Advances in Engineering Research (AER), volume 148
3rd Workshop on Advanced Research and Technology in Industry Applications (WARTIA 2017)

Instrumentation Based Dynamic Android Application Security Protection System

Ming Li[1,a], Zhikang Piao[2,b] and Yong Wang[2,c,*]

[1] The Third Research Institute of the Ministry of Public Security, Shanghai, China

[2] 58 Corp, Beijing 100029, China

[3] School of Software, Beijing Institute of Technology, China;

[a] li.m@vip.163.com, [b] 764780373@qq.com, [c] wybit@aliyun.com

*corresponding author

Keywords: Android security, Instrumentation, Permission usage behaviour, Fine-grained access control

Abstract: Nowadays, smart phones have converted the people's life style; convenience of the application has a big impact on all aspects of necessities. Considering that applications need to keep sensitive and private data, the trustworthiness of the application becomes a matter of concern to end users, and privacy protection is an urgent need. The current Android permission mechanism has the goal of fast, simple, and easy enforcement, so it designed to be a coarse-grained permission mechanism. In this paper, a fine-grained access control model for Android applications, which makes authorization decisions according to the permission usage behaviour and sensitive data accessed, is proposed. Furthermore, the fine-grained access control model is integrated into actual APPs through a so-called dynamic Android Application Security System (dAASS), which make use of instrumentation technology, static analysis and dynamic analysis technology. dAASS can enhance the security of Android system by fine-grained access control, permission usage tracking and security violation reminding. Performance experiment shows that deploying dAASS on Android applications takes less than 1 minute and secured applications exhibit no noticeable slowdown and Application Not Responding (ANR). dAASS can provide fine-grained real-time protection from malicious Android applications.

1. Introduction

Android permission mechanism, as the main mechanism of access control for application in Android framework lev-el, has been widely criticized for its coarse access control. Android is mainly used in mobile phones, tablet PCs, smart watches and other mobile devices. However, the permission mechanism does not make specific design for the characteristics of mobile devices, and it cannot effectively protect the privacy of end users. For Example, an application re-quests the SEND_SMS permission, and users install it with-out security consideration, the application may have the authority to send SMS arbitrary. As for the widely-used permission – INTERNET, Feizollah A et al analysed 7406 applications and found that nearly 98% of those applications had INTERNET permission [1]. The applications that have INTERNET permission, can send request to all the HTTP or HTTPS domains, and connect to any host and port. Much research on optimizing and improving the An-droid permission mechanism have been carried out [2-5].

However, there are some weakness in existing research. First, the access control policy did not focus on permission usage behaviour and sensitive data. Secondly, the extension of permission mechanism mainly improved the ability for users to configure permissions during installation, while seldom payed attention to the runtime permissions usage. Although, Android Marshmallow (API 23 v6.0) updated Android's permission model, it placed heavy burden on application developers to insert permission checking code. Finally, modifying Android source code might have negative effect on the performance of application, such as crash or ANR.

Copyright © 2017, the Authors. Published by Atlantis Press.
This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

The remainder of this paper is structured as follows: firstly, we discuss the architecture of dAASS(Section 2), followed by details about the fine-grained access control policy and implementation of dAASS(Section 3); then we evaluate the performance and security of the dAASS in Section 4; afterward, in Section 5 we conclude and discuss the future work.

2. System Design

2.1. Components of dAASS

dAASS consists of four main architecture components: The Policy Decision Point(PDP), the Policy Enforcement Point(PEP), the Run-time Permission Analysis Tool(RPAT), and the Permission Usage Behaviour Record Server(PUBRS). Figure 1 shows an overview of dAASS's architecture. The PDP is a central independent application responsible for storing and managing user-defined access control policies like those described in 2.2 and sending authorization decisions to PUBRS. The PEP is in charge of intercepting per-mission usage point and prohibiting permission usage behaviours that violate an access control policy. PEP is instrumented into target application's permission usage point. RPAT is responsible for tracking run-time permission behaviours, and sending it to PUBRS. The PUBRS is a server receiving and recoding the permission behaviours from RPAT and PDP. In addition, PUBRS has a security reminder function that helps users configure his/her access control policies. The final sentence of a caption must end with a period.

