Android Application Visual Safety Analysis Based On Component Relations

Hao Chen[1], Li Pan[*2]

[1,2]National Engineering Laboratory for Information Content Analysis Technology,
Department of Electronic Engineering, Shanghai Jiao Tong University, China
[1]chandmn@sjtu.edu.cn; [*2]panli@sjtu.edu.cn

***Abstract*-This study attempts to reduce the hidden danger of Android application installation. Regular methods for detecting malicious software need a great amount of sample data to implement the feature extraction and behavior matching, which makes the detection difficult and complex. We propose an automatic modeling method based on the analysis of the source code to describe the behavior of the application with its components. An attack graph was then drawn to visualize the application framework and elements related safety, with which the users could have a deeper acknowledgment about the hidden danger instead of the fuzzy recognition of the traditional show of permission applied before the installation of the application. The automatic modeling method allows the user to protect their private data with little difficulty, and less complexity than traditional software analysis methods.**

Keywords- Android Safety; Component; Attack Graph; Static Analysis; Visualization

I. INTRODUCTION

With the rapid development of mobile technology, mobile terminals have become increasingly powerful and will gradually replace personal computers as the most popular processing platforms. As the most popular mobile operating system, Android has been applied to various important areas beyond smartphones, including education, medicine, and automation [1]. Android allows users to install a variety of applications to enhance the system, with which users can complete more information interations. Therefore, Android system is often used to store traditional SMS (Short Message Service), digital address books, and other important and sensitive information. The Android operating system is more vulnerable to malware compared with other systems, such as iOS and Blackberry OS since it is open-source software and has a programmable framework feature.

Researchers mainly study privilege analysis and taint analysis for privacy protection of Android devices. Michael Grace et al. divided the privilege attacks of Android applications into two categories, explicit attacks and implicit attacks [2]. While the explicit attack is when a low-privilege application directly imitates the actions of high-privilege applications through interfaces of high-privilege applications. The implicit attack is when the malicious application can get the user identification shared by multiple applications, to obtain the same privilege as the other normal applications [3]. Geneiatakis D et al. combined both runtime information and static analysis to profile mobile applications and identify if they are over-privileged or follow the least privilege principle [4]. They proposed a framework that could automatically assure whether an application follows the least privilege principle, and identifies over-privileges with a certain confidence level. V Moonsamy et al. proposed a novel pattern mining algorithm to identify a set of contrast permission patterns that aim to detect the difference between clean and malicious applications [5]. J Choi et al. studied the relationship between the main characteristics of each category of permissions request and privileges on information disclosure and categorized the target applications [6]. Then they compared the permissions and privileges to detect any present malware. Also, the sensitive interfaces and components of applications, and data flow analysis have been studied [7-9].

For the private data leak, the basic idea is marking the target data as tainted, and monitoring its dissemination. W Enck W et al. proposed a taint analysis tool named TaintDroid [10]. It uses variable-level tracking within the VM (Virtual Machine) interpreter and stores the taint markings as a single taint tag. When applications execute native methods, they could then acquire the data flow by monitoring taint tags. G Bal et al. summarized these two pitfalls in the design principle that actual and potential information flows should be made transparent to the user, and proposed a system named Styx to monitor the sensitive information flows to assess long-term privacy impacts [11]. Considering the difference of applications' lifecycle between Android and computer operating system, C Fritz et al. proposed a static taint analysis tool based on the lifecycle model [12].

Here, we establish a model describing the application in components. The model analyzes software behaviors about privilege and data privacy as mentioned above, and illustrates to users what the application does with a graph showing the relations among the components. In this manner, misjudgment caused by purely static analysis could be avoided, and users could judge if the application's behavior consists with its application scenario.

II. SYSTEM DESIGN

We aim to visualize software behavior in the form of components and assist users in estimating the security of their applications. There are several studies on Android permission modeling [13-14]. Previously models were built based on application components and permissions to enhance Android's permission system to support rich policies. Here, however we

DOI: 10.5963/IJCSAI0501004

simplify our model and only target the security properties related to privacy leakage and latent attacks between applications, so as to process code packages quickly and generate a visual overview to describe an application before users install it. We do not analyze the Android permission system, but describe the application in a straightforward way with fewer variables. As shown in Fig. 1, we define the componentization model based on related literature research analysis. Then, we describe the application in components regarding the source code according to the model defined, and mark the vulnerable components and sensitive components. Following the analysis of the relations among components, we make an attack graph containing all components, relations and security factors, to indicate the application visually.

