International Journal of Hybrid Information Technology
Vol.9, No.3 (2016), pp. 199-210
http://dx.doi.org/10.14257/ijhit.2016.9.3.19

Efficient Privacy Leakage Discovery for Android Applications Based on Static Analysis

Songyang Wu, Yong Zhang and Xiong Xiong

The Third Research Institute of Ministry of Public Security, Shanghai 201204, China.
zhanyonglab@yeah.net

Abstract

Android smart phones often carry personal sensitive information, which makes Android a tempting target for malwares. Recent studies have showed that Android applications frequently are over privileged and the risk of personal privacy leakage is very high. With known Android statics security analysis techniques in literatures, due to lack of considering the control flow between components, the static analysis of sensitive data transmission paths often costs a larger computation overheads, static analysis methods have to make a trade-off between computing time and the precision of analysis results. In this work, we propose a static analysis framework to discovery the sensitive data propagation paths and extract execution conditions (including data inputs and events inputs) of these paths. Our approach first extract a asynchronously executing events sequence graph that directly handles inter-components control flows, it then can be used to archive higher efficient taint analysis when the data propagation path asynchronously cross the boundaries of multiply components. The represented analysis results (data and events inputs) will make the analyst easier to determine if the sensitive data transmission is really a privacy leakage. We present an evaluation with a typical Android malicious app. The result of case study shows that our scheme can effectively help discover the privacy leakage behaviors in the malicious apps.

Keywords*: Privacy leakage, Static Analysis, Android Security*

1. Introduction

With the growing market share and popularity of Android platform, the number of android applications (apps for short) which are available to users has increased dramatically. It is estimated that 44.62% of the world's smartphones run Android as of July 2014(http://www.netmarketshare.com), with over 900,000 applications available on the official Google application store. While users enjoy the rich features of the apps, their sensitive personal data stored in their smart phone, such as phone numbers, short messages (SMS), locations, and other privacy information, may be stolen by malicious apps. Recent studies showed that Android apps frequently are over privileged [1] and may transmit private data to unknown destinations without user consent [2]. To protect users, there is a great need for sound analysis tools that can identify and defend against malicious apps.

To detect malicious behaviors in Android apps. Some literatures [3, 4] focus on solutions to construct effective behavior representations. They mostly use system calls to depict software behaviors because the usage of system API captures the characteristics of the interactions between an app and the Android system. However, system calls do not appear to be a good standard for judging malicious behaviors [5] because it is hard to distinguish the high privileged apps from malicious apps. A rich approaches focus on detecting the sensitive data transmission, i.e., whether personal data leaves the device [6, 7]. However, benign Android apps with maps and social networks normally need to

ISSN: 1738-9968 IJHIT

collect personal data such as device information, location, contact, Apps of cloud storage even send out personal documents to remote cloud servers. These benign apps and malicious apps that steal user data often exhibit the same behavior, namely transmitting private data to outsider. Therefore, what constitutes a privacy leakage by mobile applications is a subject that needs reconsideration [8]. In Yang, *et al.*, work [8], they determine that whether sensitive data transmission is a privacy leakage or not actually depends on whether the transmission is user intended or not. Zhang, *et al.*, work [5] selectively inserts instrumentation code into the app to keep track of private information and detect leakage at runtime. The users can get a notification when some suspicious behaviors occur and make a proper decision (allow or deny this transmission request). The two above schemes leverage the dynamic analysis to help to make a judgment. However when user authentications are required in dynamic executions, the analysis is hard to continue.

In this paper, we propose a static analysis system to provide a human analyst with the execution conditions while a sensitive data transmission occurs. Our approach, first extract a asynchronously executing events sequence graph that directly handles inter-component control and data flows, it can be used to archive higher accuracy and efficient taint analysis when the data propagation path asynchronously cross the boundaries of multiply components. After a static taint analysis of the private data transmission, the system conducts a symbolic execution to obtain the path conditions of sensitive data propagation path. The path conditions are always composed of: a) Data inputs which contain data inputs from outside; b) Event inputs that trigger the sensitive data transmission. Finally, the presented critical information (data and events input) will make the analyst easier in determining if the sensitive data transmission is really a privacy leakage.