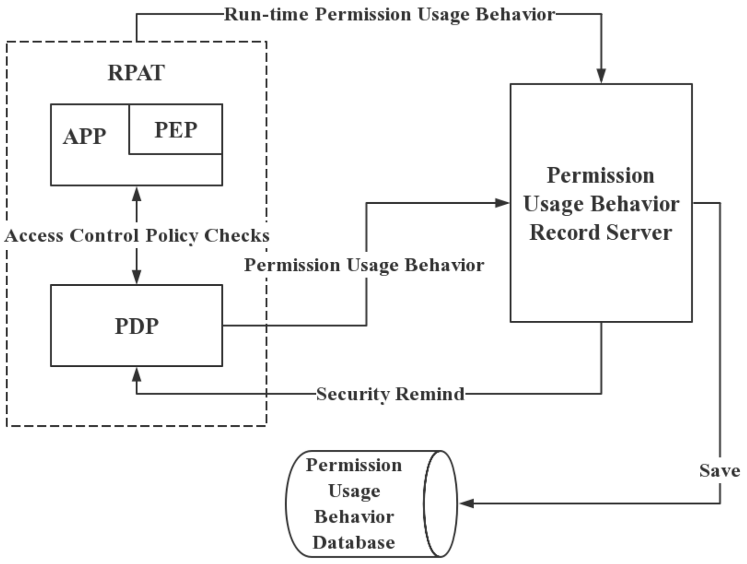

Run-time Permission Usage Behavior
RPAT
APP
PEP
Access Control Policy Checks
PDP
Permission Usage Behavior
Permission Usage Behavior Record Server
Security Remind
Save
Permission Usage Behavior Database

Figure 1 dAASS's Architecture.

Whenever a potential dangerous permission usage behaviour like sending SMS is about to occur in an application, PEP generates a PDP request, like "Application A is about to send SMS message with text X to phone number Y". All requests are transmitted to the PDP. Only if the PDP confirms that the behaviour is allowed under the current ac-cess control policy, the PEP allows the respective application to continue. If the PDP denies the execution of the behaviour, the protected behaviour is skipped, and the execution resumes from the next instruction in the application code. In the meantime, PDP uploads those permission usage behaviours to PUBRS. Furthermore, if RPAT also gets those behaviours, it will upload them to PUBRS as well. PUBRS may push those behaviours to PDP as security reminder message, after PUBRS gets those permission usage behaviours. Security reminder message can help users update their access control policies accordingly. Finally, PUBRS saves run-time permission usage behaviours and permission usage behaviours from PDP in database by permission category.

Executing dangerous permission usage behaviours such as sending SMS messages, or opening Internet connections are done via calls to permission-related API methods in Android applications. During the instrumentation phase, dAASS uses Soot to wrap permission usage point with policy checks that (1) generate a request for the PDP, (2) send it, and (3) only execute the permission usage behaviour if the response is positive (allow). RPAT and PDP can optionally specify additional compensative actions to take in response to at-tempted violations, such as reporting the permission usage behaviour, sending security reminder message or executing user-defined code. This feature is useful as it allows policies to take corrective measures like reporting targeted attacks on company smartphones to the IT security department or automatically blocking and uninstalling the respective applications via the company's mobile device management (MDM) solution. The code can either be executed in the context of the application itself, allowing for further inspection into the monitored application's state, or in the context of the PDP application, which is a secured environment.

The PEP must be able to intercept dangerous permission usage behaviours in all applications on the phone. dAASS therefore instruments a decentralized PEP into every user application by using Soot[6] during a static pre-processing step on a desktop computer or a server before the respective application is deployed on the device. On this server, dAASS first analyses the application for data flows using the FlowDroid [7] data flow tracker. FlowDroid is a context, flow, field, object-sensitive and lifecycle-aware data flow tracking tool, and is one of the most precise data flow trackers currently available. The statically extracted dependencies between sources and sinks are instrumented into the application in form of a table, alongside the code responsible for policy enforcement. After the instrumented application is installed on the user phone, the enforcement function can interact with the PDP at runtime to check whether permission usage behaviour shall be allowed or prohibited. The data flow table is queried to obtain the origin of the data being transmitted across components or applications, in order to generate proper PDP requests. RPAT is modified from Droidbox [8] source code, Droidbox is developed to offer dynamic analysis of Android applications. PUBRS is designed by using Django [9], Django is the web framework for perfectionists with deadlines.

