Don't Be Surprised: I *See* Your Mobile App Stealing Your Data

Swarna Gopalan, Aditi Kulkarni, Arpitaben Narendrabhai Shah,
Jun Dai, Jinsong Ouyang, Pinar Muyan-Ozcelik and Xiaoyan Sun
Department of Computer Science
California State University, Sacramento, CA 95819, USA
Email: jun.dai@csus.edu

***Abstract*—To detect the potential malicious mobile applications that may cause data leakages, this paper[1] proposes to take advantage of the attack-neutral and hard-to-avoid system calls of the mobile system, reconstructing an activity graph for each application to reflect its interactions with the system. This paper makes efforts to automate the activity graph generation in Android, with the main contribution being an open source tool that can be of great assistance for the test team of application stores. The tool's effectiveness has been validated by our evaluation against some existing or home-brewed mobile applications that leak data.**

Keywords*—*data leakage, mobile application, security test, activity graph, system call, Android

I. Introduction

The recent security breaches like XcodeGhost hack [1] and iCloud nude leaks [2] scandal emit strong signals of potential occurrences of private data leakages associated with smart phones. With business data and personal data mixed and co-residing on the same device, it's a huge concern to have device owner's confidential information exposed to the unexpected audience. The concerns further extend to employers when the usage of mobile devices reaches the entry point to the business networks and infrastructure, namely entry point security.

However, revealed by a survey administrated as a part of this project which attracted 503 anonymous responses, most of the mobile users haven't gained adequate attention for the risks that come along with their mobile devices. Specifically, according to the survey, 34% of the users do not analyze and restrict permissions for mobile applications, 63% do not know that third party applications may be malicious and hidden after installation, and over 90% of the users who root their devices install third party applications with root privilege, etc.

The above data shows the significant gap between the mobile security demand and reality. On one side, the gap is inherently due to the vulnerabilities inherited from the insecure programming practices during system or application development, which is pretty hard to cope with; on the other side, the gap is rooted in the user-side lack of security understanding and awareness of how the system or application runs. Taking XcodeGhost as a good example, where the vulnerability originated from the altered versions of Apple Xcode development environment, both the authors and users of the infected applications were unaware of this security issue, even two months after the initial vulnerability report [3].

This paper diagnoses the unawareness as a result of the *black-box* view that is delivered by application developers and held by users. As indicated by its name, "black-box" describes the way to test or use an object by viewing it as merely a monolithic intermediate between given inputs and expected outputs. This view greatly eases users' device operation by hiding implementation complexity behind navigation interfaces, but also leads to poor awareness and management of the inside data flow. The same problem was observed and solved by Jiang et al [7] to reconstruct the semantic view hidden inside individual virtual machines. Compared to virtual machines which are inherently no different from traditional computers, mobile applications have more risks in privacy data leakage, as they run in an environment that is also a repository of personal data, such as the videos, audios, images, and texts collected or transmitted through the phone cameras, microphone, chatting tools, etc. In other words, proximity to user data turns into a big security concern due to the black-box view. To provide examples, this paper demonstrates how malicious mobile applications leak data in Section V.

To detect the potential malicious mobile applications that may cause data leakages, this paper proposes to break the aforementioned "black-box" view by our approach which features *visualization* of the mobile application activities. This feature allows us *see* mobile data leakages if any. And the most appropriate entity to perform this operation is the intermediate between the end users and application developers, i.e. the auditing department of the application stores, which also has

[1]This paper is a report of research outcomes from open source master projects [4]–[6], and the resultant code can be shared upon request.

the responsibility, resources and capability to do so. The insight of our approach is to take advantage of the attack-neutral and hard-to-avoid system calls of the mobile system, reconstructing an *activity graph* for each application to reflect its interactions with the system. In the event that a random test triggers the mobile application to leak user data, it will be captured and revealed visually by the activity graph. Figure 1 is an illustration of a home-brewed application named "Data Peek" leaking mobile contacts (phone numbers) through emailing via SMTP protocol.

This paper is an effort to automate the generation of such activity graphs, with the main contribution being an open source tool which can be of great assistance for the application test team. The tool's effectiveness has been validated by our evaluation against some existing or home-brewed mobile applications that leak data.

The rest of this paper is structured as follows: Section III describes the model and approach, Section IV provides the details of the tool implementation, and Section V gives the evaluation results. Section II reports the literature review, and Section VI concludes this paper.

II. Related Work

This paper mainly falls into the following two categories of research.