2. Related Works

Static Taint Analysis [9] focuses on identifying the possible privacy leakage path with the help of reachability analysis and program slicing. On the other side, Dynamic Taint Tracking techniques [7] track the sensitive data at runtime by instrumenting profiling code to the original app code. They report leakage only if such dangerous propagation happens to occur in the execution. However, these approaches commonly only detect sensitive data transmission, while cannot consider if the data transmission is user intended or not.

AppIntent [8] seems to be the first to systematically study a method to separate user-intended Android data transmission from unintended ones. For each data transmission, AppIntent can efficiently provide a sequence of GUI manipulations corresponding to the sequence of events that lead to the data transmission, thus helping an analyst to determine if the data transmission is user intended or not. Symbolic execution is an effective technique to extract feasible inputs that can trigger special behaviors of a program such as particular transmission of sensitive data.

Chen et al. proposed Pegasus [10], which focuses on detecting malicious app behaviors that are inconsistent with the GUI events. Pegasus detects malicious behaviors that can be characterized by the temporal order in which an application uses APIs and permissions. Nevertheless, privacy leakages cannot be modeled as app usage of permissions or APIs, thus many privacy leakages cannot be detected by such approach.

VetDroid [5] enhances Dynamic Taint Tracking by generating specifications for sensitive operations. However, the specification mainly focuses on the application logic but does not pay attention to the trigger condition of each operation.

FlowDroid [6] analyzes the apps bytecode and configuration files to find potential privacy leaks, either caused by carelessness or created with malicious intention. Opposed to earlier analyses, FlowDroid is the first static taint-analysis system that is fully context, flow, field and object-sensitive while precisely modeling the complete Android lifecycle, including the correct handling of callbacks and user defined UI widgets within the apps.

3. Design

An automated detecting sensitive data transmission for Android apps is an extreme complex mission. Instead, our system is designed to be an automated tool that finds the suspicious privacy leakage execution paths. From these execution paths, the path conditions including data inputs and events can be extracted. These critical information will facilitate the analyst understand how the malicious behavior is triggered and determine whether it is really a privacy leakage.

Our Goals. We have the following three goals:

- Produce the execution paths with good code coverage over most of suspicious malicious behaviors in the android apps. We need to thoroughly traverse diverse paths that may lead to a sensitive data leakage.
- Extract the path conditions that lead to sensitive data transmission. The path conditions are always composed of: a) Data inputs which contain data inputs from outside; b) Event inputs that trigger the sensitive data transmission.
- Provide an automated tool to represent what circumstance the sensitive data transmission happens. This helps the human analyst to judge whether the execution is a malicious behavior.

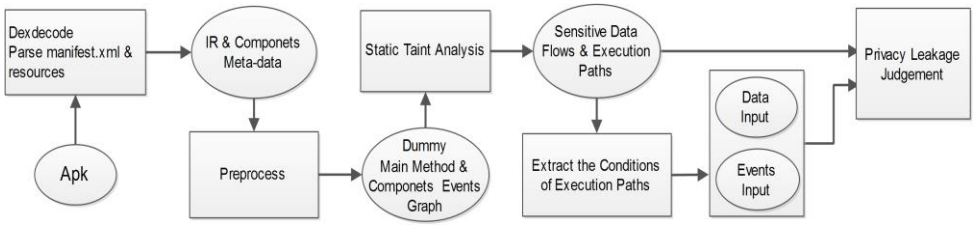


Figure 1. The Analysis Architecture

Overall Design. Figure 1 depicts the overall architecture of our scheme, which analyzes a target app in five steps:

- Apk decode. The first step is to convert an app's Dalvik bytecode to an intermediate representation (IR) amiable to static analysis. We do it with the Soot framework [11] and Dexpler [12]. An Android app does not have a main method, rather, components (activity, service, receiver, *etc.*,) are invoked through the various callback methods (including lifecycle methods). The definitions of components that be contained in manifest.xml and resources (layout of activities) are staring point of our analysis.
- Preprocess. Based on the meta-data of components and the IR, we as complete as possible to find all components and related callback functions. In addition, we also need to extract the event sequences among different components, which called component events graph.
- Static taint analysis. Static taint analysis labels (taints) data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and intercrosses messages. When tainted data leave the system (sinks), the subsystem records the sensitive data flow (the source, sink and

the path of propagation). We continue to extract the concrete execution paths which cover all connections of sources and sinks.

