2012 Third World Congress on Software Engineering

LeakMiner: Detect information leakage on Android with static taint analysis

ZheMin Yang
Parallel Processing Institute
Fudan University
Shanghai, China

Min Yang
Parallel Processing Institute
Fudan University
Shanghai, China

***Abstract*—With the growing popularity of Android platform, Android application market becomes a major distribution center where Android users download apps. Unlike most of the PC apps, Android apps manipulates personal information such as contract and SMS messages, and leakage of such information may cause great loss to the Android users. Thus, detecting information leakage on Android is in urgent need. However, till now, there is still no complete vetting process applied to Android markets. State-of-the-art approaches for detecting Android information leakage apply dynamic analysis on user site, thus they introduce large runtime overhead to the Android apps.**

This paper proposes a new approach called LeakMiner, which detects leakage of sensitive information on Android with static taint analysis. Unlike dynamic approaches, LeakMiner analyzes Android apps on market site. Thus, it does not introduce runtime overhead to normal execution of target apps. Besides, LeakMiner can detect information leakage before apps are distributed to users, so malicious apps can be removed from market before users download them. Our evaluation result shows that LeakMiner can detect 145 true information leakages inside a 1750 app set.

***Keywords*-Information security; Mobile computing**

I. Introduction

Android operating system becomes more and more popular these years. With the growing number of Android apps, users can download various apps from Android markets. However, downloading an un-verified app from market is not safe, because malicious apps can load sensitive information without user notification and send it to unauthorized third party groups. Unlike apps in PC, users commonly store person information like SMS messages, contracts, and calender on smart phone. Study[2] shows that sensitive information are sent to unknown destination without user awareness. Thus, privacy and security of personal data is becoming more and more important. Although Google reports its application validation process called Google Bouncer, reports[1] show that it is still incomplete and may miss leakage apps because of its behavior-based analysis approach. State-of-the-art leakage detection tools[5], [6] for Android use dynamic taint propagation to detect information leakages at runtime. However, such approaches detect information leakage at user site and cannot filter malicious apps before they are distributed to Android users. Thus, Android markets urgently need a process to detect and remove malicious apps on market site.

This paper propose an approach to identify and remove information leakage cases on market site, called LeakMiner. To identify possible information leakage, LeakMiner applies a static taint analysis to apps within Android market. Our approach introduces three steps in identifying possible leakages: first, apk files of Android apps are transformed to Java bytecode so that the following analysis can directly work on Java bytecode. Besides, application metadata are extracted from the manifest file of Android app. Then, LeakMiner identifies sensitive information according to the extracted metadata. Finally, taint information is propagated through call graph to identify possible leakage paths. By introducing multiple entry point call graph, we can cover all the code of Android app. We choose a set of 1750 apps to evaluate the accuracy of LeakMiner. LeakMiner can identify 145 real leakages in this app set. Among the evaluated cases, 160 benign apps are false alarmed as possible leakage, and we show that false positives of LeakMiner are mainly caused by long propagation path of Device ID in Android apps.

II. Overall Architecture

This section describes the overall architecture of LeakMiner. As depicted in Figure 1, LeakMiner detects information leakage with three steps:

1) **Preprocessing.** Because of the limited resources in smartphones, Android applied a different format from Java bytecode, call DEX bytecode, to save space. In order to analyze Android apps, we first transform DEX bytecode back to Java bytecode, which can be analyzed by state-of-the-art static analysis frameworks for Java. Besides, we also extract application metadata, like permissions, from manifest files to help identify sensitive data in the next step.
2) **Sensitive data identification.** Android apps load sensitive data by predefined interfaces. Although many Android interfaces can load sensitive information from smartphone, only a small part of API calls are allowed in each app. Whether an app can use a certain API depends on the permission granted to this app. By using the application metadata extracted in the preprocessing phase, we can filter irrelevant API calls

