User-Centric Dependence Analysis For Identifying Malicious Mobile Apps

Karim O. Elish, Danfeng (Daphne) Yao, and Barbara G. Ryder
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
Email: {kelish, danfeng, ryder}@cs.vt.edu

***Abstract*—This paper describes an efficient approach for identifying malicious Android mobile applications through specialized static program analysis. Our solution performs offline analysis and enforces the normal properties of legitimate dataflow patterns to identify programs that violate these properties. To demonstrate the feasibility of our user-centric dependence analysis, we implement a tool to generate a data dependence graph and perform preliminary evaluation to characterize both legitimate and malicious Android apps. Our preliminary results confirm our hypothesis on the differences in user-centric data dependence behaviors between legitimate and malicious apps.**

I. Introduction

The proliferation of mobile handheld devices has led to the development of a large number of mobile applications provided by many application markets. Unfortunately, the user may download and install some applications which contain Trojans and malware, and consequently these malicious apps can affect the security of her/his mobile device. Therefore, there is a need to find an approach to detect the malicious apps before the user installs them.

In this work, we address the important problem of malware classification, that is, given an unknown program, how to determine whether or not it is malicious software (malware). The novelty of our work is that we take the approach of anomaly detection (i.e., identifying deviations from normal patterns), as opposed to the conventional methods of identifying malware characteristics. We aim at strategically enforcing the normal properties of legitimate dataflow patterns and identifying programs that violate these properties. We extract these properties by tracking the dependence between the definition and use of user-generated data in programs. Our approach is complementary to the existing classification approaches based on identifying known malicious code or behavior signatures, such as [1], [2], [3].

Our approach focuses on analyzing the relations between user inputs/actions and entry points to methods providing critical system functions. Hence, what distinguishes our work is that we uniquely integrate user data dependence in our analysis. Our technical contributions are summarized as follows.

1) We demonstrate the use of dependence analysis for identifying malicious Android mobile applications. In particular, we implement our analysis tool to statically construct data dependence graphs with inter-procedural call connectivity information [4] that capture the data consumption relations in programs through identifying the directed paths between user inputs (e.g., data and actions) and entry points to methods providing critical system services.
2) We use our tool to conduct an initial set of experiments to characterize the data consumption behaviors of legitimate and malicious Android apps, specifically on how they respond to user inputs and events.

This work has been supported in part by Security and Software Engineering Research Center (S^2ERC), a NSF sponsored multi-university Industry/University Cooperative Research Center (I/UCRC).

II. Overview of Our Approach and Threat Model

Our goal is to correlate critical system function calls with user-initiated input events through programs in order to identify software execution characteristics of these relations. The technical challenge associated with our anomaly detection approach is that the (normal) behaviors of legitimate programs are diverse and difficult to define. We address the challenge through analyzing user-centric data dependence by constructing the data dependence graph for the whole program.

Definitions. In the program analysis literature, a data dependence graph (DDG) is a directed graph representing data dependence between program points, where a node represents a program point (e.g. assignment statement), and an edge represents a data dependence between two nodes. The data dependence edges are identified by data-flow analysis. A direct edge from node n_1 to node n_2, which is denoted by $n_1 \rightarrow n_2$, means that n_2 uses the value of variable x which is defined by n_1.

We define our terminology used on the DDG as follows:

- ***User inputs/actions (source)***: refers to the user's inputs or actions to the program through the input devices (e.g. keyboard).
- ***Sensitive function calls (sink)***: refers to entry points to methods providing critical system functions such as network I/O, file I/O, telephony services.
- ***Data dependence path***: refers to a directed path that captures the dependence relations between a source and a sink.

The motivation behind proposing our approach is based on our following key observations: *i)* legitimate critical system events are typically initiated by user inputs/actions, and *ii)*

Android mobile applications require a lot of user interactions. Therefore, we hypothesize that the legitimate function calls are triggered by inputs/actions from authorized users.

In our user-centric data dependence graph, the sensitive function call is considered legitimate if there is a directed path that represents a dependence relation between this function call and user inputs/actions. Otherwise, the call to the sensitive function is considered suspicious. We use this basic dependency rule for malware classification. The procedure we used to identify the data-dependence path in a given program is described as follows.

1) Identify sensitive functions that perform any critical operations such as network I/O operations, and file I/O operations (e.g., write, delete).
2) Identify any inputs/actions provided by the user to the program.
3) Construct the data dependence graph for the whole program.
4) Identify the data-dependence paths between user inputs and sensitive functions according to the dependency rule by using the generated data dependence graph to track the dependences between the definition and use of user-generated inputs.