- Extract the conditions of execution path. Travel every executional path from the entry points to the sink point, an analyst can obtain the executional path conditions, where the data inputs (from outside) and the event inputs can be extracted.
- Privacy Leakage Judgment. Based on the global control graph and critical inputs, it can effectively visualize the inputs data and the UI events of the transmission so that the analyst can intuitively judge whether the transmission is privacy leakage or not.

4. Implementation

We implement the prototype based on the Java bytecode optimization framework Soot [11], which provides important prerequisites for a precise static dataflow analysis, in particular the intermediate representation "Jimple" and the accurate call-graph. A plugin of Soot called Dexpler [12] is used to convert Dalvik bytecode into Jimple code.

4.1. Pre-processing

1. Multiple entry points.

Android applications do not have a main method, it is composed of one or more components which work together to fulfill the functionality. We need to tell Soot what the entry points to the app are for a whole-program analysis. When constructing a call graph for dataflow analysis, we use the "custom entry points" feature of Soot and construct a custom dummy main method emulating the Android app's lifecycle as that of Arzt, *et al.*, work [6].

There are four different kinds of components for an android application can be defined: an activity is a single, focused thing that the user can do, services perform background tasks, content providers define a database-like storage, and broadcast receivers listen for global events. Most of components are registered via the AndroidManifest.xml file, however a malware may prefer to register and unregister a receiver at runtime. In our approach, we first collect components and callbacks that are defined in AndroidManifest.xml and layout XML files. We assume that all components inside an application can run in an arbitrary sequential order and put all collected callbacks into the dummy main method. According to this temporarily formed main method, we leverage soot to explore the IR code to look for dynamically registering of broadcast receivers. The found new components and callbacks function will be added into above dummy main method.

2. Asynchronously executing events sequence

However, a challenge for the application-wide dataflow analysis is that the sensitive dataflow maybe go through multiple asynchronous events belonging to different components. Our dummy main method constructed in above step simply assumes that all components can run in an arbitrary sequential order, this makes the subsequent dataflow analysis cost huge computational overheads because of searching a large number of invalid paths. Consider the case seen in Figure 2. It describes three asynchronous events sequence graph (for simplicity we omit essential events such as *OnStart(), OnRestart(), OnResume()*):

- The interaction between multiple Activities. *{MainActivity :: OnCreate()} → {(Button1) OnClickHandler()} → {Activity1 :: OnCreate()} → {(Button2) OnClick()};*
- A receiver catches an interest system event and starts a thread instance by calling *Thread.start(),* which invokes the *run()* method implementing *Runnable.run()*;
- The same receiver starts a service by calling *StartService(),* which triggers *Service :: OnStar()*.

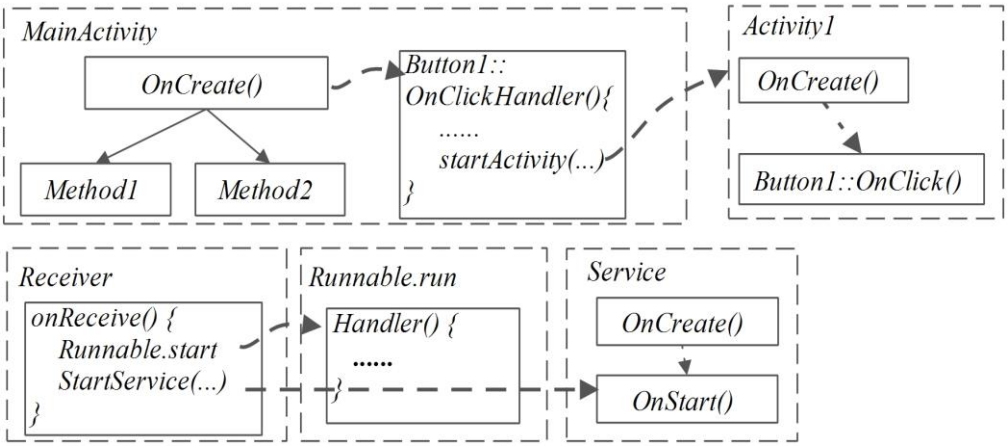


Figure 2. A simple Android Components and Events Sequence

A sensitive data propagation path can't cross an events sequence. Therefore, it's necessary to construct the sequence graphs of components' event to achieve higher efficiency and accuracy of static analysis. The process of construction sequence graph is presented in **Algorithm 1**.

