log without information loss. A SFG has a more compact form. It is thus more readable than a log. Formally, a SFG is a labelled directed graph $G = (V, E)$. Each node $v \in V$ represents a container of information and each edge $e \in E$ from a node v_1 to a node v_2 an information flow from v_1 to v_2. Each node has three attributes: its system id, its name and its type (process, file or sockets). These three attributes are respectively denoted $v.id$,$v.name$ and $v.type$. An edge has two attributes identified as $e.flow$ and $e.timestamp$. $e.flow$ is a collection of information identifiers involved in the flow corresponding to e. The attribute $e.timestamp$ is a list of timestamps at which Blare observed the flow corresponding to e. SFG construction relies on the Blare log. We have a tool that transforms a log into an SFG.

4 System Flow Graph as behaviour-based signatures

As malware authors are used to repackage original applications with their malicious code, applications infected with the same malicious code exhibit a common behaviour. Their corresponding SFGs therefore share a common sub-SFG that represents this common behaviour. We propose to use this common sub-SFG as a malware signature and explain below how to compute it.

A SFG describes how a piece of data disseminates in the whole system. In this study we propose to monitor how data originating from an application archive (`apk` file) is processed and thus how an application contaminates the operating system. To obtain the corresponding SFG we mark the `apk` file of the application right after it is installed on the device. Once the archive marked, the application is launched and monitored by Blare.

In the following we present a core algorithm used to classify a set of SFG into different subsets where each subset share a common sub-SFG that is the signature of the subset. The main algorithm is a classifier computing a fixed-point on the call of the function named `one-step-classification` on a classification list. A classification list is a list that associates a signature with all the SFGs that include this signature. A signature is a sub-SFG that appears at least in two SFGs given in input. This way, if the SFG list given as input contains at least two samples of the same malware family, our algorithm will output a signature.

More precisely, the main algorithm takes as input a list `[`g_1`, ...,`g_n`]` of SFG and `white` a *white list* of SFG. First it stores in a variable named `assoc` a classification list initially set to $[(g_1, [g_1]), \ldots, (g_n, [g_n])]$. The main part is then a loop that computes a fix point of `one-step-classification(assoc, white)`. The function `one-step-classification` is described in algorithm 1. In short, it computes the biggest common part of a list of SFGs deprived of all part that also appear in SFGs of benign applications characterized by the SFG list `white`.

When a fixed point is finally reached, the main algorithm will output a classification list of the form `[(`s_0`, [`g_{0_1}`, ...,`g_{0_i}`]), ...,(`s_m`,[`g_{m_1}`, ...,`g_{m_k}`])]` where s_0, s_1 $\ldots s_m$ are the resulting signatures and $[g_{l_1}, \ldots, g_{l_i}]$ is the list of SFGs from the input that includes the signature s_l.

Algorithm 1: `One-step-classification` function

Input:
assoc a SFG classification list
white a list of trusted SFG
Output: a SFG classification list, one step further
begin
  new_assoc ← [] ;
  forall the $g_1 \in$ `keys`(*assoc*) **do**
    forall the $g_2 \in$ `keys`(*assoc*) **do**
      v ← [`value`(*assoc*, g_1)] ;
      s ← `clean` ($g_1 \cap g_2$,white) ;
      if $s \neq \varnothing$
      then
        forall the $g \in$ `keys`(*assoc*) **do**
          if *s is included in g* **then**
            v ← v @`value` (*assoc*, *g*)
        new_assoc ← *add*(*new_assoc*, (s, v));
  return *new_assoc*;

5 Computing SFG signatures

To evaluate our algorithm, we propose to extract SFG-signatures from the SFG of 19 malware samples: 5 samples of BadNews [14], 7 samples of DroidKungFu1 [7], 3 samples of DroidKungFu2 [15] and 4 samples of jSMSHider [16].

BadNews is a malware of which samples are disguised as legitimate applications. Based on a manual analysis of these samples, we know that they are clients of a Command and Control (C&C) server from which they receive commands to execute. The different commands they can receive are to download and to install Android applications, to display news notifications (web-page to visit, software

update etc) on the device, to install icons which links to an url or a downloaded Android application and to change the address of the C&C server. During the period of our experiment, the C&C server only sent the news command to advertise two infected-application updates: Doodle Jump and Adobe flash. DroidKungFu1 and DroidKungFu2 are malware families that attempt to gain root privileges on the device and stealthily dump malicious applications on the device when root privileges are gained. Like DroidKungFu families, jSMSHider also exploits a vulnerability to install applications on the device. According to Zhou et al. [9], samples of DroidKungFu1, DroidKungFu2 and jSMSHider are repacked applications to which a malicious code was added.

Analysis environment To dynamically analyze applications and produce Blare log, we used a Samsung Nexus S device running the version of Android Ice Cream Sandwich from the Android Open Source Project. We used a kernel to which Blare was added. In user-space we added Blare related tools to set tag values, a standalone version of a toolbox named `busybox` and an application named `Super User` to get notifications when applications use the `su` command. No additional applications or components were added or modified.

Application analysis We wanted to observe how an application disseminates its data within the system. We therefore installed the application and marked its `apk` file before its first execution. The `apk` file contains the resources and the code of the application when it arrives on the device.