The procedure described above can be used to enforce other more complex dependency rules as well. The use of static program analysis for approximating the data dependence of sensitive library operations on user data, to identify dataflow associated with user inputs has not been explored previously as a general approach for malware identification.

Threat model. We consider stealthy malware that can be installed with the installation of malicious Android apps on the mobile device. Malware that runs as a user-level application can perform botnet command & control, attacks, data exfiltration, or abuse system resources. Data exfiltration refers to secretly exporting sensitive information such as personal data without the permission of the user. There are two cases of Android malware, namely *i)* malware that is packaged or added as add-on component to an existing legitimate app such as *GoldDream* and *GGTracker.A*, and *ii)* malware that is a stand-alone user-level app such as *Fake Netflix, Walk & Text*, and *Dog Wars*.

III. Construction of Data Dependence Graph

A. Implementation Details

In order to obtain user-centric data dependences, we need to track the dependences between the definition and use of user-generated data in programs. Hence, we developed automatic tool based on Soot (a static analysis toolkit for Java) [5] for obtaining data-dependence analysis. We utilized the def-use structures provided by Soot. Unfortunately, it does not provide inter-procedural call information. Thus, we implemented our own program to augment the def-use relations across the boundaries of methods. We use GraphVis package with our implementation to generate the graphical representation of program data dependences. Our tool can analyze Java bytecode or source code and statically construct data dependence graphs with inter-procedural call connectivity information that captures the data consumption relations in Java programs through identifying the directed paths between user inputs (e.g., data and actions) and entry points to methods providing critical system services. In addition, our tool provides context-sensitive data-flow dependence analysis. The context-sensitive analysis considers the context of the caller function when analyzing a callee function. In particular, it differentiates multiple function calls of the same function with respect to provided arguments. On the other hand, a context-insensitive analysis does not differentiate multiple calls of the same function with different arguments. Thus, a context-insensitive analysis does not provide as accurate an analysis and may increase the false positive rates. Currently, our implementation can handle intra-application activity-related Intents for Android apps. However, we plan to enhance our tool to handle inter-application Intents to provide more robust analysis.

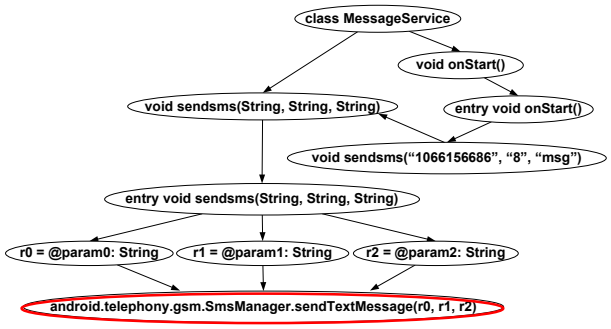

Fig. 1. Partial data dependence graph for HippoSMS malware. The sensitive function `sendTextMessage()` is called without any user inputs/actions, and hence there is no direct path showing the data dependence between user inputs and this sensitive function.

B. An Example

In this section, we provide an example of our analysis using real Android malware called HippoSMS which is found in Chinese App Markets. This malware infects Android smartphones by registering to premium SMS service. It sends SMS messages to a hard-coded premium-rated number without the user's permission. As a result, it will cost the user some additional phone charges. Furthermore, it automatically removes SMS from legitimate mobile service providers to hide the additional charges from the users. Figure 1 depicts an example of partial data dependence graph generated using our tool for HippoSMS. In this example, we can observe that `onStart()` method calls `sendSMS(p1, p2, p3)` method with a hard-coded premium-rated number as a `p1` parameter once the app launches. The `sendSMS` method, in turn, calls a sensitive function `sendTextMessage(phoneNum, scAddress, msg)` with the value of its `p1` (i.e., hard-coded premium-rated number) as a `phoneNum` parameter. The user does not provide any input or action in order to call this `sendTextMessage` method to send an SMS. This means that the malware calls `sendTextMessage` method to send a predefined message to this hard-coded premium-rated number

without the user's permission. In this case, we can not identify any directed paths between user inputs (e.g., data and actions) and `sendTextMessage` method, and hence we can infer that this app performs malicious activity.