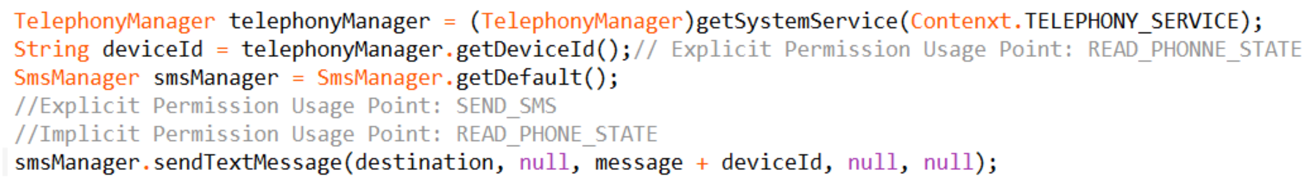
2.2. dAASS Fine-grained Access Control Policy

dAASS uses fine-grained access control policy based on permission usage behaviours and sensitive data. The definition of permission usage behaviour and sensitive data are introduced in following sections (2.2.1 and 2.2.2), and the principle of dAASS Policy is introduced in 2.2.3.

2.2.1. Permission Usage Behaviours

Permission usage behaviours can track the behaviour of using permission protected system resources inside the application. Based on the life cycle of using system resources inside the application, we can definite different kinds of Permission Usage Point.

Firstly, an application need to call permission-related API (Application Programming Interface) to request access to system resources during Resource Request Phase. If the requesting resources are protected by some permission, An-droid permission enforcement system will check whether the application is granted related permission. In this paper, the location where the application calls API to request protected system resources is called Explicit Permission Usage Point. After Resource Request Phase, depending on the API used when requesting resources, the system resources may be delivered synchronously or asynchronously to the application, which is called Resource Delivery Phase. Finally, after the requested resource is delivered to the application, the application enters the Resource Usage Phase. The application handles the system resource with specific logic, that is, the behaviour of using sensitive resources inside the application. We refer to these internal usages of sensitive resources as Implicit Permission Usage Point. The examples of Permission Usage Point is shown in Figure 2.



```
TelephonyManager telephonyManager = (TelephonyManager)getSystemService(Contenxt.TELEPHONY_SERVICE);
String deviceId = telephonyManager.getDeviceId();// Explicit Permission Usage Point: READ_PHONNE_STATE
SmsManager smsManager = SmsManager.getDefault();
//Explicit Permission Usage Point: SEND_SMS
//Implicit Permission Usage Point: READ_PHONE_STATE
smsManager.sendTextMessage(destination, null, message + deviceId, null, null);
```

Figure 2 Example of Permission Usage Point.

In summary, Explicit Permission Usage Point can track which and where permissions are used, and Implicit Permission Usage Point can track how the application uses these permissions to implement specific logic. Based on Explicit Permission Usage Point and Implicit Permission Usage Point, this paper gives a formal definition of *Permission Usage Behaviour*.

Definition: a Permission Usage Behavior is a function call diagram G = (V, ε, α) on permission set ρ, and meets the following conditions. V is the collection of function nodes; ε is the collection of edge and α is the label function.

$$V = V\_explicit \cup V\_implicit \tag{1}$$

$$\varepsilon \sqsubset V \times V$$

$$\alpha = V \rightarrow \rho$$

The collection of function nodes V should contain all of Explicit Permission Usage Point and Implicit Permission Usage Point; each edge only connects the function node using the same permissions; α denotes the permission information used by each function node.

Through using Permission Usage Behaviour to abstract the interaction behaviour between applications and Android system, the Permission Usage Behaviour can effectively describe how the application requests system resources and how the application uses the acquired system resources.

2.2.2. Sensitive Data

System permissions are divided into several protection levels. The two most important protection levels to know about are normal and dangerous permissions.

Normal permissions are used when your app needs to access data or resources outside the app's sandbox, but where there is very little risk to the user's privacy or the operation of other apps. For example, permission to set the time zone is a normal permission. If an app declares that it needs a normal permission, the system automatically grants the permission to the app.

Dangerous permissions are used when the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps. For example, the ability to read the user's contacts is a dangerous permission. If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app.

dAASS Policy mainly aims at dangerous permissions and the related sensitive data. Table 1 shows the dangerous permissions and permission groups and related sensitive data.

Table 1 Dangerous permissions and permission groups.

