Journal of Korea Multimedia Society Vol. 20, No. 5, May 2017(pp. 827-834)
https://doi.org/10.9717/kmms.2017.20.5.827

Crowdsourced Risk Minimization for Inter-Application Access in Android

Youn Kyu Lee†, Tai Suk Kim††

ABSTRACT

Android's inter-application access enriches its application ecosystem. However, it exposes security vulnerabilities where end-user data can be exploited by attackers. While existing techniques have focused on minimizing the risks of inter-application access, they either suffer from inaccurate risk detection or are primarily available to expert users. This paper introduces a novel technique that automatically analyzes potential risks between a set of applications, aids end-users to effectively assess the identified risks by crowdsourcing assessments, and generates an access control policy which prevents unsafe inter-application access at runtime. Our evaluation demonstrated that our technique identifies potential risks between real-world applications with perfect accuracy, supports a scalable analysis on a large number of applications, and successfully aids end-users' risk assessments.

Key words: Android, Inter-app Security, Risk Analysis, Crowdsourcing

1. INTRODUCTION

End-users store their private information in their mobile devices. Although a large body of research has focused on implementing various security measures to protect private information in mobile devices, security risks still exist in Android's inter-application (inter-app) access [1], in which components belonging to different apps communicate via a particular type of message called *intent*. Specifically, Android's inter-app access is exposed to inter-app attacks [2] that exploit intents to eavesdrop private information or manipulate critical functionalities of victim apps.

Detecting a potential risk of inter-app attacks is challenging because malicious apps disguise their behavior as benign or stealthy. Furthermore, especially for non-expert users, it is more challenging to correctly assess the riskiness of detected vulnerabilities. Despite recent efforts, the existing techniques for preventing inter-app attacks suffer from inaccuracy in potential risks detection [3, 4] or primarily target end-users who have expertise in Android [4-8]. While crowdsourcing technique has been proposed to aid non-expert end-users in security configuration of mobile devices, it neither targets inter-app attacks, nor it detects potential risks between apps [9].

In this paper, we present a novel technique that minimizes the risks of inter-app attacks between target apps. It automatically identifies potential risks between a given set of apps, aids end-users to assess the identified risks by crowdsourcing their assessments, and generates an access control policy which prevents unsafe accesses at runtime. Our technique is distinguished from prior works because (1) it detects potential risks of inter-app attack more accurately than existing techniques,

※ Corresponding Author : Tai Suk Kim, Address: Dept. of Computer Software Engineering, Dong-eui University 176 Eomgwangno Busan_jin_gu, Busan 614-714, Korea, TEL : +82-51-890-1707, FAX : +82-51-890-1724, E-mail : tskim@deu.ac.kr
Receipt date : Feb. 2, 2017, Revision date : Apr. 2, 2017
Approval date : Apr. 6, 2017
† Computer Science Department, University of Southern California USA
(E-mail : younkyul@usc.edu)
†† Dept. of Computer software Engineering Dongeui university

(2) it supports a scalable analysis of a number of apps, (3) it enables non-expert users to effectively assess potential risks by introducing crowdsourcing approach, and (4) it integrates static detection with runtime access control of inter-app access.

This paper makes the following contributions: (1) we proposed a novel technique that enables end-users to protect their devices from inter-app attacks; (2) we developed a prototype tool that implements the proposed technique; (3) we provided the results of evaluations that involve real-world Android apps, comparable techniques, and end-users.

Section 2 illustrates inter-app attacks that motivate our research. Section 3 details our approach and Section 4 presents the three evaluations of our technique. A discussion of related work is provided in Section 5, and our conclusions are presented in Section 6.