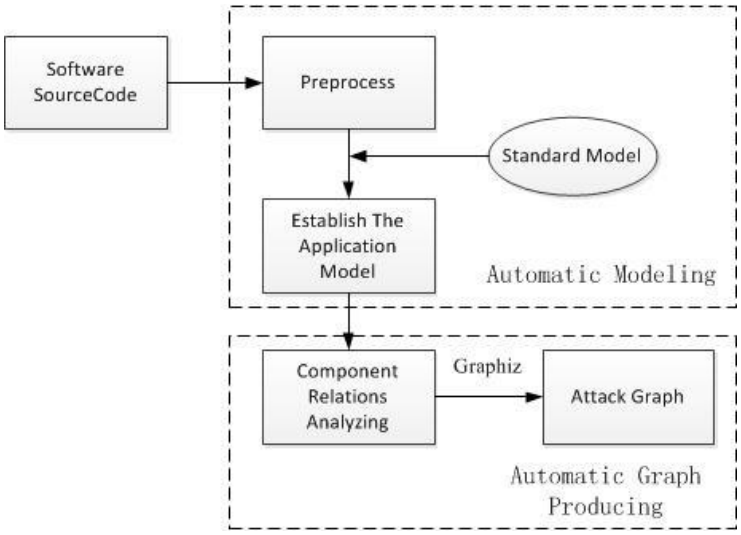

Fig. 1 The visual analyzing flows based on source code

We define the Android applications as composed of the component C.

$$C = \{name, action, per, C_{from}, C_{to}, type, data\} \quad (1)$$

where $name$ is the class name of the component, $action$ describes the action of the component, and it includes system action and user-defined action; per indicates the permission of the component. Only if a component holds the specific permission, can it access the relative protected API. C_{from}, in the form of $\{C_1, C_2, \bullet\bullet\bullet, C_n\}$, contains all components that call the present component. Likewise, C_{to} contains all components the present component calls. The $type$ term indicates the kind of the component, one of *Activity*, *Service*, *BroadcastReceiver*, *ContentProvider*. The $data$ term is the set of data the component accesses.

III. AUTOMATIC MODELING

A. Type Description and Name Description

We obtain the class declaration through the regular matching search and get the superclass of the component. For example, the following code shows that the types of these two components are $C\{type\}$ = *Service*.

```
Public class MyService extends Service {
…
}
```

The description of the component name is based on the analysis of text with respect to basic component types. Taking the component mentioned above for example, we name the components as $C\{name\}$ = *MyService*, by matching the class name between the keywords *extends* and *class*.

B. Description of Component Action

Component actions are divided into system-defined actions and user-defined action [15]. The former one is mainly related to the basic behavior of the system, which is also classified into *Activity* action and *Broadcast* action. The main method of automatic analysis on system standard action is as follows. First, we store system action into the matched file

DOI: 10.5963/IJCSAI0501004

SystemActionRepo , and then we match the source code in components with related keywords in *SystemAction* , finally reconstruct the matched results above to obtain the system action description set $C\{actionS\}$.

TABLE 1 THE DETECTION ALGORITHM OF THE USER-DEFINED ACTION

Input: *java* file, *AndroidManifest.xml* file

Output: the user-defined actions set $C\{actionS\}$

1: $\phi(\eta_1, t_1, n_1) \leftarrow$ action receiver in the *XML* file

2: $\psi(\eta_2, t_2, n_2) \leftarrow$ action sender in *java* file

3: $\phi, \psi \rightarrow \theta(\eta, C_1, C_2)$

4: $\theta(\eta, C_1, C_2) \rightarrow C_1\{actionS\}, C_2\{actionS\}$

As shown in Table 1, through text analysis of the *XML* file, we obtain the keyword η , type t_1 , and name n_1 of the action acceptor component. Then we define the set of action acceptor components as $\phi(\eta_1, t_1, n_1)$. Through text analysis of *java* file and regarding the common form of the action implement code, we obtain the set of action sender components as $\psi(\eta_2, t_2, n_2)$. After matching the corresponding relations between the two sets, we get the action description of sender component $C_1\{actionS\}$ and the acceptor component $C_2\{actionS\}$.

In most cases, the component action is of little use to help users acknowledge the application except in analyzing the component relations. And in the attack graph (&3), the component action-related private data leakage really needs to be shown. In general, the malicious software sends users' private data out by three primary routes, text messages, phone calls and HTTP networks. It is needed to find out the components which have an approach to sending the private data, so we revise the format of $C\{action\}$ to $C\{action(message,call,web)\}$. The message outlet component is taken as an example to show how to detect whether the component has a data outlet.