Permission Group	Permissions	Sensitive Data
CALENDAR	READ_CALENDAR WRITE_CALENDAR	Calendar info
CAMERA	CAMERA	Camera info
CONTACTS	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS	Contacts
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION	GPS
MICROPHONE	RECORD_AUDIO	IMEI etc
PHONE	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS	
SENSORS	BODY_SENSORS	Sensors data
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS	SMS info
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE	Storage info

2.2.3. Fine-grained Access Control Policy

dAASS Policy is fine-grained by making good use of Permission Usage Behaviours and sensitive data. Considering the previous research [6], we decided to use OSL (Obligation Specification Language) [10] as policy specification language. OSL is a first order temporal-logic language (e.g. “A until B”), with support for cardinality (e.g. “A at most 3 times”) and time constraints (e.g. “A within 30 seconds”). Policies are in form of Event-Condition-Action rules: if a system event E is detected and allowing its execution would make OSL condition C true, then action A should be performed. Action A states whether the execution of the event should be allowed or not, and if any additional action should be executed. Additional actions could for instance include asking the user for permission or reporting the violation via a popup window. By default, any event is allowed, unless a mechanism requires its prohibition.

(1) A fine-grained policy based on Permission Usage Behaviour

For instance, an appropriate policy could be “no more than 2 SMS messages per day may be sent to 12345”. In our framework, this translates into

E=sendTextMessageto12345   C=not (replim(24,0,1,E))   A=Inhibit

The semantics of the *not* operator is intuitive, whereas replim (time, min, max, φ) is true if φ has been true in the past time at least min times and at most max times. Otherwise, the operator evaluates to false. In this example, a message to +01-234-5678 can be sent if, in the previous 24 hours, other SMS to the same number have been sent at least 0 times and at most one time. Recall that in order for the trigger event to be allowed, the condition must evaluate to false, which explains the presence of the not operator.

(2) A fine-grained policy based on Sensitive Data

For instance, an appropriate policy could be “no SMS message with IMEI may be sent to 12345”. In our framework, this translates into

E=sendTextMessage to 12345   C=(parameter IMEI_DATA =true) $\wedge$ eventually E   A=Inhibit

Where *eventually* means “at least once so far” and if the other SMS message with IMEI has been already sent in the past, then the event should be forbidden.

(3) A fine-grained policy based on both Permission Usage Behaviour and Sensitive Data

For instance, an appropriate policy could be “no more than 2 SMS messages per day may be sent to 12345 and no SMS message with IMEI may be sent to 12345”. In our framework, this translates into

E=sendTextMessage to 12345 *C_data= (parameter IMEI_DATA=true)* $\wedge$ *eventually E*

C_event= not(replim(48,0,1,E)) *C=C_event* $\wedge$ *C_data* *A=Inhibit*

Considering both Permission Usage Behaviour and Sensitive Data, we only need to do is logical *C_event* AND *C_data* to make a new condition *C*.

2.2.4. XML Expression of OSL policy

The OSL language allows access control policies to be dynamically deployed by changing Event-Condition-Action tuples, and can generate so many Conditions that make access control policies flexible. However, OSL also has its own shortcomings, that is, OSL has high learning costs, and the user experience is poor.

Since the OSL language is abstract and not conducive to integrating into the system, this paper uses XML to represent OSL language. XML is cross platform, independent of the development language and has a strong structure. XML can be used to mark information in a simple way, so it is easy to parse.

3. Implementation

3.1. PEP

The PEP is instrumented into target application using Soot framework and FlowDroid static analysis tool. FlowDroid is used to extract *Permission Usage Point*, and instrument PEP at *Permission Usage Point*. PEP consists of major components, the *Policy Enforcement Function* and the *Auxiliary Classes*.

Policy Enforcement Function is in charge of intercepting permission usage point and prohibiting permission usage behaviour if allowing it would violate an access control policy.

Auxiliary Classes is in charge of interacting with PDP to send permission usage request to PDP and delivering results from PDP to the policy enforcement function. The auxiliary classes are also instrumented into target application.

Figure 3 shows how the policy enforcement function and the auxiliary classes work together with PDP. Firstly, the auxiliary classes connect to PDP, whenever a Permission Usage Behaviour will occur in an application. The policy enforcement function at related Permission Usage Point intercepts Permission Usage Behaviour and sends the behaviour parameters to the auxiliary classes. The auxiliary Classes send request to PDP and get the decision from PDP. Then, the auxiliary classes call the policy enforcement function to allow or prohibit the behaviour.