2. MOTIVATING EXAMPLES

Fig. 1 presents an example of inter-app attack, *intent spoofing*, where a malicious component (=*Malicious1*) exploits a critical function (i.e., *sendTextMessage()*) of victim component (=*Victim1*)

```
// Victim Component
1 public class Victim1 extends Activity {
2  public void onStart() {
3    Intent i = getIntent();
4    String smsNum = i.getStringExtra("Num");
5    String smsText = i.getStringExtra("Text");
6    SmsManager Mngr = SmsManager.getDefault();
7    Mngr.sendTextMessage(Num, null, Text, null, null);}}
// Malicious Component
1 public class Malicious1 extends Activity {
2  public void onCreate(Bundle savedInstanceState ){
3    Intent i = new Intent();
4    i.setComponent(new
ComponentName("VicApp1","VicApp1.Victim1"));
5    i.putExtra("Num", "000-000-0000");
6    i.putExtra("Text", "http://unsafe/spoofing");
7    startActivity(i); }}
```

Fig. 1. An Example of Intent Spoofing.

by sending an intent. While those components belong to different apps, in Android, a component can initiate another component by sending an [1]explicit intent. As depicted in Fig. 1, if *Malicious1* sends an explicit intent targeting *Victim1* and containing spoofed information (i.e., unsafe number and text), *sendTextMessage()* in *Victim1* will be subsequently triggered and send a text message with the spoofed information bundled in the incoming intent. In this case, a potential risk of inter-app attack exists between *Malicious1* and *Victim1.*

Reversely, in the case of *intent hijacking*, if a victim component was designed to send or broadcast an implicit intent that contains private information without any protection (e.g., permission [7]), a malicious component can eavesdrop the intent's bundled information by declaring attributes to receive the intent.

3. RISK MINIMIZATION OF INTER-APP ACCESS

As depicted in Fig. 2, our approach is divided into five distinct phases. (1) *Construct component models*: A set of apps is analyzed and each app's

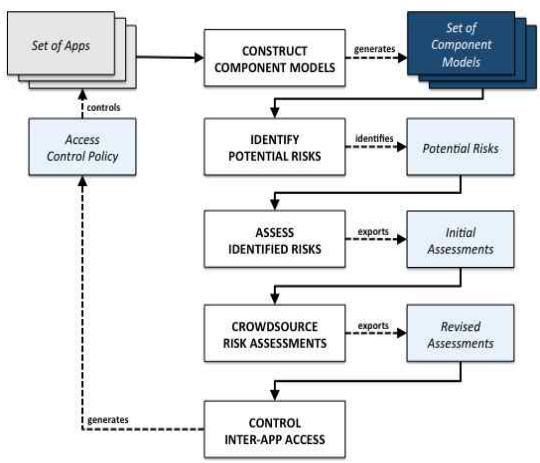


Fig. 2. An Overview of Our Approach.

1) An explicit intent specifies target component, while an implicit intent does not, but instead declares attributes, such as action, category, and data type [12].

information is transformed into component model; (2) *Identify potential risks*: Potential risks between the apps are detected from the component models; (3) *Assess identified risks*: An end-user assesses the safety level of each identified potential risk. (4) *Crowdsource risk assessments*: Each end-user's assessment is crowdsourced and filtered based on the similarity of safety levels. (5) *Control inter-app access*: An access control policy is generated inorder to prevent unsafe runtime accesses. In the remainder of this section, we will detail our approach.

3.1 Construct Component Models

Identifying potential risks between apps requires intra-/inter-component flow analysis. However, for a large number of apps, traversing control-flow graphs through every component is not scalable [5, 6]. To address this, we employ a component model which summarizes the information of each component. Our model enables a scalable risk analysis by providing a macro perspective of a set of apps.

Constructing component models is processed as follows. For a given set of apps, by parsing each app's manifest file and statically analyzing its bytecode, five different types of information are extracted: *app name*, *component names, intents, intent filter, and permission (intent* and *intent filter* contain the information of attributes, i.e., target component, action, categories, and data; *permission* includes the name of permission an app holds or requires [7]). Meanwhile, each component's safety is examined by checking whether a component contains a data-/control-flow between an *Intent method* and a *critical method*. The *critical method* refers to an Android system method which can handle an end-user's sensitive information or trigger private operation such as *getDeviceId()* and *sendTextMessage()* [8]. The *Intent method* refers to an Android system method for sending or receiving an intent (e.g., *startActivity()* and *onCreate()*) [7]. If a flow directs through an *Intent method* to a critical method, the risk type of component is *InComing*. In the reverse case, the risk type is *OutGoing*. The risk type presents whether a component is vulnerable to an incoming intent or its outgoing intent is unsafe. If no flow is found, the risk type is set to *null*. Finally, the extracted information is maintained per component as a component model (see Fig. 3). Note that *criticalmethod* exists only when the *risktype* is not *null*. and only explicit intent contains the information of *target-component*.

```
Component{ componentname=String; risktype=String;
*criticalmethod=String }
App{ packagename=String }
Intent{ action=String; category=String; data=Boolean;
scheme=String; *targetcomponent=String }
IntentFilter{ action=String; category=String;
data=Boolean; scheme=String }
Permission{ hold=String; required=String }
```

Fig. 3. A Component Model.