978-0-7695-4863-0/12 $26.00 © 2012 IEEE
DOI 10.1109/WCSE.2012.26

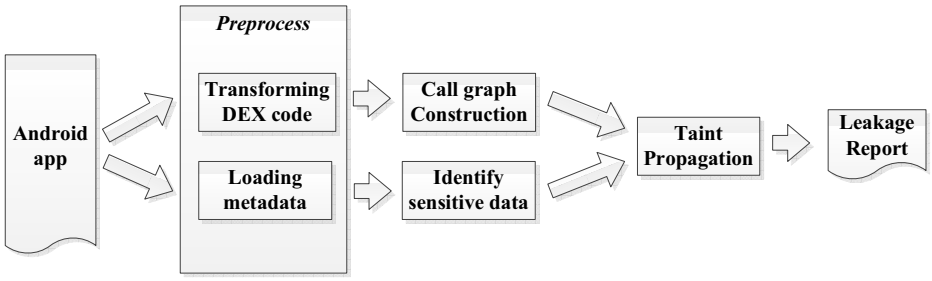

Figure 1. Overall Architecture of LeakMiner

immediately. We search each interface in the pre-defined permission-interface mapping, and only the permitted interfaces will be analyzed in the next step.

3) **Information flow propagation.** For each of the sensitive source point, we statically analyze the possible propagation paths. A data flow analysis technique is applied to find all instructions which depend on this sensitive source point. If sensitive data is propagated to network or local logging system, a leakage path is identified and reported to user.

III. Android Information Leakage Detection

In this section, we describe our design of LeakMiner. Section 3.1 introduces the sensitive information tracked by LeakMiner. Section 3.2 describes how we construct call graph for Android apps and which pointer analysis approach is applied to it. Finally, our taint analysis approach is depicted in Section 3.3.

A. Identify sensitive information

In LeakMiner, the following contents are considered as sensitive information:

1) **Unique Device ID.** Device ID of smartphones is the unique identifier of each devices. It can be used to locate certain smartphone user because of its immutable feature. Besides, because of its uniqueness, it is also used in apps to bind user accounts. Thus, it is dangerous if device ID is sent to unauthorized users together with other user information. In this paper, we classify both IMEI number and subscriber ID as device ID. Android apps can read device ID from smartphone if READ_PHONE_STATE permission is granted.
2) **Location.** Location information represents the current location of Android user. Malicious apps can use user location and device ID to track the current location of Android user. In Android, two permissions, which are ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION, are necessary for apps to obtain location data.
3) **Phone number.** Like device ID, phone number of Android user can also be used to identify certain Android user. LeakMiner marks both phone number and SIM number in Android as sensitive personal information if permission READ_PHONE_STATE is granted to the corresponding app.
4) **Contract book.** Android apps read contact book content if it is granted the READ_CONTACT permission.
5) **SMS message.** SMS texted messages received by smartphone or messages written and sent from local phone contain most of the personal information of smartphone user. Thus, leakage of SMS message is most dangerous to users. Android apps can read such messages if they are granted the READ_SMS permission.
6) **Calendar.** Smartphone users record their personal schedule in the calender, malicious apps can obtain such personal schedule by reading calender content if permission READ_CALENDAR is granted.

B. Call Graph Construction and Pointer Analysis

Our framework employs on the fly call graph construction introduced by Jeffrey, et.al. [3]. Unlike others, Android apps do not contain the root function, which is usually named the main function in C/C++ or Java. Instead, Android apps may have multiple entry points, pre-defined by Android activity and service interfaces. Thus, our static taint analysis first constructs call graphs which starts at these entry points. A new root function node is then used to link these call graphs by building function call edges from the root node to each of the entry nodes. The entry points we tracked are listed below:

- **Fundamental activity lifecycle callbacks.** Android provides six basic lifecycle callbacks (*onCreate*, *onStart*, *onResume*, *onPause*, *onStop*, and *onDestroy*). Any Android activity can override these callback functions to do appropriate work, and they are hooks that are invoked by Android activity manager when the state of activity changes. These callback functions are fundamental activity entry points.
- **Activity supplementary callbacks.** Fundamental activity lifecycle callbacks are invoked each time the state of activity changed. Besides, Android provides supplementary hooks which assist the function of Android

system resource management. For example, when the system destroys an activity in order to recover memory, *onSaveInstanceState* is invoked to save the current state of activity. A corresponding *onRestoreInstanceState* hook is invoked to restore the state when user navigates to the activity.