Sample execution and monitoring The malicious code is not always automatically executed when the application is launched. We therefore introduced events in the system to trigger the malicious behaviour of some malware samples to shorten the duration of the application analysis. For some samples of DroidKungFu1 and DroidKungFu2, the associated value of a key named `start` in a file named `sstimestamp.xml` must be set to a small value (e.g 1). For BadNews, the malicious code is executed once a component of the application named `MainService` receives an `intent` that asks him to start running. In order to launch the application malicious code, the `intent` must come with an extra boolean value set to `true`. We manually craft this intent and send it to the application during the analysis of the samples of BadNews. In addition to introducing these key-events, we also use the application as a normal user.

SFG signature computation Once we obtained the logs resulting from the analysis of 19 samples, we built the corresponding 19 SFGs. We then gave these SFGs to a program that implements the classification algorithm. The program returned a classification of 4 groups. 17 out of the 19 samples are exactly classified as in their origin database. The two remaining are samples of DroidKungFu1 that were classified as belonging to the same group as the samples of DroidKungFu2. This is due to the fact that these two samples exhibit the same behaviour as samples of DroidKungFu2 and also produce the same information flows. This is thus not an error of our algorithm. The SFG-signatures associated to each class describe the malicious behaviour of the code introduced in repackaged applications. We present in figure 1 the signature computed for BadNews. It describes the download, a part of the installation and execution of two appli-

cations (`doodle.jump.apk` and `adobe.flash.apk`). We can see from the figure that the browser sends data to a server : `213.x.x.x`. We intentionally replaced a part of the IP addresses with the letter `x` to avoid revealing the original address. This address correspond to a remote server from which malicious applications are downloaded.

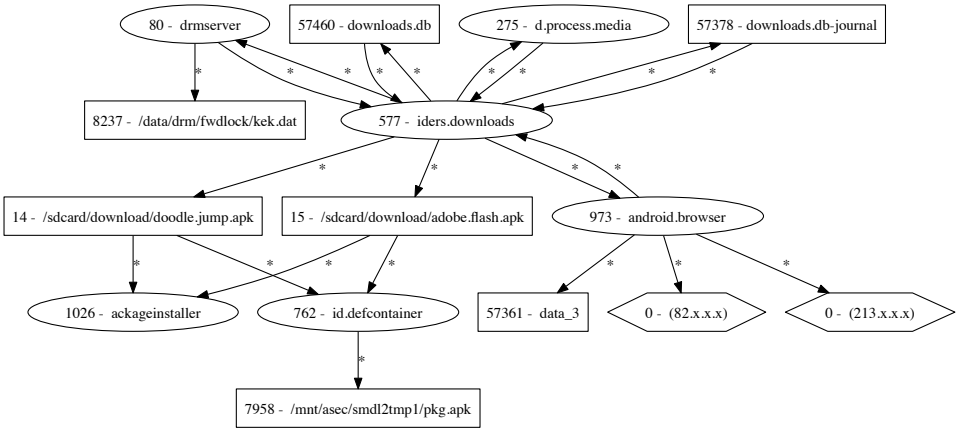

Fig. 1: SFG signature of BadNews

6 Conclusion

We made a proposition to classify Android malware in this work. First, we proposed to use a data structure named System Flow Graph (SFG) as a profile of an application. It describes in a compact and human readable way how a particular execution is responsible of information dissemination within the operating system and can be constructed from the log of an information flow monitor.

Second, we proposed to use SFGs to characterize malware samples. The main idea behind the approach is that when applications are infected by the same piece of malware, it should be possible to exhibit a similar sub-SFG in their respective SFGs. Following this idea we have proposed a classification algorithm that regroups SFGs according to the maximal sub-SFG(s) they have in common. We have applied the proposed algorithm to compute the signature of pieces of malware discovered during the last two years. Our algorithm has successfully extracted a signature for each malware family from which we picked samples for our experiment and each signature only matches the samples of the malware family to which they belong. In future work, we plan to use these signatures in a new form of malware detection engine.

References

1. Enck, W., Gilbert, P., gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., , Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI). (2010)
2. Vidas, T., Votipka, D., Christin, N.: All your droid are belong to us: a survey of current android attacks. In: Proceedings of the 5th USENIX conference on Offensive technologies, Berkeley, CA, USA, USENIX Association (2011) 10–10
3. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. J. Comput. Secur. **19**(4) (dec 2011) 639–668
4. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and classification of malware behavior. In: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA '08, Berlin, Heidelberg, Springer-Verlag (2008) 108–125
5. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009). (1 2009)
6. Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: Accessminer: using system-centric models for malware protection. In: Proceedings of the 17th ACM conference on Computer and communications security. CCS '10, ACM (2010)
7. Jiang, X.: Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html`.
8. Andriatsimandefitra, R., Viet Triem Tong, V., Mé, L.: Diagnosing intrusions in android operating system using system flow graph. In: Workshop Interdisciplinaire sur la Sécurité Globale. (2013)
9. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. SP '12, Washington, DC, USA, IEEE Computer Society (2012) 95–109
10. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM conference on Computer and communications security (CCS'07). (2007)
11. Yan, L.K., Yin, H.: Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Security Symposium. (August 2012)
12. Viet Triem Tong, V., Clark, A., Mé, L.: Specifying and enforcing a fine-grained information flow policy: Model and experiments. In: Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications. (2010)
13. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux security module framework. In: OLS2002 Proceedings. (2002)
14. Rogers, M.: The bearer of badnews `https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/`.
15. Jiang, X.: Security alert: New droidkungfu variants found in alternative chinese android markets `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu2/`.
16. Strazzere, T.: June 15, 2011 security alert: Malware found targeting custom roms (jsmshider) `https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting\-custom-roms-jsmshider/`.