3.2 Identify Potential Risks

In this phase, potential risks between components are identified by checking component models. Unlike existing techniques [5, 6], checking component models is scalable to a number of apps as evaluated in Section 4. Our approach regards a risk exists between two components in different apps if one or both of them are unsafe components. Specifically, every component whose *risktype* is not *null* is examined based on two cases: (1) When its *risktype* is "*InComing,*" every component belonging to different apps is inspected whether it can access the component by sending an intent. The corresponding components' *intent* and *intentfilter* are matched in order to check if an access can be established. During the matching process, comparing *permission* between those components is also performed in case when the unsafe component holds particular permissions to limit an access. If a component can access to the unsafe

component, the risk between those two components is determined as "Intent Spoofing." (2) When its *risktype* is *"OutGoing"*, every component belonging to other apps is inspected to check whether it can receive the intent sent from the component, which may contain critical information. The inspection process is same as the first case (i.e., matching *intent*, *intentfilter*, and *permission*). Once a component is found, the risk between them is determined as "Intent Hijacking."

3.3 Assess Identified Risks

Once potential risks are identified, an end-user is required to assess the safety level of each risk. The assessment uses a 5-point Likert scale (1 = very safe, 2 = safe, 3 = neutral, 4 = unsafe, 5 = very unsafe). Since a static analysis may cause falsely identified results [4], an actual safety of each risk needs to be confirmed by an end-user. Furthermore, since the safety level between components can be different depending on the user-context, end-users' assessments on the same pair of components can be different. For example, if an end-user has developed two different apps and two components belonging to each of those apps which can access each other, she may regard the access as safe since she developed them herself. However, in case when an end-user downloaded the same apps from unreliable online sources, he may consider the access between the same pairs of components as unsafe.

3.4 Crowdsource Risk Assessments

In this phase, our technique leverages the crowd-based exploration of the risk assessments in order to recommend proper safety levels to an end-user. For non-expert users who cannot determine the safety levels, crowdsourced and refined information can aid their assessments. We adapted a user-based collaborative filtering [9] to identify appropriate assessments based on the similarity of safety levels between users. Specifically, for a user (=a) and the other user in the crowd (=$b \in$ Users), the similarity of safety levels between a and b (=$s(a, b)$) is computed using the cosine similarity of their safety level vectors, (see formula (1)). For each risk where the same pair of components is involved, ra and rb indicate the safety levels of a and b, respectively. For the remaining risks in which different components are involved, their level vectors are allocated as 0. Once the similarity is calculated, every user in the crowd is prioritized based on the similarities. As a user sets a similarity-level as k, k-nearest users' safety levels are recommended for each risk. The recommended safe levels can help an end-user confirm her assessment.

$$s(a,b) = \frac{\sum_{i=1}^{n} r_{a,i}\ r_{b,i}}{\sqrt{\sum_{i=1}^{n} (r_{a,i})^2}\ \sqrt{\sum_{i=1}^{n} (r_{b,i})^2}} \quad (1)$$

3.5 Control Inter-app Access

In the last phase, an access control policy is generated, which can control unsafe accesses at runtime. An end-user can set access control options for safety levels. For example, an end-user can set "always block" for an access whose safety level is ≥ 4, while setting "always allow" for the safety level ≤ 2. An access control policy contains two different types of information: (1) identified risks, each of which comprises the information of a pair of components that may cause an inter-app attack (e.g., *componentname, packagename* and *risktype*); (2) safety levels and corresponding access control options.