- **Basic service lifecycle callbacks.** Like activities, Android services also have life callbacks which are automatically invoked by Android service manager (*onStartCommand*, *onBind*, *onCreate*, and *onDestroy*). These callback hooks are the basic entry points for analyzing the service application behaviors.
- **Service supplementary callbacks.** Service supplementary callbacks are usually invoked when the configuration is changed(eg. *onConfigurationChanged*), or when the resources are used up(eg. *onLowMemory*). These functions are infrequently invoked, and our analysis includes such interfaces for the sake of code coverage.

In Android apps, if activity and service entry points are overridden, they will be called automatically when the state of application is changed. On the other side, event handler can be triggered if the corresponding event listener is registered. The event handler can also be the entry point if the corresponding listener is registered in the application. Thus, once LeakMiner encounters a event listener registration, it will add the corresponding handler to the call graph, and incrementally re-construct it. Besides, callback function is also tracked if its owner component is attached to the application.

For all the reachable methods, pointer analysis based on Spark [7] is applied to generate pointer information. This information is organized as summary information for each method. Thus we can directly use this information during taint propagation, instead of repeatedly applying them. Due to the specific calling convention of Android application, context insensitive analysis suffers huge false positives in identifying source and sink methods. For example, Android apps can get a name from local database through *Settings.System.getString* method, but device identifier is returned if and only if a name *android_id* is queried. Thus, the pointer analysis we applied adds one more level of *string* context information to function calls which relate to source and sink points.

C. Thin Slicing and Taint Propagation

After the call graph is constructed and pointer information is attached to each method, LeakMiner figures out data dependencies between source and sink points in a similar way as Thin Slicing introduced by Sridharan et al. [8].

Thin slicing generates a set of nodes which are connected by data dependencies. A statement l is data dependent on statement s when:

1) l can read from storage location o.
2) s can write to o.
3) There exists a control-flow path from l to s on which o is not re-defined.

In thin slicing, only data dependencies are considered, so that implicit information flow leakage is not supported in LeakMiner. Fortunately, unlike server apps, implicit information flow leakage in mobile apps are rare, and till now, we still do not find any such cases. Base pointer flow dependencies, which are data dependencies "due solely to the use of a pointer in a field dereference", are also removed in thin slicing. For example, statement a = b.f is treated as a read operation to b.f, but it is not treated as a read to b because the read to b is solely a field dereference.

In our framework, thin slicings are generated on each source point, then we traverse each slicing to see whether the tainted information is propagated to the sink points.

As described in Section 3.1, before analysis, permission required by analyzed application is first extracted from the manifest file, and only the API interfaces which relate to the required permissions are tainted during analysis. Android permission model prevents apps from acquiring unauthorized sensitive information, thus there is no need to analyze the corresponding program code. This optimization can reduce the performance overhead caused by slicing irrelevant resources.

We currently track leakage of device ID, location, phone information, contact, SMS message, and calendar information. To the best of our knowledge, device ID, location, and phone information are most likely to be leaked.

Previous works on taint analysis for Android mostly focus on transmitting information out the network interface. Our work argues that logging sensitive information into local logging system is also dangerous. Because Android logging system does not distinguish log files of the owner activity from that of other activities, the logged information can be read by any hostile activities if the READ_LOG permission is granted to them. Thus, we track sink points which transmit data out to the logging system as well as network interface in our framework.

IV. Evaluation

In this section, we evaluate the accuracy of LeakMiner. We use an Intel Xeon machine with 2 eight-core 2.0Ghz CPUs and 32 GB physical memory, which runs a Debian Linux with kernel version 2.6.32. LeakMiner is implemented on Soot, and about 1750 apps from Android Market are used to evaluate the soundness and completeness of LeakMiner.