Algorithm 1: Construct Call Graph of Components' Events

$C \leftarrow \{Components\ of\ the\ App\}$;
Function ConstructEventsSequence
    $ECG \leftarrow \varnothing$;
    for $c \in C$ **do**
        $S_c \leftarrow \varnothing$;
        if $IsActivity(c)$ then
            $S_c \leftarrow \{Callbacks\ defined\ in\ XML\ files\}$;
        end if
        $S_c \leftarrow S_c \cup \{Framework\ Callbacks\}$;
    end for
    $cg \leftarrow BuildCallGraph$;
    for $m \in \{All\ methods\ of\ the\ App\}$ **do**
        if $HasComponentInvorkApi(m)$ **then**
            $e_1 \leftarrow ParseTargetEvent(InvorkApiInfo)$;
            for $e_0 \in GetRootEvents(m)$ **do**
                $ECG \leftarrow ECG \cup BuildEventsSequence(e_0, e_1)$;
            end for
        end if
    end for
    return ECG;
end function

Output: The Call Graph of Components' Events

4.2. Static Taint Analysis

1. Sources and Sinks

A source is a system call which reads data from a shared resource (*e.g.,* the address book, the current location), the data from a source is often sensitive. A sink means a method which writes data to outside, the data which flows into a sink leaves the current environment [6]. Our data leakage flow analysis heavily depends on the given set of sources and sinks, we use SuSi [13], a fully automated machine-learning approach for identifying sources and sinks. After running SuSi on the current Android source code, the sources and sinks will be written into resulting text files. Categories include amongst Account, SMS, Contact, File, Location, *et al.*

2. Find the propagation paths from sources to sinks

After finding all sources and sinks in the IR code of an Android app, we start a backward taint analysis from every sink to certain the propagated path of the tainted values. Listing 1 shows a sample where the call sites of sources and sink are belong to a same entry point (a event). Line 14 is a sink call site, with the backward analysis the following variables and methods are sequentially marked as tainted: *msg*→*strFlg*→*info*→*getInfo()*→*imei*. We can find that *imei* is the output of the source *Tel.getDeviceId()*. Listing 2 shows a sample where the call sites of source and sink are belong to different entry points (even different components). Note that "android:onClick="sendMessage"" is defined in "layout/activity buttonTest.xml". According to the sink at line 12, the variable *ButtonTest.imei* is labeled as tainted. Although the source call can't be found in the control path of the *onClick()* event, we get the *ButtonTest :: onCreate()* as the previous event based on the sequence graphs build in **4.1**

Listing 1: Example1

```
1:  private String getInfo(){
2:        ......
3:             TelephonyManager Tel = (TelephonyManager)getSystemService(Context.
                                       TELEPHONY_SERVICE);
4:        String imei = Tel.getDeviceId();
5:        ......
6:        return imei;
7:  }
8:  private void sendMsg(){
9:         ......
10:       StringBuilder msg = new StringBuilder();
11:       OutputStream outStream = socket.getOutputStream();
12:       String info = getInfo();
13:       msg.append(strFlag).append(info);
14:       outStream.write(msg.toString());
15:        ......
16: }
```