Dynamic Analysis for Android Malware Detection In contrast to static analysis, the dynamic analysis detects malware by watching its behavior at runtime. It may still work in the situations of code obfuscation and dynamic code loading, where static analysis definitely does not work. Taitdroid [8] tracks information at runtime to monitor privacy leakage for smartphones. The mobile dynamic analysis can be system call-based or API call-based. For example, Afonso et al. [9] leverages the frequency of system calls and API calls to differentiate malware from benign software. This paper differs from the above-mentioned previous work by being more application-specific, and is designed to promote the validation of mobile application security before applications go to the end users.

System Call based Intrusion Detection System calls were introduced as one way to detect whether a program has infected by Forrest et al. [10] and Lee et al. [11]. At the beginning, only sequence [12] [13] of the system calls was leveraged. Later system call arguments [14] [15], temporal properties [16], and other more sophisticated mechanisms start to be considered. Following the approaches presented in the study by King and Chen [17] and in our prior study [18] [19], this paper values causality relationship embodied by system calls, and extends the efforts to make this approach work in the context of mobile malware detection.

Table I: Tested Malicious Mobile Applications.

Malware	Description
Home-brewed ones	
Data Peek	Steal contact information (phone numbers)
Spy Cam	Take and upload photos to a remote server
Call Tracker	Record and upload calls to a remote server
Existing ones	
Exprespam [20]	Steal device info, contacts, and emails
Armor for Android [21]	Steal device (IMEI) information

III. Model and Approach

The proposed activity graph is a visualization to break the "black-box" view. For this purpose, a graphical model is expected. Considering that Android, which is our choice of study platform, is modified based on Linux kernel, it is rational to adopt the graphical model that we proposed in a prior study [18] that involves Linux systems. In this prior study we used the graphical model for a different purpose, i.e. the zero-day attack path identification. However, we can easily update the definition for *per-host* system object dependency graph in the prior study with minor modifications, and get the following Definition 1 which is for *per-application*. From this perspective, the mobile application activity graph is a simplified mutation of the per-host System Object Dependency Graph (SODG) in the prior study.

Definition 1. *Mobile Application Activity Graph (a simplified mutation of Definition 1 in our prior study [18])*
If the system call trace for the mobile application is denoted as Σ, then the activity graph for the application is a directed graph $G(V,\ E)$, where:

- V is the set of nodes, and initialized to empty set $\varnothing$;
- E is the set of directed edges, and initialized to empty set $\varnothing$;
- If a system call $syscall{\in}\Sigma$, and dep is the dependency relation parsed from $syscall$, where $dep{\in}\{(src{\to}sink),\ (src{\leftarrow}sink),\ (src{\leftrightarrow}sink)\}$, src and $sink$ are mobile OS objects (mainly a process, file or socket), then $V = V{\cup}\{src, sink\}$, $E = E{\cup}\{dep\}$. dep inherits timestamps *start* and *end* from *syscall*;
- If $(a{\to}b){\in}E$ and $(b{\to}c){\in}E$, then c transitively depends on a.

Figure 1 illustrates one example activity graph. Compared to per-host SODGs in our prior study, mobile application activity graphs are usually much smaller due to its per-application nature instead of being per-host. As shown in Figure 1, at the center is often a process, which interacts with other system entities (files or sockets). Hence, mobile application activity graphs are also more cognition-friendly than per-host SODGs, which is deemed to be very valuable for practice.

Another benefit of adopting the same graphical model is the reuse of the system call dependency rules

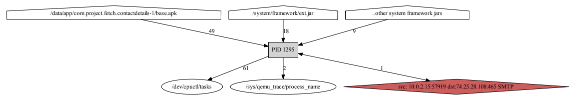
/data/app/com.project.fetch.contactdetails-1/base.apk
/system/framework/ext.jar
..other system framework jars
49
18
9
PID 1295
61
2
1
/dev/cpuctl/tasks
/sys/qemu_trace/process_name
src: 10.0.2.15:57919 dst:74.25.28.108:465 SMTP

Figure 1: An example activity graph for an application named "Data Peek" (tailored due to space constraints), in which a rectangle denotes a process, a diamond denotes a socket, an ellipse denotes a plain file, and a house denotes an Android file. The red color highlights the data leakage. The number on the edge is the number of system calls. Readers need not discern text in small font.

listed in Table 1 of our prior study [18]. The rules are used to parse system calls and generate the directed edges in activity graph, according to Definition 1. For space constraints, the table is not shown here. Considering that a mobile system is less capable and more power-consuming, the Android-based implementation allows user customization to just use a subset of the rules (always including some fundamental ones such as *sys_read*, *sys_write*, *sys_connect* and *sys_accept*, etc.) for system call parsing. The tradeoff is the potential escape of data leakages due to the miss of some system calls.