A. Accuracy

As depicted in Figure I, 305 cases are reported by LeakMiner as possible information leakage cases. We manually checked the reported cases and found that about half of them are true leakages. False positives are introduced mainly by the long propagation path inside apps. We found that about 40 of the false positives were caused due to the insufficient

context information during our taint analysis, they can be eliminated by applying a context sensitive approach. Most of the remaining false positives are caused because they involves some debugging assistant code. Developers may write such code to aid the debugging of Android apps and wrap them inside a condition like *if(DEBUG)* or *isEmulator()*. After the app is distributed, such code will not be executed because the corresponding flag is set in release mode. Dead code elimination can filter part of these false positives.

Leakage Source	LeakMiner Report	True Leakage
Device ID	278	127
Phone Information	53	50
Location	35	27
Contacts	12	12
Total Detected	305	145

Table I
INFORMATION LEAKAGE REPORTED BY LEAKMINER. ABOUT HALF OF THE REPORTED INFORMATION LEAKAGE ARE TRUE INFORMATION LEAKAGE.

B. Analysis Time

Our static taint analysis costs about 75 hours to analysis all the 1750 apps. Because each application costs about 2.5 minutes on average, the analysis time is negligible to the Android markets. Besides, the execution time can be further reduced by distributing the analysis workload to multiple machines. Unlike dynamic approaches, because running on the market site, LeakMiner implies no runtime overhead to users' execution.

V. RELATED WORK

LeakMiner detects information leakage of Android apps by a static taint analysis approach. The design of LeakMiner focuses on identifying information leakage apps on the Market side. State-of-the-art information leakage detection approaches mostly focus on the user side. TaintDroid[5] proposes to detect unauthorized information leakage through dynamic taint propagation. It detects possible leakages after Android apps are deployed on user sites. However, user site approaches introduce overhead to the execution of Android apps. Moreover, detecting leakage of sensitive data on user site is dangerous because app users have little knowledge about the security issues and it is difficult to understand the generated reports. AppFence[6] enhanced this approach by automatically blocking possible leakages, but it also blocks benign operations and make Android apps not functional.

To the best of our knowledge, LeakMiner is the first static taint analysis approach which works on Android. TAJ[9] proposes static taint analysis for Java apps and uses a set of bounded analysis techniques to speedup analysis for large-scale industry apps, and PiOS[4] uses a similar taint analysis to detect information leakage in iOS apps. However, their technique can not be applied to Android apps, because Android apps follow an unique event-triggered application model, which is handled by LeakMiner. AdRisk[10] uses a reachability analysis to detect possible information leakage in Android. However, because it does not track the data flow in Android apps, AdRisk introduces huge false positives.

VI. CONCLUSION

This paper presents our LeakMiner which detects information leakage in Android apps. LeakMiner uses static taint analysis to detect possible leakage paths, we apply a special call graph construction algorithm because Android apps follow an unique event-based application model. We sketched the high-level design of LeakMiner and discussed challenges and design for each phase of LeakMiner. LeakMiner can detect 145 true information leakage from the chosen Android apps and produce about 160 false positives. We analyzed the cause of false positives and found that they are mainly caused by insufficient context information. Applying context sensitive analysis can eliminate these false positives, but it will also slowdown the analysis process.

REFERENCES

[1] Android and security, feb 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[2] Your apps are watching you. http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. ECOOP'95*, pages 77–101, 1995.

[4] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *Proc. NDSS'11*, 2011.

[5] W. Enck, P. Gilbert, 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. OSDI'10*, pages 1–6, 2010.

[6] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proc. CCS'11*, pages 639–652, 2011.

[7] O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In *Proc. the 12th international conference on Compiler construction*, CC'03, pages 153–169, 2003.

[8] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proc. PLDI '07*, pages 112–122, 2007.

[9] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *Proc. PLDI '09*, pages 87–97, 2009.

[10] J. Xuxian, G. Michael, S. Ahmad-Reza, and Z. Wu. Unsafe exposure analysis of mobile in-app advertisements. In *ACM Conference on Wireless Network Security*, WiSec '12. ACM, Apr 2012.