C. Description of Component Permission

The permissions in Android applications are mainly divided into application-applied permission and component-access permission [16]. The former is applied in the user-permission attribute of AndroidManifest.xml file. The latter, whose permissions are constrained by the type of the component, is also declared in the correspondent part in the AndroidManifest.xml file. Only the application that has the corresponding declared permission can access the component. And the component can not be accessed as a non-private component by any other application unless the "android: exported" is "true" in AndroidManifest.xml file.

Based on the format of the permission declaration in the XML file, we analyzed the keywords in different components (*e.g.,*<*keyword,keyword/*>) to judge if the component is private and quantify its *ProtectionLevel* as an integer $C\{per\}$ so as to evaluate the possibility of its being maliciously used.

TABLE 2 THE QUANTITATIVE VALUE OF THE SECURITY LEVEL OF THE COMPONENT

exported	**ProtectionLevel**	$C\{per\}$
exported="true"	NULL	0
exported="true"	normal	1
exported="true"	dangerous	2
exported="true"	signature	3
exported="false"		4

As shown in Table 2, higher $C\{per\}$ indicates less security risk and smaller chance that this component is utilized by other malicious applications; in reverse, lower $C\{per\}$ suggests that the component is more likely to be used through cross-application attacks, thus the the software security risk is higher.

D. Description Of Caller and Callee Components

As described in the user-defined actions, *Intent* is the carrier of the component communication and it's also the medium between the caller and callee components [17]. We define *ComponentRepo* as the separated component set we obtain. First, by *Intent* analysis to determine whether the component has any calling behavior, we check if $C_{from} - C_{to}$ is NULL, or not. In reverse, based on the calling method of different components we can acquire the component name of the called components, and then implement $C_{from} - C_{to}$ matching upon all components to gain the caller and callee relationships.

DOI: 10.5963/IJCSAI0501004

E. Description of Component Operation Data

According to an important level of data, we listed the private data and illustrated the commonly used interface of these private data. These private data include contacts, call history, SMS, location information, the user schedule, camera data, microphone data, the current screen contents, SIM card information, browser bookmarks information, browser history.

In summary, eleven kinds of private data in this study can be accessed through *ContentProvider* and *URI*, and the others can also be obtained via specific approaches. We conduct the matching analysis of the components in our model and obtain the data $C\{data\}$. Then we judge whether the data obtained is private data compared with the private data list and describe it as $data(BOOL)$, which indicates if the data belongs to the private part.

IV. AUTOMATIC ATTACK GRAPH

Attack graph analysis is a traditional security analysis method, and it is mainly used in internet security and system security scanning. It detects the vulnerability of the system or network from the attackers' perspectives, and lists all possible attacking ways in terms of core files or key nodes to help IT engineers take defensive measures.

In the work presented here, we set the target as the core private data in Android operating system and the nodes as the components in the graph. By matching the component relations and analyzing the component attributes, attack graph of the application was drawn and shown to the users to give them a deeper knowledge of the application's security.

Attack Path Constraints and Implementation Guidelines are as follows:

$$P_{ki} = C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow \cdots C_n \rightarrow D_k \tag{2}$$

where P_{ki} is an attack path ends at the private data D_k, where 0< k <12, i >0.

Component - Component attack path

For any component C_i, C_j, if $C_j \in C_i\{C_{to}\}$ and $C_i \in C_j\{C_{from}\}$, the path $C_i \rightarrow C_j$ is established.

Component - Data attack path

For any component C_i and any data block D_k, if $D_k \in C_i\{data\}$, the path $C_i \rightarrow D_k$ is established.

To draw the attack graph, first we find out the components whose C_{from} is NULL from the whole components we obtain from &2. The components are just the starting points like StartNodes@{C} shown in the Fig. 2. Then we find out the components whose $C\{data\}$ is not NULL, setting them as EndNodes@{C} referring to the related data block. By matching the other components data blocks, we describe all the paths among the nodes with code in Graphviz format and draw up with vulnerable components marked.

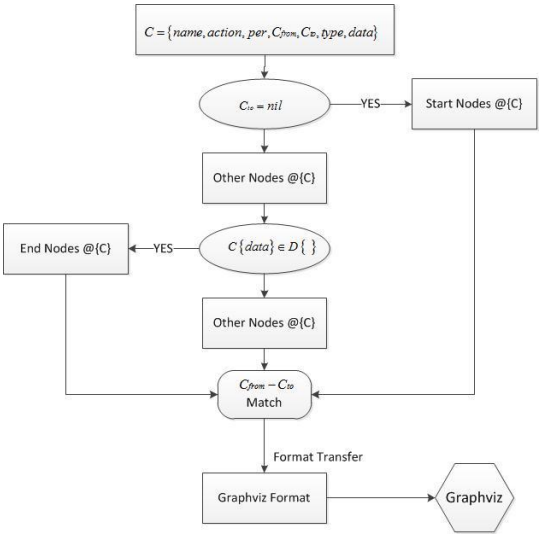

Fig. 2 The flow of generating the attack graph based on nodes analysis

DOI: 10.5963/IJCSAI0501004

V. EXPERIMENTS

An open-source application named Express Track was selected in this study. It was published in the Android platform and used to query the express information. Its source code could also be obtained on the open-source site.