Definition 1 also defines the steps of the approach for generating the activity graph. Step 1 logs the system calls for the target application. Step 2 generates the activity graph based on the collection of system calls. Step 3 highlights and discerns the socket objects in the activity graph which connects to unexpected IPs.

IV. Implementation

The tool is implemented based on an Android emulator with Android SDK, NDK, and Goldfish Kernel installed. Goldfish Kernel is for supporting Loadable Kernel Module (LKM).

System Call Auditing As the only step that has to run on the fly, we perform system call auditing via Loadable Kernel Module. By locating the system call table and changing the system call's address (changed back when the module exits), the module detours the system calls of our interests to record their parameters and return information. It also preserves the object's identity information from the corresponding data structures, such as *struct task_struct* for the process, *struct files_struct* for the file, and *struct sockaddr_in* for the socket. As a result, the activity graph objects parsed from the collection of system calls could have a name or pathname instead of just a meaningless number. This constitutes another reason why activity graph is cognition-friendly for analysts.

Activity Graph Generation This step is implemented off-line. Taking in the system calls, the parsing process turns them into a set of objects and edges, where the edges connect the objects in a directed fashion. Using the way specified the Definition 1, they can be represented in a *.dot* file defined by Graphviz [22]. Graphviz then could be further used to turn the *.dot* file into a JPG or PDF format picture. In the resultant activity graphs, a rectangle denotes a process, a diamond denotes a socket, an ellipse denotes a plain file, and a house denotes an Android file.

Socket Object Validation This step is also implemented off-line. This step needs to ask the analyst to give a white-list of IP addresses. Then, the white-list is matched with each of the socket objects in the activity graph. In the event that a socket object is associated with an unexpected IP address, an alert is raised and highlighted. For socket object validations, we could also deploy baits in the system, such as a file (a photo, an audio, or a contact) which contains information that should not flow to Internet. In the event that a socket object has one of the baits as its predecessor or ancestor node, an alert is also raised and highlighted, since this indicates that the bait information is being flowed across the network boundary.

V. Evaluation

A. Effectiveness

For evaluation, this paper identified two existing and three home-brewed malicious mobile applications, all of which are used for effectiveness test. The application information is summarized in Table 1. According to Table 1, the tested applications steal different types of data, respectively phone numbers, photos, phone calls, emails, device information and the like. This demonstrates that our approach is effective in overseeing the leakages of a diverse set of data.

Figures 1-5 illustrate activity graphs for the five mobile applications listed in Table 1. Figure 1 shows that the information of */data/app/com.project.fetch.contactdetails-1/base.apk* flows to the tainted socket which leaks data to the remote machine with *IP: 74.25.28.108* and *port: 465* via *SMTP*. This reveals that the contact details are indeed stolen by the application named "Data Peek" via email.

Figure 2 depicts that the application calls */bg_non_interactive/tasks* to run in the background for taking photos. The photo is saved to */storage/0AF1-1D07/Pictures/MYGALLERY/1455508625702.jpg* and also transmitted to the machine with *IP: 10.0.0.3* and *Port: 8888*. This reveals that the candid photos are indeed stolen by the application named "Spy Cam".

Figure 3 illustrates that the application calls */bg_non_interactive/tasks* to run in the background for recording the incoming or outgoing calls when the user receives/makes them. After completing, the audio recordings are converted from RAW to *.wav* format, and transferred to the machine with *IP: 10.0.0.3* and *Port:*

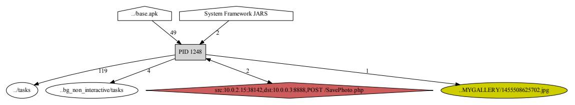
../base.apk
System Framework JARS
49
2
PID 1248
119
4
2
1
../tasks
..bg_non_interactive/tasks
src:10.0.2.15:38142,dst:10.0.0.3:8888,POST /SavePhoto.php
..MYGALLERY/1455508625702.jpg

Figure 2: An activity graph for an application named "Spy Cam" (tailored due to space constraints). Same notations used in Figure 1 are also used here.

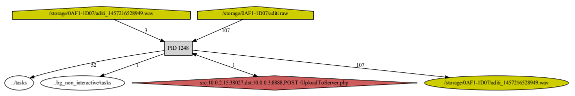
/storage/0AF1-1D07/aditi_1457216528949.wav
/storage/0AF1-1D07/aditi.raw
3
107
PID 1248
52
1
1
107
../tasks
..bg_non_interactive/tasks
src:10.0.2.15:38027,dst:10.0.0.3:8888,POST /UploadToServer.php
/storage/0AF1-1D07/aditi_1457216528949.wav

Figure 3: An activity graph for an application named "Call Tracker" (tailored due to space constraints). Same notations used in Figure 1 are also used here.