The interaction between PDP and PEP uses Android’s IPC (Inter-Process Communication) mechanism called Messenger [11]. Figure 4 shows the principle of Messenger.

3.2. PDP

The PDP is a central independent application responsible for storing and managing user-defined access control policies and sending related permission behaviour to PUBRS. The PDP consists of three components, Policy Deploy Module, Policy Decision Service Module, and Policy Decision Module. Figure 5 shows the workflow of PDP modules.

The Policy Deploy Module reads a policy file and configures it by sending a message to the Policy Decision Service Module. The Policy Decision Service Module is a service handling the permission usage request from PEP, the policy deployment request from the Policy Deploy Module and delivering it to the Policy Decision Module. The Policy Decision Module is in charge of making decisions according to user-defined policies, sending result back to PEP and uploading related Permission Usage Behaviour to PUBRS.

3.3. RPAT

RPAT is responsible for tracking run-time permission related behaviours, and sending it to PUBRS. RPAT is developed using Droidbox scripts. Figure 6 shows the workflow of RPAT, RPAT optimizes Droidbox’s static check process and output.

Firstly, RPAT uses Apktool for decompiling apk files instead of AXMLParser used in Droidbox scripts. The reason is that AXMLParser does not support decompiling the apk file that is instrumented PEP. Apktool is a tool for re-engineering Android APK files quickly and easily for many operations no matter their nature and legal-wise actions.

Secondly, RPAT reformats Droidbox’s json output to the format defined in PUBRS’s database in order to make the Permission Usage Behaviour from PDP Compatible with that from RPAT. The same data structure is helpful for PUBRS to save Permission Usage Behaviour by permission category.

3.4. PUBRS

PUBRS has two main functions, one is saving the Permission Usage Behaviour uploaded from PDP and RPAT, the other is security reminder. We use Django as the web framework to develop PUBRS, because of its convenience for configuring database such as MySQL, SQLite and so on.

The PUBRS uses MySQL to save Permission Usage Behaviour. The security reminder function is that whenever the Permission Usage Behaviour is about to occur, PUBRS sends it to PDP in real time. The users can be informed what permission protected behaviours are requested by the application (no matter PDP allows or prohibits). This approach can make Permission Usage Behaviour more intuitively and easy to use since the users do not need to know what system resources the permission protects.