Listing 2: Example2

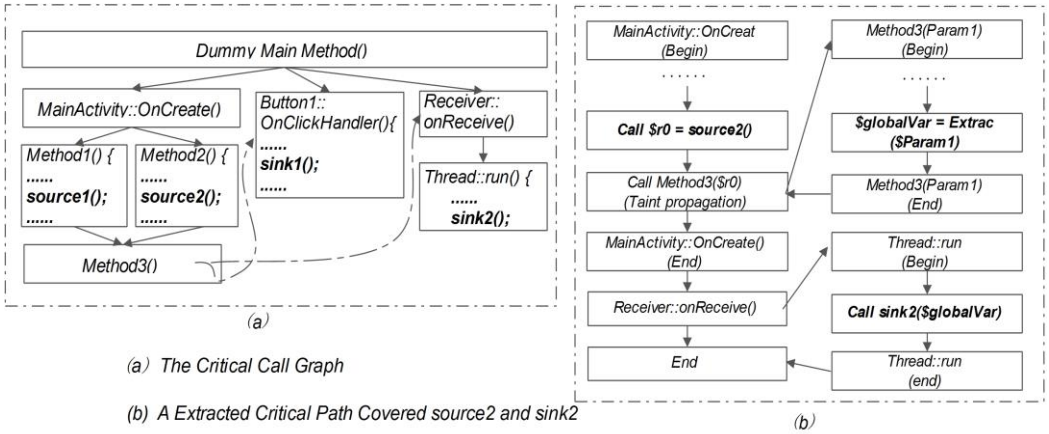
```
1:  public class ButtonTest extends Activity {
2:   private static String imei = null;
3:   protected void onCreate(Bundle savedInstanceState)
4:  {
5:        ......
6:        TelephonyManager tel = (TelephonyManager)
                          getSystemService(Context.TELEPHONY_SERVICE);
7:        imei = tel.getDeviceId();
8:  }
9:  public void sendMessage(View view){
```



```
10:        .....
11:        SmsManager sms = SmsManager.getDefault();
12:        sms.sendTextMessage("+49", null, imei, null, null);
13:  }
14:}
```

3. Construction of critical execution paths

After finding the connection between sources and sink, we also have got the most important sensitive data propagation path. The critical execution path can be extracted by backward traveling the call graph (produced by soot). The paths between entry points and the sources will be found. Figure 3 illustrates the detail of extracting critical path that is covered the source and sink.



(a) The Critical Call Graph

(b) A Extracted Critical Path Covered source2 and sink2

Figure 3. Critical Call Graph and Extracted Critical Path Connected the Source and Sink

4.3. Extract the Conditions of Execution Paths

One method of systematically generating input data (test cases) for programs is symbolic execution. However applying Symbolic Pathfinder (SPF, a symbolic execution tool) [14] to generate input data for Android apps is challenged by the fact that Android apps run on Dalvik Virtual Machine (DVM) instead of JVM. In addition, Android apps are event driven and susceptible to path-divergence due to their reliance on an application development framework. In order to do so, we first need to generate the Java byte-code of the app that can be executed on JVM. It can be well done by soot. In addition, a typical app is composed of multiple components (i.e., Activities and Services) that communicate using Androids Intent messages. These components' events can be extracted as a set of disconnected sub-call graphs that collectively represent the apps logic. To extract the conditions of execution paths, we use the critical call graph constructed in section 4.2 as a guideline to run symbolic execution as follows.

- Select a critical execution path (CEP) from the critical call graph that contains a single source-sink connection;
- Run symbolic execution with the guideline of CEP, output the input data (test cases) and the UI events input.

Using Figure 3 as an example, guided by the critical call graph, the symbolic execution explores a much smaller space to test the path from source2 to sink2. By using the choco data constraint solver [15], we generate corresponding data inputs according to the data constraints calculated in the symbolic execution. The events inputs also can printed during the symbolic execution.

4.4. Privacy Leakage Judgment

A better indicator of a true privacy leakage should be whether the transmission is user intended or not [8]. With the data inputs, events inputs and the layout design (layout.xml), we almost can answer that what triggered the occurrence of suspicious behavior and whether the user is aware of what is happening.

5. Effectiveness

We extensively experimented with our system for sensitive data leakage analyses using a set of apps: 100 popular apps from Google Play. Our experiments are conducted on a machine with 2×2.26 GHz Quad-Core Xeon and 128GB of RAM. We excluded the Official Android libraries and other popular third-party libraries since they are huge in size and cause a long time execution. Figure 4 presents the time taken by our system to construct whole analysis for the 100 apps. Our system successfully parse and execution for 87 apps. Executions with the result of the experiment, we notice that the top free apps still have user unintended leakages. The leakage of device information, phone number and location are found in some apps as shown in Table 1, the number represents the found cases' count.