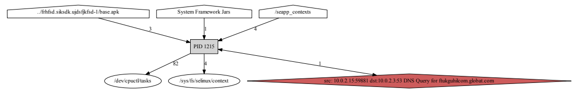
../frhfsd.siksdk.ujdsfjkfsd-1/base.apk
System Framework Jars
/seapp_contexts
3
1
4
PID 1215
82
4
1
/dev/cpuctl/tasks
/sys/fs/selinux/context
src: 10.0.2.15:59881 dst:10.0.2.3:53 DNS Query for ftukguhilcom.globat.com

Figure 4: An activity graph for an application named "Exprespam" (tailored due to space constraints). Same notations used in Figure 1 are also used here.

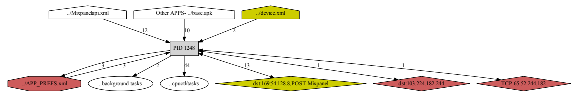

Figure 5: An activity graph for an application named "Armor" (tailored due to space constraints). Same notations used in Figure 1 are also used here.

8888. This reveals that the audio recordings are indeed stolen by the application named "Call Tracker".

Figure 4 shows that the application is trying to query the IP for the site *ftukguhilcom.globat.com*. The site was used for uploading the stolen information by this application when it was discovered in Jan 2013, but is not active right now. This reveals that the application named "Exprespam" tried to steal data, but failed due to unavailability of attacker site.

Figure 5 depicts that the application accesses the device information (IMEI, IMSI, etc) and tried to connect to several sites, but some of them are unavailable now. This reveals that the device information is indeed stolen by the application named "Armor".

B. *Comparison*

This paper also compares our system call-based approach to API call-based approach. For comparison, we implemented the graph generation based on API calls, via application call interception and script-based text mining. Figures 6-8 are the resulting graphs that were generated from the three home-brewed mobile applications. Compared to Figures 1-3 which use system call-based approach, API call-based graphs are more complex and ad-hoc. In contrast, the system call-based activity graphs are concise and natural. In addition, system call-based activity graphs could be used for logical inference as the edges represent causality relationships, where the API call-based graphs are more like profiles merely providing contextual information for analysis.

VI. Conclusion

Inspired by our mobile security survey which attracted over 500 responses and revealed huge risk in mobile data leakages, this paper presents an open-source tool based on system calls to build activity graphs for mobile applications. The activity graph enables the visualization of an application interacting with the mobile system and network, and thus could be leveraged as an assistance facility for mobile security personnel to detect and diagnose covert data leakages caused by the mobile malwares. With Android as our choice of study platform, this paper implements the tool and tests it against two existing and three home-brewed malicious mobile applications. The evaluation results show that our tool is effective in visually revealing the causes and processes of data leakages.

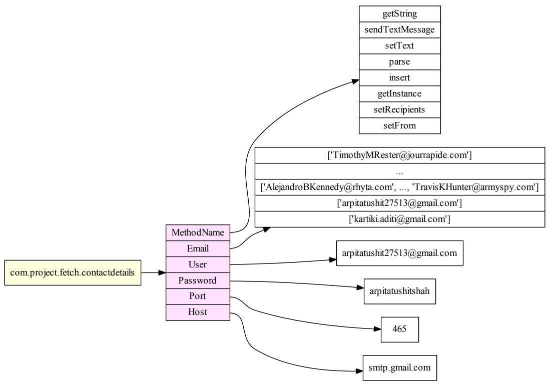
getString
sendTextMessage
setText
parse
insert
getInstance
setRecipients
setFrom
['TimothyMRester@jourrapide.com']
...
['AlejandroBKennedy@rhyta.com', ..., 'TravisKHunter@armyspy.com']
['arpitatushit27513@gmail.com']
['kartiki.aditi@gmail.com']
MethodName
Email
User
Password
Port
Host
com.project.fetch.contactdetails
arpitatushit27513@gmail.com
arpitatushitshah
465
smtp.gmail.com

Figure 6: A application-call graph for an application named "Data Peek", in which a yellow rectangle denotes a project, a pink rectangle denotes an attribute, and a white rectangle denotes a value. Readers need not discern text in small font.