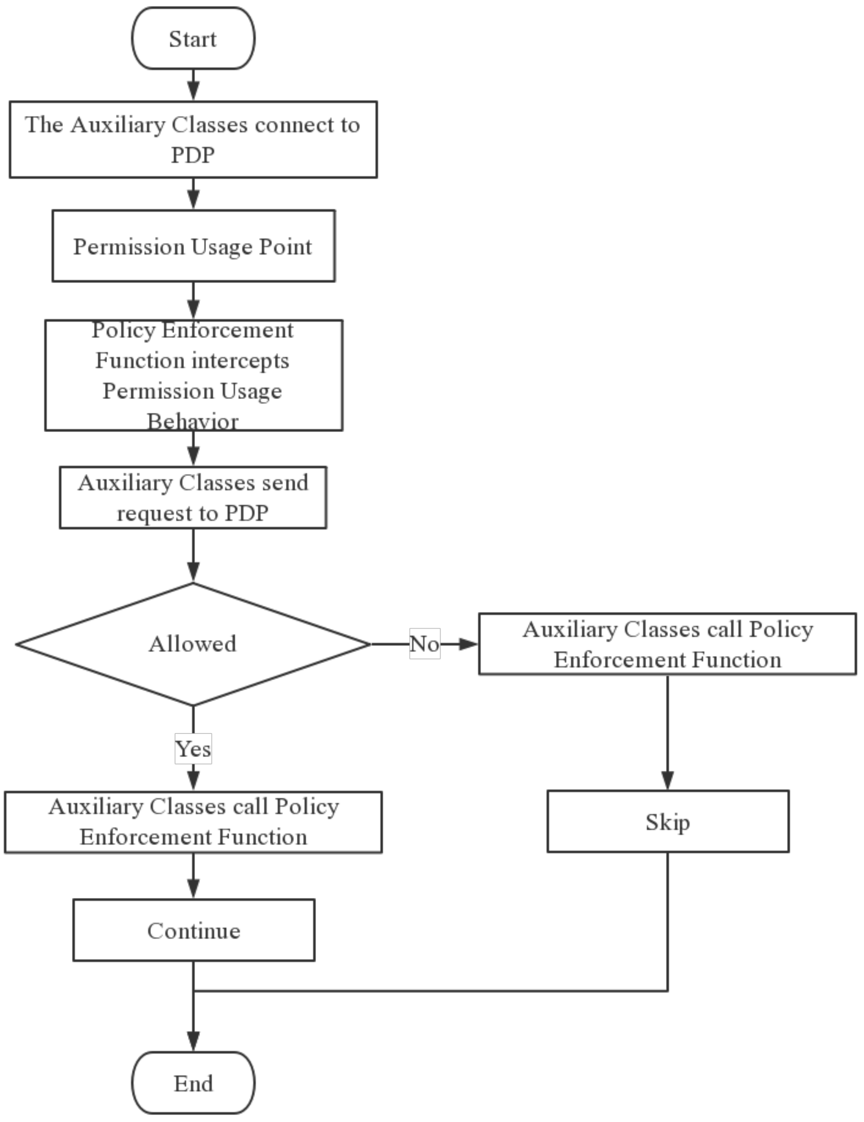

Start
The Auxiliary Classes connect to PDP
Permission Usage Point
Policy Enforcement Function intercepts Permission Usage Behavior
Auxiliary Classes send request to PDP
Allowed
No
Auxiliary Classes call Policy Enforcement Function
Skip
Yes
Auxiliary Classes call Policy Enforcement Function
Continue
End

Figure 3 The workflow of PEP.

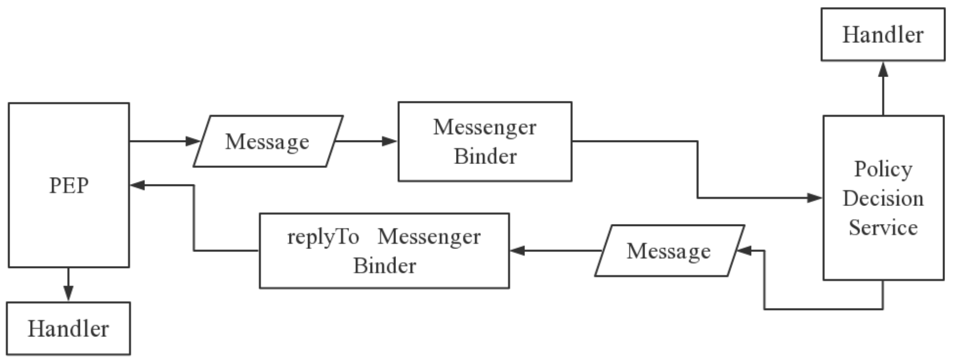


Figure 4 The process of Messenger.

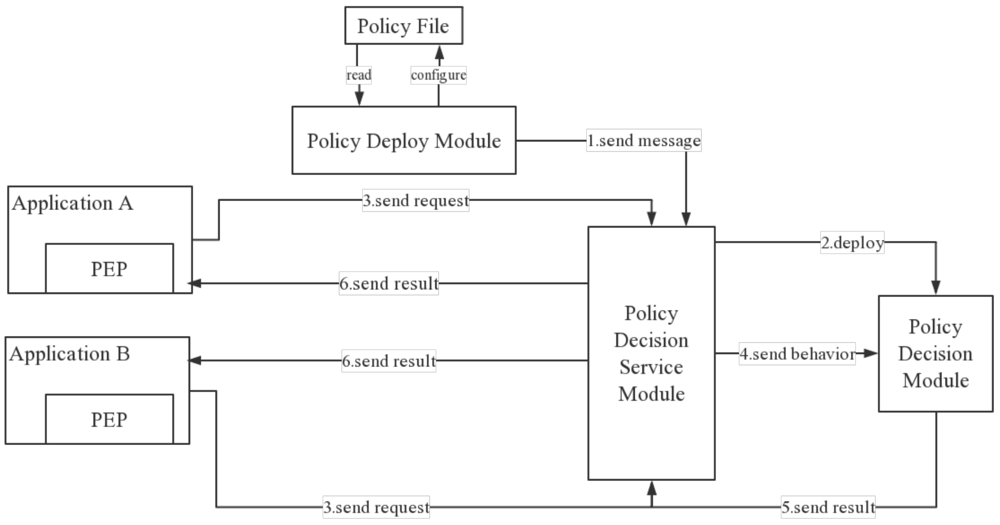


Figure 5 The workflow of PDP.