In this section, we generated an attack graph in terms of the software package to describe the application framework with component relations and the security status with private data leakage analysis and inter-application attack analysis. With the graph, the users could know if the application is just a tool to query the express information and if it has risks of being attack by other applications or privacy disclosure.

A. Automatic Modeling

To generate the attack graph, the model needed to be established on the software by the automatic modeling program. In terms of the form of the application component we defined, the program extracted all components from the software package.

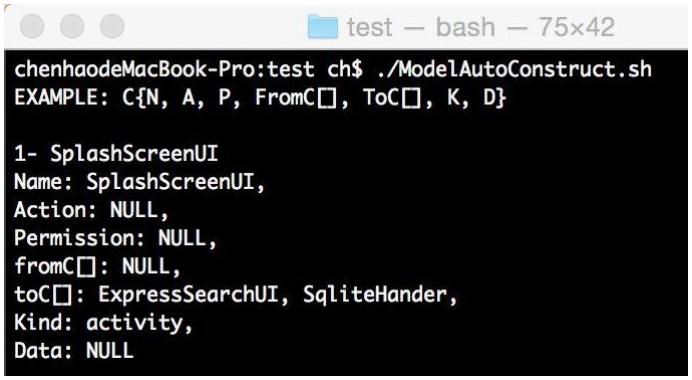

Fig. 3 The result of automatic modeling

As shown in Fig. 3, one component was extracted from the software package. The property *Action* indicated the component had no data exit such as the message, call or the web. *Permission* suggested the property "android: exported" is "false", so the property *Per* which is quantified from 0 to 4 to describe the risk of being attacked by another application is 4. The component had no caller components and two called components. It acted as an *Activity* component and did not touch any data. The output was the set of all the separated components.

TABLE 3 THE DESCRIPTION OF THE COMPONENTS IN THE APPLICATION

	Name	Action	Per	C_{from}	C_{to}	Type	Data
C_1	SplashScreenUI	NULL	4	NULL	C_2, C_7	activity	NULL
C_2	ExpressSearchUI	NULL	4	C_1	C_6, C_3	activity	NULL
C_3	SearchResultUI	NULL	4	C_2	C_7	activity	NULL
C_4	HistoricalUI	NULL	4	C_6	C_7	activity	NULL
C_5	BaseControl	NULL	4	C_6	NULL	other	NULL
C_6	MenuControl	NULL	4	C_2	$C_4, C_8, C_9, C_5,$	other	NULL
C_7	SqliteHander	NULL	4	NULL	NULL	other	data (NO)
C_8	CheckUpdateService	NULL	4	C_6	C_{10}	other	NULL
C_9	ExpressInfoService	NULL	4	C_6	C_{10}	other	NULL
C_{10}	SearchTools	A(web)	4	C_8, C_9	NULL	other	NULL

In Table 3, $name$ shows the component name. The component action is revised into the format $C\{action(message,call,web)\}$ and NULL means the component has no approach to sending data out. The permission value per is quantified to show the vulnerability of the component. C_{from} and C_{to} contain the caller components and called components. Type indicates the component type. Data demonstrates the data block this component can access. NULL means no data accessible and $data(BOOL)$ indicates there is data accessible, where the BOOL value shows whether the data is core privacy.

B. Automatic Attack Graph Drawing

After matching all the components displayed above, we draw the link between components according to the attack path principle define in section 4. Every oval is an application component, among which the purple one is data block and the green is the data outlet component. In this experiment, the data block contains no core private data and the outlet of data is the web only.