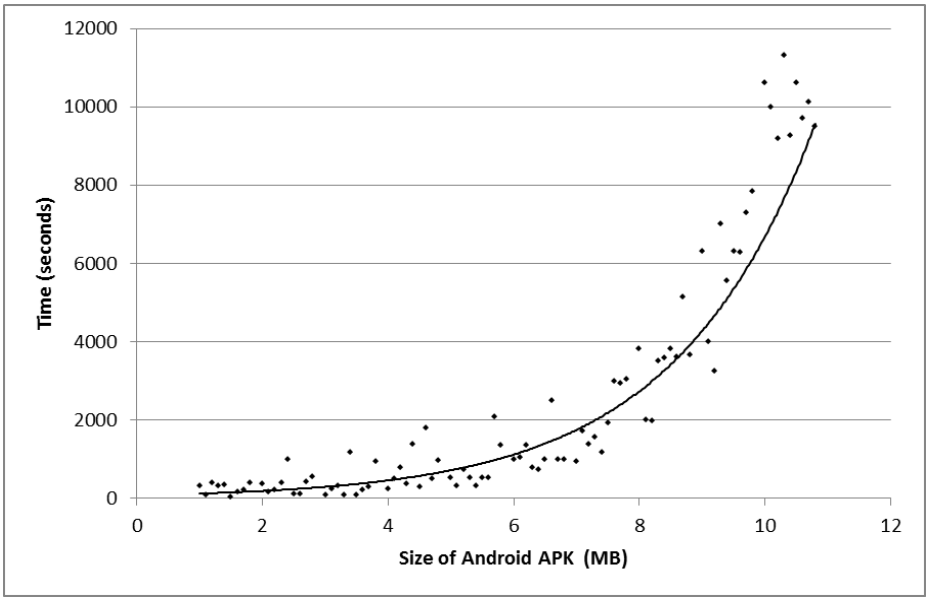


Figure 4. Time of Sensitive Data Leakage Analysis

Table 1. Results of Sensitive Data Leakage Analysis

	Underground (*maybe user unintended*)	With Active Events (*user intended*)
Device ID	67	11
Phone Number	18	8
SMS	1	6
Contacts	2	3
Location	19	2

6. Case Study

Here we use an android malware sample iBanking [16] to illusion how our system detects the privacy leakage. First, our system construct the whole components' relative graph by parsing the AndroidManifest.xml, layout.xml and the jimple code of these

components as presented in Algorithm 1. Figure 6 show the main components and their relationship.