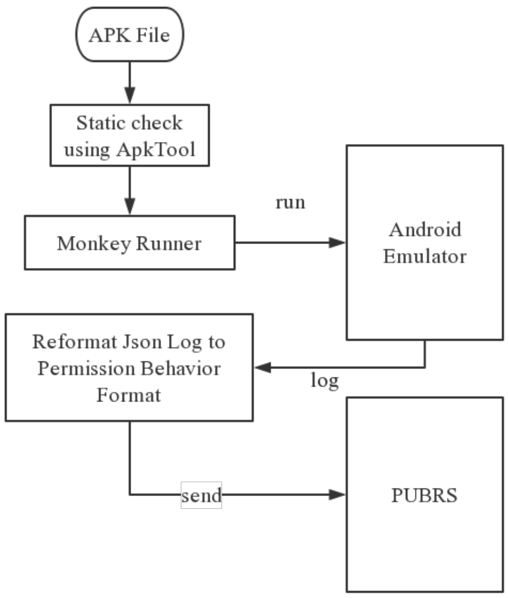

APK File
Static check using ApkTool
Monkey Runner
run
Android Emulator
Reformat Json Log to Permission Behavior Format
log
send
PUBRS

Figure 6 The workflow of RPAT.

4. Evaluation

In this section, we evaluate dAASS. Section 4.1 discusses the effectiveness of dAASS by doing comparative experiment and multi-application deployment experiment, whereas in Section 4.2 deals with dAASS's performance.

The tests were carried out on a Dell computer running Ubuntu version 14.04 on a 2.2 GHz Intel Core i5 processor and 8 GB of memory. The Android emulator configuration simulates the Nexus 4 with Android 4.1.2(API 16), 768 * 1280xhdpi, the CPU is ARM (armeabi-v7a), the memory is 1.5G, the heap is configured as 512M, the internal storage size is 2G, and the SD card is 2G.

The target app is a SMS sender, which can send SMS message with IMEI or without IMEI. The International Mobile Equipment Identity or IMEI [12] is a number, usually unique, to identify 3GPP (i.e., GSM, UMTS and LTE) and iDEN mobile phones. IMEI is a kind of sensitive data related with dangerous permission READ_PHONE_STATE.

4.1. Effectiveness

Comparative Experiment uses two Android emulators with the same configuration, one called D and the other called A. Then we deploy dAASS on D, and configure an appropriate policy could be "no SMS message with IMEI may be sent to 12345" using PDP as shown in Figure 7.

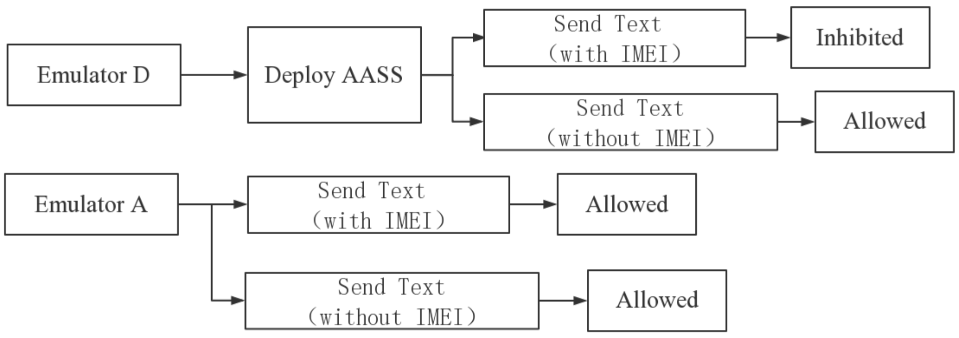


Figure 7 Comparative Experiment Process.

The result shows that the SMS messages with IMEI cannot be sent to D on which was dAASS deployed. However, the SMS messages with IMEI can be sent on A - the original Android emulator.

Multi-Application Deployment Experiment uses three applications, these three applications have the same function that is sending SMS messages (All the three apps contain calls to sendTextMessage()) as shown in Figure 8. Then we deploy dAASS on D, and configure an appropriate policy "no more than 2 SMS messages per day may be sent to 12345" using PDP.