DOI: 10.5963/IJCSAI0501004

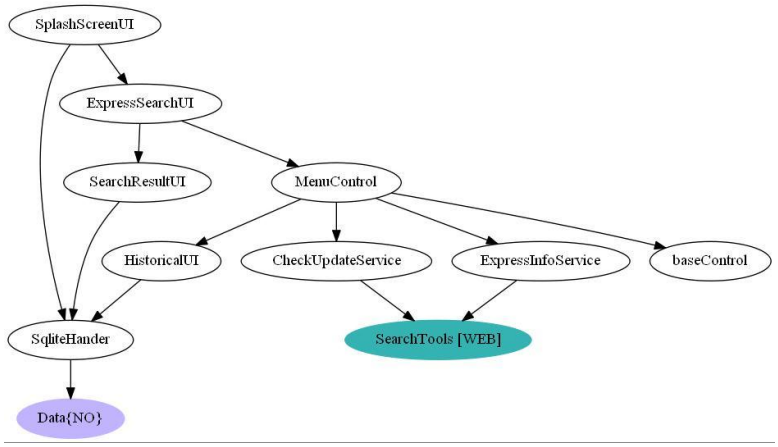

Fig. 4 The attack graph of the application

The attack graph shows as Fig. 4. The database part shows that the software does not involve the core privacy data, which accords with the function of the application. There is no need to acquire additional data if it is not malware. The component named *SearchTools* includes web outlet. Since the application is used to query express information online, the ability accessing the internet is acceptable. With the graph, the user has a visual assessment of the behavior of the application and make a wisdom decision whether to install it.

VI. CONCLUSION

In this paper, we proposed a model to describe the Android application in components at first. Different from other research on Android modeling which aims to analyze the system security mechanism, we commit ourselves to describing the application in a straightforward way with fewer variables. We focused on the core private data leakage and the latent attack by other applications. A program was implemented which could automatically generate an attack graph describing the component relations and the key components clearly. Users could conduct informed installation of the application with the knowledge of application security in the graph instead of knowing little security information as before.

ACKNOWLEDGMENT

This work is sponsored by The National Key Technology R&D Program (2014BAG01B02).

REFERENCES

[1] Deokar P T and Nagmode M S, "Cloud Server Based Home Automation System Using Android Phone," *International Journal of innovative Research in science & Engineering*, 8 pages.

[2] Grace M C, Zhou Y, Wang Z, et al., "Systematic Detection of Capability Leaks in Stock Android Smartphones," *NDSS*, 15 pages, 2012.

[3] Bugiel S, Davi L, Dmitrienko A, et al., "Towards Taming Privilege-Escalation Attacks on Android," *NDSS*, 18 pages, 2012.

[4] Geneiatakis D, Fovino I N, Kounelis I, et al., "A Permission verification approach for android mobile applications," *Computers & Security*, vol. 49, pp. 192-205, March 2015.

[5] Moonsamy V, Rong J, and Liu S, "Mining permission patterns for contrasting clean and malicious android applications," *Future Generation Computer Systems*, vol. 36, pp. 122-132, July 2014.

[6] Choi J, Sung W, Choi C, et al., "Personal information leakage detection method using the inference-based access control model on the Android platform," *Pervasive and Mobile Computing*, vol. 24, pp. 138-149, December 2015.

[7] Zhongyang Y, Xin Z, Mao B, et al., "DroidAlarm: an all-sided static analysis tool for Android privilege-escalation malware," *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM*, pp. 353-358, May 2013.

[8] Chin E, Felt A P, Greenwood K, et al., "Analyzing inter-application communication in Android," *Proceedings of the 9th international conference on Mobile systems, applications, and services. ACM*, pp. 239-252, June 2011.

[9] Chan P P F, Hui L C K, and Yiu S M, "Droidchecker: analyzing android applications for capability leak," *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. ACM*, pp. 125-136, April 2012.

[10] Enck W, Gilbert P, Chun B G, et al., "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Communications of the ACM*, vol. 57(3), pp. 99-106, June 2014.

[11] Bal G, Rannenberg K, and Hong J I, "Styx: Privacy risk communication for the Android smartphone platform based on apps' data-access behavior patterns," *Computers & Security*, vol. 53, pp. 182-202, September 2015.

[12] Fritz C, Arzt S, Rasthofer S, et al., "Highly precise taint analysis for Android applications," EC SPRIDE, TU Darmstadt, Tech. Rep, 14 pages, May 2013.

[13] Shin W, Kiyomoto S, Fukushima K, et al., "A formal model to analyze the permission authorization and enforcement in the android

DOI: 10.5963/IJCSAI0501004

framework," *Social Computing (SocialCom), 2010 IEEE Second International Conference on. IEEE*, pp. 944-951, August 2010.

[14] Fragkaki E, Bauer L, Jia L, et al., "Modeling and enhancing Android's permission system," Computer Security–ESORICS 2012. Springer Berlin Heidelberg, vol. 7459, pp. 1-18, 2012.

[15] http://developer.android.com/guide/topics/manifest/action-element.html.

[16] http://developer.android.com/guide/topics/manifest/permission-element.html.

[17] http://developer.android.com/guide/components/intents-filters.html.

DOI: 10.5963/IJCSAI0501004