An access control policy is compatible with *Access Control Module* (*ACM*), an additional module which controls runtime access between apps. Two different types of alternatives exist for implementing *ACM* (i.e., instrumenting installed apps' bytecode and extending Android framework). Whenever an access is requested, *ACM* checks

whether its risk is specified on the policy. Specifically, when *Comp1* of *App1* tries to access *Comp2* of *App2* by sending an intent, ACM pauses the access and captures the information of *Comp1* and *Comp2*. It then checks if the risk of corresponding access is specified on the policy. If specified, *ACM* enforces Android's access controller (e.g., ActivityManager [7]) to handle it as the corresponding control option directs; otherwise, the access is processed.

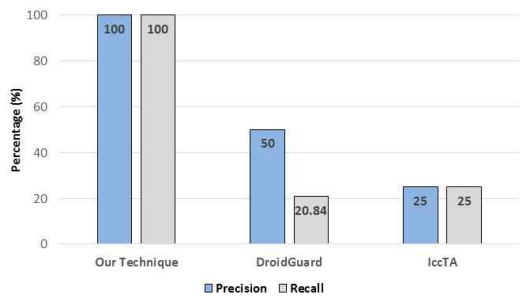


Fig. 4. Precisions and Recalls of Risk Identification on Real-world Apps.

4. EVALUATION

4.1 Accuracy of Risk Identification

Experimental Setup: To evaluate our technique's accuracy in risk identification, we selected 19 real-world apps that are exposed to inter-app attacks [3, 12-15]. Since some of those apps did not have preexisting malicious apps that implement inter-app attacks on them, we had to build some malicious apps ourselves. We also included 12 trick apps that contain vulnerable but unreachable components, whose identification falls under a false warning. In total, our test suite contained 25 potential risks (12 for intent spoofing and 13 for intent hijacking) between 19 vulnerable apps and 25 malicious apps that attack them. To confirm if an actual risk exists in each pair of apps (i.e., a vulnerable appand a malicious app), we manually inspected their source code and observed runtime behaviors via logcat [14], an Android debugging tool.

Results: We compared accuracy of our technique in potential risk identification with other techniques, IccTA [4] and DroidGuard [2], state-of-the-art tools for inter-app risk detection. Specifically, we measured those technique's precisions (i.e., whether the identified risks were actually vulnerable to inter-app attacks) and recalls (i.e., whether all potential risks in our test suite were identified).

As depicted in Fig. 4, only our technique illustrated perfect precision and recall in identifying potential risks. Our technique correctly ignored all trick cases as well. DroidGuard had 50% precision, but its recall was only 20.84%. This is mainly because DroidGuard targets individual surfaces (e.g., interfaces) rather than specific risk information between components, which hampers finer-grained characterization of risks, and returns risks only when both sender and receiver contain critical methods [8]. IccTA showed 25% precision and recall. This is because IccTA primarily targets a limited type of risk (i.e., intent hijacking). Moreover, since its app instrumentation does not support more than two apps simultaneously, it returns either inaccurate risk information or execution errors. Our evaluation has shown that, unlike other techniques, our finer-grained model-based analysis not only accurately detects potential risks, but is also applicable to analyze a set of real-world apps simultaneously.

4.2 Performance of Risk Identification

To prove our technique's scalability, we measured performance of its risk identification. Specifically, we checked (1) whether it incrementally analyzes the increasing number of apps and (2) whether it successfully analyzes a large number of apps. We created four test bundles each of which contains different number of apps (25, 50, 75, and 100). The apps were randomly selected from public repositories [14-16]. Considering the fact that an

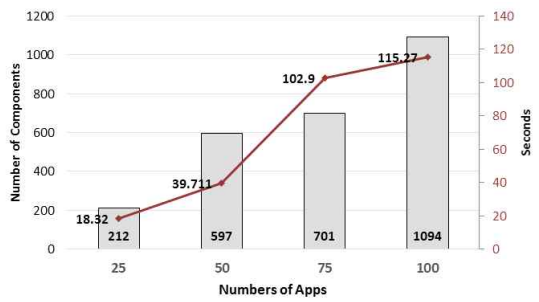


Fig. 5. The Number of Components and Analysis Time for Each Bundle.

average user uses 30 apps per month [15], the selected numbers fairly reflect the real-world usages. We ran our prototype tool on each bundle with a PC (Intel core i7 2.3GHz CPU and 8GB of RAM).