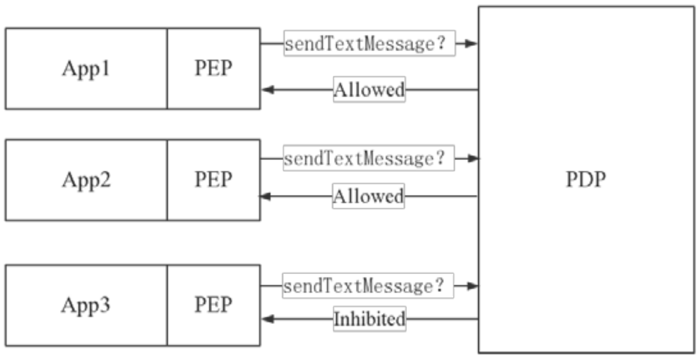


Figure 8 Multi-Application Deployment Experiment

When the first application asks PDP for permission, the PDP's counter is still zero, so it gets incremented and the request is allowed. The second application's request is also allowed and the counter is increased once more. Since the policy only allows at most two SMS messages to be sent, the request made by the third app (assuming it happens within 24 hours from the first request) is prohibited and no further SMS message is sent until 24 hours from the first request has passed.

In summary, the first experiment shows the effectiveness of sensitive data based policies of dAASS, the second experiment shows the effectiveness of Permission Usage Behaviour based policies of dAASS. During the experiment, PDP can receive the security reminder from PUBRS, and RPAT can get run-time permission usage behaviour.

4.2. Performance

During the experiments, none of applications exhibited any perceivable slowdown at the aspect of user experience, and none of the applications crashed, partially thanks to the additional initializations injected by dAASS. Based on these preliminary experiments we have good reasons to believe that our approach is feasible on real-world applications.

5. Conclusion

The main contribution of this paper is a dynamic Android Application Security System (dAASS) with fine-grained access control policy based on Permission Usage Behaviour and sensitive data (for example, to prohibit a particular phone number to send more than 2 times SMS). No modifications to the Android operating system are required except RPAT implementation, nor rooting the phone. The PEP code is instrumented into applications in a very short time and introduces no perceivable delay or ANR at runtime. We thus consider dAASS as an important step towards Android application security.

As future work, we plan to extend dAASS to automatically re-instrument applications. This can be solved by updating applications through a trusted instrumentation server on the Internet. Furthermore, we also plan to improve the ease of using RPAT, RPAT is based on Droidbox that can only run in Android emulator. This can be solved by developing RPAT directly based on TaintDroid[13], that can run in Android mobile phone.

References

[1] Feizollah A, Anuar N B, Salleh R, et al. AndroDialysis: Analysis of Android Intent Effectiveness in Malware Detection. Computers & Security, 2016, 65(C):121-134

[2] Rosen S, Qian Z, Mao Z M. AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users. ACM Conference on Data and Application Security and Privacy. ACM, 2013:221-232

[3] Pandita R, Xiao X, Yang W, et al. WHYPER: towards automating risk assessment of mobile applications. Usenix Conference on Security. USENIX Association, 2013:527-542

[4] Jeon J, Micinski K K, Vaughan J A, et al. Dr. Android and Mr. Hide: fine-grained permissions in android applications. ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. 2012:3-14

[5] Lam P, Bodden E, Lhotak O, et al. The Soot framework for Java program analysis: a retrospective. 2011.

[6] Rasthofer S, Arzt S, Lovat E, et al. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. Ninth International Conference on Availability, Reliability and Security. IEEE Computer Society, 2014:40-49

[7] Arzt S, Rasthofer S, Fritz C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices, 2014, 49(6): 259-269.

[8] Desnos A, Lantz P. Droidbox: An android application sandbox for dynamic analysis (2011). URL https://code. google. com/p/droidbox, 2014.

[9] Django. https://www.djangoproject.com/, 2017.

[10] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter, “A policy language for distributed usage control,” in Proc. 12th Europ. Conf. on Research in Computer Security. Springer-Verlag, pp. 531–546.

[11] Messenger. https://developer.android.com/reference/android/os/Messenger.html , 2017.

[12] IMEI. "3GPP TS 22.016: International Mobile Equipment Identities (IMEI)" (ZIP/DOC; 36 KB). 2009-10-01. Retrieved 2009-12-03

[13] Enck W, Gilbert P, Han S, et al. TaintDroid. ACM Transactions on Computer Systems, 2014, 32(2):1-29.