IV. Preliminary Results and Evaluation Plan

We have performed preliminary studies using our tool on several real legitimate and malicious Android apps found in official or alternative Android markets such as the Chinese Android market, and obtained promising results that are shown in Table I. We compared user-related data dependences in those apps, namely the number of user inputs/actions taken, the number of sensitive function calls, and the percentage of sensitive function calls that are not dependent on any user inputs (i.e., there does not exist a data-dependence path between the user input and the sensitive function invocation). We identified sensitive function calls after examining the apps to be those that utilize system resources such as network I/O, file I/O, GSM-specific telephony services. For example, `sendTextMessage()` provided by *android.telephony.gsm.SmsManager* library for sending SMS, and `openFileOutput()` provided by *android.content.Context* library for opening and writing to a file.

Discussion. Table I summarizes the results of our preliminary evaluation. We found that in all legitimate apps, all function calls depended on user inputs (i.e., the user needs to enter certain information before the request to the call is made). In most of the malicious Android apps, this property of data dependence is not observed; the apps abuse the system resources without user's authorization confirming our hypothesis on the differences between user-centric data-dependence behaviors of legitimate and malicious programs.

As shown in Table I, the Fakeneflic malware is a phishing app that tricks the user to enter their Netflix login, which behaves similarly to a legitimate app in terms of user-related data dependence. This type of malware behavior can circumvent our approach and lead to false negatives because of the existing dependency path between user inputs and sensitive function calls. Detecting it requires site authentication (i.e., certification verification) and user education. Some malware may attempt to forge user input events in order to trick our approach. Nevertheless, the input-forgery attack can be prevented by existing solutions such as [6].

Limitations. Some malware may attempt to circumvent our data dependence checking. One possible attack scenario is where the malware may require superfluous user inputs and actions (before making sensitive function calls to conduct unauthorized activities) attempting to satisfy the dependency, but the user inputs are not consumed by the calls. This type of data-flow can be detected by our current method by tracking the dependency between the user inputs entered and the sensitive function calls, thus the malware can be identified. The more challenging scenario is where the malware misuses the user inputs while performing malicious activities. For example, in the malware the user inputs are appended to malware's own arguments and are consumed by the sensitive function call, which still satisfies our current data dependence policy. We plan to address this problem by performing in-depth semantic-level data dependence analysis in the future.

Some legitimate sensitive function calls can be triggered without direct user inputs/actions. For example, the user may accept the prompt for OS updates, which can increase false positives. This can be considered as a limitation of our current implementation and it needs more investigation to find the dependency relationship.

Currently, our tool performs static analysis. However, some malicious apps can use obfuscation or Java reflection techniques to evade the detection which may lead to false negatives as well as false positives. In order to overcome this problem, we need to use dynamic taint analysis [3], [7] to provide insights about the program's runtime execution. Hence, we plan to investigate a combined static and dynamic tool in future research to mitigate this problem.

We plan to perform more evaluation on our user-centric data dependence solution with different types of apps behaviors and different categories of malware apps to identify user-centric policies to be used in our malware classification.

V. Related Work

User-centric based security has not been explored in the literature as a general approach for malware detection. In general, all existing malware detection solutions aim at detecting characteristic malware behaviors [1], [2], [3] or signatures in binary code such as commercial anti-virus scan tools. For example, Christodorescu et al. [2] extracted malware specifications by comparing the behavior of malware against the behavior of benign programs, and used the extracted knowledge to detect the variants. In Panorama [3], whole-system taint analysis is performed to characterize malicious activity to detect malware. However, malware patterns are constantly evolving which results in an endless race, as the existing solutions need to be updated rapidly to accommodate the evolving of malware patterns. Our work uses an anomaly detection approach and focuses on enforcing correct data-flow patterns in legitimate programs and identifying programs that violate these properties, as opposed to chasing evolving malware patterns.

A number of static analysis techniques have been used for malware detection by analyzing the source code [8], [9] or the binary [10], [11], [12] to provide insights on the intended control flow, system call context, and call dependences of a program. For example, Bhatkar et al. [9] proposed an approach for anomaly detection based on analyzing the data flows involving system call arguments through the program. However, these works do not consider user-centric data dependences in their analysis to capture the casual relations between user inputs/actions and system function calls compared to our work.

In the mobile OS security literature, a lot of researchers have studied Android OS platform security [13], [14], [15], [16], [17], [18]. However, all these works aim to protect and improve Android OS in general, and do not provide a solution for detecting malicious Android apps compared to our work.