The average analysis time (especially for two phases: (1) construct component models and (2) identify potential risks) took 69.05 seconds. Fig. 5 depicts the number of analyzed components in each bundle (212, 597, 701, and 1,084) and the analysis time for each bundle (18.32*s*, 39.71*s*, 102.9*s*, and 115.27*s*). The results show that our technique is able to analyze a large number of apps in the order of few minutes (i.e., 100 apps containing 1,084 components within two minutes) on an ordinary PC and its analysis time is linearly scalable to the number of apps, confirming that the proposed technique is feasible.

4.3 Effectiveness of Crowdsourcing

We conducted a user study to evaluate (1) the correctness of crowdsourced safety levels and (2) the differences among individual, crowdsourced, and revised assessments.

Experimental Setup: The user study included 30 non-expert participants, all of whom were graduate students at the University of Southern California and whose majors spanned business, communication, social work, and science. The participants have used Android devices for 28 months on average and were aged between 18 and 34. 15 of the participants had experiences using Android as their primary mobile platform; 15 had not used Android previously. None of them had experience in mobile software programming or analysis. We provided each participant an Android device (i.e., Google Nexus 7) with 15 pre-installed apps that were built by authors. The app types spanned various categories (e.g., game, SMS, and GPS) and were designed to launch actual inter-app attacks. Among those 15 apps, ten different potential risks existed, whose safety level should be considered as ≥ 4. Firstly, the participants were asked to use each app for five minutes. Then our prototype presented potential risks between those apps, and asked the participants to assess their safety levels. Once the initial assessments completed, our proto-

Table 1. The Result of Risk Assessments

	1	2	3	4	5	CR	R
r1	0.00	6.67	0.00	10.00	83.33	4.70	5
r2	0.00	0.00	0.00	0.00	100.00	5.00	5
r3	10.00	0.00	6.67	6.67	76.67	4.40	4
r4	0.00	0.00	0.00	0.00	100.00	5.00	5
r5	0.00	0.00	6.67	0.00	93.33	4.87	5
r6	0.00	16.67	16.67	0.00	66.67	4.50	5
r7	0.00	0.00	3.33	33.33	63.33	4.17	4
r8	0.00	0.00	20.00	6.67	73.33	4.53	5
r9	0.00	0.00	23.33	6.67	70.00	4.47	4
r10	0.00	0.00	0.00	0.00	100.00	5.00	5

(1=very safe, 2=safe, 3=neutral, 4=unsafe, 5=very unsafe, CR=crowdsourced, R=rounded-off)

type provided them with the crowdsourced safety levels and let them revise their initial assessments if they wanted.

Results: As for the crowdsourced safsety levels, an average safety level for each risk (r1-r10) was calculated and rounded off. As shown in Table 1, every crowdsourced safety level is correct (i.e., ≥4). Table 1 also depicts the percentage of safety levels for each risk. In total 11% (1% for very safe; 2.33% for safe; 7.67% for neutral) of risks were assessed as 1-3 (from *very safe* to *neutral*), which is considered as incorrect. However, after the crowdsourced information was provided, 10% of incorrect assessments were corrected to be *unsafe* (≥4), which supports the conclusion that 10% of potential risks could be minimized by providing crowdsourced risk assessments.

5. RELATED WORK

A large volume of research has focused on detection of security vulnerabilities in Android apps [16]. ComDroid [1] statically analyzes an app to identify inter-app vulnerabilities. IccTA [4] is a static taint analyzer that identifies privacy leaks among components. These techniques support only a single app analysis or limited attack types.

App instrumentation and framework extension are largely employed for runtime prevention of inter-app risks. DroidForce [5] instruments target app's bytecode in order to enforce data-centric policies. DroidGuard [2] synthesizes automatically-generated security policies on the set of apps. XmanDroid [3] enforces Android framework to control inter-app access as a permission-based policy specifies. While app instrumentation is relatively easier to implement, modification of bytecode may result in errors or additional attacks [17]. Meanwhile, Android framework extension provides a solid access control, which, however, requires expertise in its adoption (e.g., re-installation of Android framework on the device).