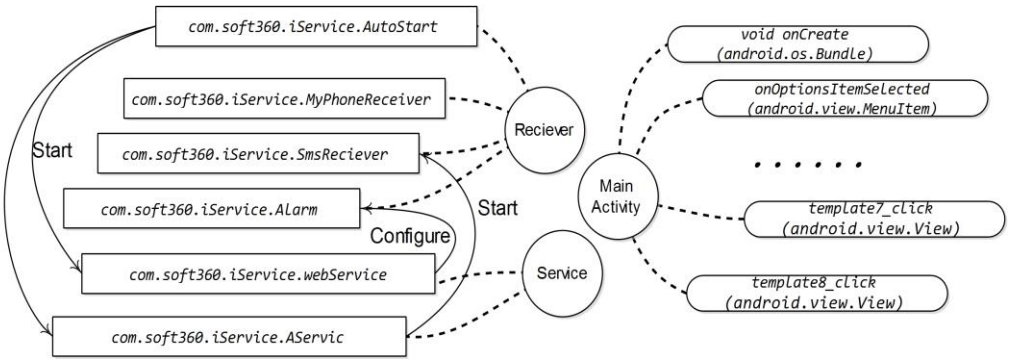


Figure 6. The Critical Components in iBanking

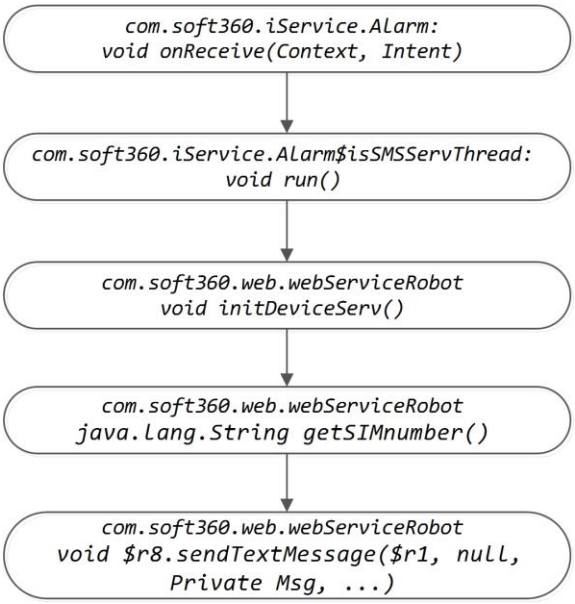


Figure 7. A Privacy Leakage Path

Base on the callback analysis of component, we can easy construct the dummy main function which is set as enter point in the soot analysis framework. In the next analysis, our system scans whole jimple code of the apk and finds 91 sources and 125 sinks. Then a backward taint analysis is started from every sink point, eventually 27 suspected privacy leak paths are found. After the symbolic execution, we extract the path conditions and the related events of all suspicious paths, and it can be judged that these paths almost are the malicious behavior execution paths. Here we choose a path, described in Figure 7, for explanation. The path condition is "none", the related events is the receiver "com.soft360.iService.Alarm: void on Receive (Context, Intent)". Then we can tell that the path is a high risk suspected path since it is executed in interval auto automatically, the "automatically" means that the users may not be aware of this.

7. Conclusion

In this paper, we propose a novel approach that finds privacy leakage of Android app. We implement a prototype system based on the soot framework. We evaluate a malware sample as cast study and experiments show that our system is capable of detecting the

privacy leakage paths with relatively high rate. However, our system require a high overhead of computation and time while exploring the taint value propagating paths. This should be improved in future.

References

[1] A. P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, "Android permissions demystified", in: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, ACM, **(2011)**, pp. 627–638, New York, NY, USA.
[2] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution", in: Security and Privacy (SP), IEEE Symposium on, **(2012)**, pp. 95–109.
[3] I. Burguera, U. Zurutuza and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android", in: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '11, ACM, **(2011)**, pp. 15–26, New York, NY, USA.
[4] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu and E. Kirda, "A quantitative study of accuracy in system call-based malware detection", in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA, ACM, **(2012)**, pp. 122–132, New York, NY, USA.
[5] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis", in: Proceedings of the ACM SIGSAC conference on Computer & communications security, ACM, **(2013)**, pp. 611–622.
[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau and P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, SIGPLAN Not., vol. 49, no. 6, **(2014)**, pp. 259–269.
[7] 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: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, USENIX Association, Berkeley, **(2010)**, pp. 1–6, CA, USA.
[8] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection", in: Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, CCS '13, ACM, **(2013)**, pp. 1043–1054, New York, NY, USA.
[9] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis", in: Proceedings of the 2012 Third World Congress on Software Engineering, WCSE '12, IEEE Computer Society, **(2012)**, pp. 101–104, Washington, DC, USA.
[10] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard and D. X. Song, Contextual policy enforcement in android applications with permission event graphs, in: NDSS, **(2013)**.
[11] P. Lam, E. Bodden, O. Lhoták and L. Hendren, "The soot framework for java program analysis: a retrospective", in: Cetus Users and Compiler Infastructure Workshop (CETUS), **(2011)**.
[12] A. Bartel, J. Klein, Y. Le Traon and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot", in: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP '12, ACM, **(2012)**, pp. 27–38, New York, NY, USA.
[13] S. R. Steven Arzt and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks", Ec spride technical report tud-cs-2013-0114, **(2013)**, URL http://www.bodden.de/pubs/TUD-CS-2013-0114.pdf.
[14] "Symbolic pathfinder", URL http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.
[15] "Choco data constraint solver", URL http://www.emn.fr/z-info/choco-solver.
[16] "Android.spy.49 (ibanking)", URL http://www.kernelmode.info/forum/viewtopic.php? f=16&t=3166.

Authors

Songyang Wu, He is an associate professor at The Third Research Institute of Ministry of Public Security, China. He received his Ph.D. Degree in computer Science from TongJi University, China in 2011. His current research interests are in information security, cloud computing and digital forensics.

Yong Zhang, He received his Ph.D. Degree in computer Science from East China Normal University, China in 2013. He is a research member in The Third Research Institute of Ministry of Public Security, Shanghai, China. His research interests include information security, digital forensics and cryptography.

Xiong Xiong, He received his Master's Degree in computer Science from North China Institute of computing technology, China in 2011. He is a research member in The Third Research Institute of Ministry of Public Security, Shanghai, China. His research interests include information security and digital forensics.