	App/Malware Name	# of User Inputs/ Actions (Source)	# of Sensitive Function Calls (Sink)*	% of Sensitive Func. Calls without User Inputs
Legitimate	SendSMS	3	1	0%
	BMI Calculator	2	1	0%
	BluetoothChat	2	1	0%
	SendMail	4	1	0%
	Tip Calculator	4	1	0%
Malware	GGTracker.A: sends SMS log to C&C server	0	1	100%
	HippoSMS: sends SMS to premium-rated number	0	3	100%
	Fakeneflic: steals Netflix user's account info	3	1	0%
	GoldDream: steals user's data	0	2	100%
	Walk & Text: sends SMS to all contact list	0	3	100%
	RogueSPPush: subscribes to premium SMS services	0	3	100%
	Dog Wars: sends SMS to all contact list	0	2	100%

*Sensitive function names are not shown in the table

TABLE I
PRELIMINARY CHARACTERIZATION RESULTS FOR STUDIED LEGITIMATE AND MALICIOUS ANDROID APPS

In the malware apps detection literature, Dixon et al. [19] proposed an approach to correlate power consumption patterns with the user's location in order to detect malicious code. Liu et al. [20] detected malicious behaviors on mobile devices by monitoring abnormal power consumption caused by malware. The work in [19] and [20] detect malware apps on mobile devices, while our work performs offline analysis to detect malware apps available in the markets.

VI. Conclusion and Future Work

We proposed and implemented an efficient approach for malware identification based on user-centric data dependence analysis to capture the data dependences between user inputs and critical function calls. Our user-centric dependence analysis allows the detection of suspicious Android mobile apps and Java programs that violate the data dependences. For future work, we plan to experimentally evaluate and characterize more data-consumption behaviors of both legitimate and malicious Android apps and Java programs, and extract policies for classifying them. Additionally, we plan to utilize more accurate program analysis techniques such as blended program analysis [21].

References

[1] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proc. of the 12th conference on USENIX Security Symposium*, 2003, pp. 169–186.

[2] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2007, pp. 5–14.

[3] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *ACM Conference on Computer and Communications Security*, 2007, pp. 116–127.

[4] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26–60, 1990.

[5] "Soot: a Java optimization framework," http://www.sable.mcgill.ca/soot/.

[6] K. Xu, H. Xiong, C. Wu, D. Stefan, and D. Yao, "Data-provenance verification for secure hosts," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, pp. 173–183, 2012.

[7] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of the Network and Distributed System Security Symposium*, 2005.

[8] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proc. of IEEE Symposium on Security and Privacy*, 2001, pp. 156–68.

[9] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proc. of IEEE Symposium on Security and Privacy*, 2006, pp. 48–62.

[10] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity: principles, implementations, and applications," in *Proc. of the 12th ACM conference on computer and communications security*, 2005, pp. 340–353.

[11] J. Giffin, S. Jha, and B. Miller, "Efficient context-sensitive intrusion detection," in *Proc. of the Network and Distributed System Security Symposium*, 2004.

[12] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *Proc. of IEEE Symposium on Security and Privacy*, 2004.

[13] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *USENIX Symposium on Operating Systems Design and Implementation*, 2010, pp. 393–407.

[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proc. of the 9th Int'l Conference on Mobile Systems, Applications, and Services*, 2011, pp. 239–252.

[15] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 627–638.

[16] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. of the 20th USENIX conference on Security*, 2011.

[17] W. Enck, M. Ongtang, and P. D. McDaniel, "On lightweight mobile phone application certification," in *Proc. of the ACM Conference on Computer and Communications Security*, 2009, pp. 235–245.

[18] M. Ongtang, K. R. B. Butler, and P. D. McDaniel, "Porscha: policy oriented secure content handling in android," in *26 Annual Computer Security Applications Conference*, 2010, pp. 221–230.

[19] B. Dixon, Y. Jiang, A. Jaiantilal, and S. Mishra, "Location based power analysis to detect malicious code in smartphones," in *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011, pp. 27–32.

[20] L. Liu, G. Yan, X. Zhang, and S. Chen, "VirusMeter: Preventing your cellphone from spies," in *Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009, pp. 244–264.

[21] B. Dufour, B. G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive java applications," in *Proc. of the 16th ACM SIGSOFT Int'l Symposium on Foundations of software engineering*, 2008, pp. 59–70.