6. CONCLUSIONS

This paper proposed a new technique for minimizing the risks of inter-app attacks by harmonizing a static analysis and crowdsourcing. Our technique's model-based analysis accurately identifies potential risks from a given set of apps and its crowdsourced information, and effectively aids end-users' assessments on the identified risks. We successfully evaluated our technique's accuracy, performance, and effectiveness. Our evaluation demonstrated that our technique outperforms the existing techniques in identifying potential risks from a set of real-world apps, its analysis time is linearly scalable to the number of apps, and it enables an end-user's precise risk assessments.

There are a number of research challenges remaining for future work. We can build a statistical model based on the user profiles or each app's popularity to provide more specific guidance to end-users regarding risk assessments.

REFERENCES

[1] E. Chin, A.P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," *Proceeding of the International Conference on Mobile Systems, Applications, and Services*, *Mobisys*, pp. 239-252, 2011.

[2] COVERT, http://www.ics.uci.edu/~seal/projects/covert/ (Accessed 15 September 2016).

[3] S. Bugiel, L. Davi, R. Dmitrienko, and T. Fischer, "Towards Taming Privilege-Escalation Attacks on Android," *Proceeding of the Annual Network and Distributed System Security Symposium*, 2012. (No Page Info)

[4] L. Li, A. Bartel, J. Klein, Y. L. Traon, and S. Arzt, et al. "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis," No. ISBN: 978-2-87971-129-4, 2014.

[5] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden,

"DroidForce Enforcing Complex, Data-Centric, System-Wide Policies in Android," *Proceeding of the International Conference on Availability, Reliability, and Security,* pp. 8-12, 2014.

[6] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis," *Proceeding of the International Conference on Software Engineering*, pp. 77-88, 2015.

[7] Android. App | Android Developers, http://developer.android.com/reference/android/app/package-summary.html (Accessed 10 September 2012).

[8] SuSi, https://github.com/secure-software-engineering/SuSi (Accessed 10 September 2012).

[9] Q. Ismail, T. Ahmed, A. Kapadia and M.K. Reiter, "Crowdsourced Exploration of Security Configurations," *Proceeding of Annual ACM Conference on Human Factors in Computing Systems,* pp. 467-476, 2015.

[10] DroidBench - Benchmarks, https://github. com/secure-software-engineering/DroidBench (Accessed 10 September 2012).

[11] F-droid, https://f-droid.org/ (Accessed 10 September 2012).

[12] Google Play, http://play.google.com/store/apps/ (Accessed 10 September 2012).

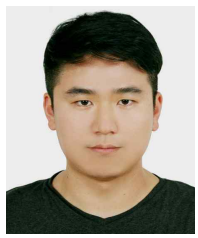
[13] Y. Zhou and X. Jiang, "Dissecting Android Malware Characterization and Evolution," *Proceeding of the IEEE Symposium on Security and Privacy*, pp. 22-23, 2012.

[14] Logcat | Android Developers, http://developer.android.com/tools/help/logcat.html (Accessed 10 September 2012).

[15] So Many Apps, So Much More Time for Entertainment, http://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html (Accessed 10 September 2012).

[16] H. Kim and J.Choi, "Research on Secure Coding and Weakness for Implementation of Android-based Dynamic Class Loading," Journal of Korea Multimedia Society, pp. 1792-1807, 2016.

[17] H. Hao, V. Singh, and W. Du, "On the Effectiveness of API-Level Access Control Using Bytecode Rewriting in Android," *Proceeding of the Symposium on Information, Computer and Communications Security,* pp. 25-36, 2013.



Youn Kyu Lee

He received the B.S and M.S. degrees in Computer Science from Korea University, Korea. He is currently a Ph.D. candidate at the Department of Computer Science, University of Southern California, USA. His research interests include software architecture and requirements engineering.

Tai Suk Kim

He received his B.S. degree from the department of electrical engineering, Kyungpook National University in 1981 and his M.S. and Ph.D. degrees from the department of computer science, Keio University in 1989 and 1993, respectively. Since 1994, he has been a faculty member of the Dong-eui University, where he is now the professor in the department of software engineering. His current research interests are information system and Internet business.