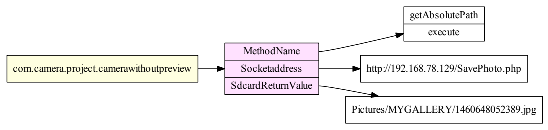
getAbsolutePath
execute
MethodName
Socketaddress
SdcardReturnValue
com.camera.project.camerawithoutpreview
http://192.168.78.129/SavePhoto.php
Pictures/MYGALLERY/1460648052389.jpg

Figure 7: A application-call graph for an application named "Spy Cam". Same notations used in Figure 6 are also used here.

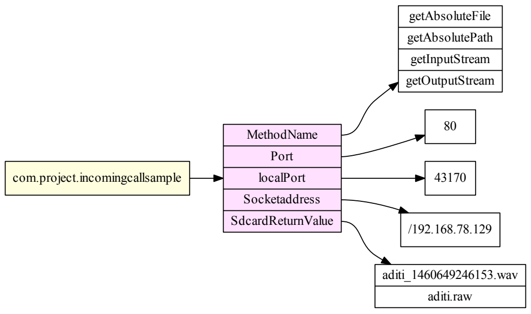

Figure 8: A application-call graph for an application named "Call Tracker". Same notations used in Figure 6 are also used here.

Acknowledgments

We want to thank the anonymous reviewers for their valuable and helpful comments. This work was supported by California State University, Sacramento's UEI (University Enterprises, Inc.) Campus Grant in 2014 for the project titled "An Investigation on Smartphone User Behavior and Its Security/Privacy Implications".

References

[1] Dan Goodin. "Apple scrambles after 40 malicious "XcodeGhost" apps haunt App Store". Ars Technica. Retrieved May, 2017.

[2] Charles Arthur. "Naked celebrity hack: security experts focus on iCloud backup theory". The Guardian. Retrieved May, 2017.

[3] Jeremy Kirk. "Many US enterprises still running XcodeGhost-infected Apple apps, FireEye says". InfoWorld. Retrieved May, 2017.

[4] Swarna Gopalan. "User Behavior Patterns: Human Factors Affecting Mobile Security and Privacy". Master report, California State University, Sacramento, 2016.

[5] Aditi A Kulkarni. "Don't be Surprised: A Demo of Android Applications to Steal Your Mobile Data". Master report, California State University, Sacramento, 2016.

[6] Arpitaben Shah. "Android Malware Detection and Forensics Based on API Calls". Master report, California State University, Sacramento, 2016.

[7] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. "Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. In proceedings of the 14th ACM conference on Computer and Communications Security, ACM Transactions on Information and System Security (TISSEC), 2010.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. 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 ACM Transactions on Computer Systems, 32(2):129, 2014.

[9] V. M. Afonso, M. F. de Amorim, A. R. A. Gregio, G. B. Junquera and P. L. de Geus. "Identifying Android malware using dynamically obtained features". Journal of Computer Virology and Hacking Techniques, 11(1):917, 2015.

[10] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff. "A sense of self for unix processes". In IEEE Oakland, 1996.

[11] W. Lee, S. J. Stolfo, P. K. Chan. "Learning patterns from unix process execution traces for intrusion detection". In AI Approaches to Fraud Detection and Risk Management, 1997.

[12] A. P. Kosoresow, S. A. Hofmeyer. "Intrusion detection via system call traces". In IEEE Software, 1997.

[13] S. A. Hofmeyr, S. Forrest, A. Somayaji. "Intrusion detection using sequences of system calls". In Journal of Computer Security, 1998.

[14] C. Kruegel, D. Mutz, F. Valeur, G. Vigna. "On the detection of anomalous system call arguments". In ESORICS, 2003.

[15] G. Tandon, P. Chan. "Learning rules from system call arguments and sequences for anomaly detection". In ICDM DMSEC, 2003.

[16] S. Bhatkar, A. Chaturvedi, R. Sekar. "Dataflow anomaly detection". In IEEE Oakland, 2006.

[17] S. T. King, P. M. Chen. "Backtracking intrusions". In ACM SOSP, 2003.

[18] Jun Dai, Xiaoyan Sun, and Peng Liu. "Patrol: Revealing zero-day attack paths through network-wide system object dependencies". In proceedings of the 18th European Symposium on Research in Computer Security (ESORICS), 2013.

[19] Xiaoyan Sun, Jun Dai, Peng Liu, Anoop Singhal, John Yen. "Towards Probabilistic Identification of Zero-day Attack Paths". In CNS, 2016.

[20] Vogella. "Android Camera API - Tutorial". Retrieved July, 2017.

[21] Tutorialspoint. "Android Camera Tutorial". Retrieved July, 2017.

[22] Graphviz. Graph visualization software. http://www.graphviz.org/. Retrieved July, 2017.