Hindawi
Mobile Information Systems
Volume 2020, Article ID 3407437, 16 pages
https://doi.org/10.1155/2020/3407437

Research Article

Multifeature-Based Behavior of Privilege Escalation Attack Detection Method for Android Applications

Limin Shen,[1] Hui Li,[1,2] Hongyi Wang,[3] and Yihuan Wang[1]

[1]*School of Information Science and Engineering, Yanshan University, Qinhuangdao 066004, China*
[2]*School of Business Administration, Hebei Normal University of Science &Technology, Qinhuangdao 066004, China*
[3]*Department of Commerce and Trade, Qinhuangdao Vocational and Technical College, Qinhuangdao 066100, China*

Correspondence should be addressed to Hui Li; lh_23@163.com

Received 5 October 2019; Revised 19 March 2020; Accepted 15 May 2020; Published 5 June 2020

Academic Editor: Salvatore Carta

This study proposed an application behavior-detection method based on multifeature and process algebra for detecting privilege escalation attacks in Android applications. The five features of application that constituted the attack were determined through an analysis of the privilege escalation attack model. On the basis of the extraction of multiple features, process algebra was used to build the application-behavior model and the attack model. Strong equivalence relation was used to verify the application behavior. Finally, dataflow path detection is conducted among the applications that can constitute privilege escalation attacks to determine those apps constituted a privilege escalation attack. The accuracy and effectiveness of the proposed method were verified using the DroidBench benchmark test and the test set that includes 55 APKs of 22 types.

1. Introduction

With the development of Mobile Internet and the popularization of smartphones, the number of downloads of applications has also increased significantly [1]. The Android system with open-source applications has become popular among smart device manufacturers and developers. It is widely used in finance, government, transportation, education, military, automobile, home, energy, and other important fields, and it has a large user base [2, 3]. According to statistics obtained by Strategy Analytics, Android accounted for 88% of the global smartphone market in the third quarter of 2016 [4]. Android system is widely used in the industrial Internet of Things (IIoT) [5]. Although there are classical solutions from the nonrepudiation record of the network to the access control of the Internet of Things (IoT) [6–9], the security of the android operating system has been a subject of concern, specifically in the mobile Internet privacy leak issue. The Nokia Threat Intelligence Report-2019 indicates that, in 2018, the average monthly infection rate in mobile networks was 0.31%, with Android devices being responsible for 47.15% of the observed malware infections [10]. According to the data collected in the Android Security Eco-Environment Research in 2018, the China Mobile Security Eco-Research Report in 2018, and the Android Malware Annual Special Report, 360 Internet Security Centers intercepted about 4.342 million new samples of malware on mobile terminals, and known security vulnerabilities have been found in 99.3% of Android applications in the last year, with 89.6% of them having high-risk security vulnerabilities. The cross-border behavior of app permissions shows an increasing integral trend. Moreover, 98.8% of the apps that applied for privacy permissions abused the permission to write call records [11–13]. Usually, the permissions applied for apps involve users' privacy and even threaten users' information security. Therefore, access to users' privacy information through collusion attacks of privilege escalation has become a research hotspot.

Currently, behavior-based detection is mainly used to detect privilege escalation attacks in Android applications. Those include some features which are extracted statically, for example, permission, component, and other behavior

characteristics' detection and dynamic system operation process application behavior information collection and detection. However, two shortcomings exist in the detection of multiapplication collusion for privilege escalation attack: (1) a single application is detected, and the dataflow path between multiapplications is not detected, so the detection effect of multiapplication collusion is not good. (2) The current detection methods are based on 1–3 features, but do not consider the joint detection of more than 4 features, and without knowing the hidden dangerous factors that constitute the privilege escalation attack.

In view of the serious threat and strong concealment of collusion attacks and not good detection effect of single application, we proposed a detection method based on multifeature and process algebra modeling. The method is described as follows:

(1) Extraction feature of attack behavior: based on the analysis of the attack model, five kinds of features are extracted, including apply dangerous permission, Intent-filter, sensitive API calls, sensitive dataflow pairs, and component Intent communication.

(2) Application behavior and attack behavior modeling: process algebra was used to build the application behavior and attack behavior model. Strong equivalence was used to verification. Then, it can determine whether the application can constitute a privilege escalation attack.

(3) The dataflow path was detected between attack applications. Path detection algorithm was used to detect the dataflow path between the apps that can constitute the privilege escalation attack. So, the apps that have the path between applications constituted the privilege escalation attacks.

The main contributions of this paper are as follows:

(1) Five features of attack behavior were extracted. The static feature extraction method is used to extract dangerous permission application, sensitive dataflow pairs, sensitive API calls, component Intent communication, and Intent-filter, which make up for the limitation that a single feature can be detected but attack behavior cannot be highly restored.

(2) Behavior modeling and path detection between applications: process algebra is used to model the application behavior and attack behavior, and strong equivalence is used to confirm the equivalence relationship between the behavior model and the attack model of the app. Path detection makes up for the limitation that traditional methods only detect a single app but do not consider the collusion behavior between applications.

(3) Explanation of hidden factors of privilege escalation attacks: based on the experiments of the case, benchmarks, and test sets, the accuracy and effectiveness of the method are confirmed, and the hidden factors that constitute the privilege escalation attack are explained.

2. Related Work

The wide use of smartphones and Mobile Internet has resulted in the year-by-year increase in the number of malicious Android apps, thus making the detection of malicious apps a research hotspot. This subject has research value due to the particularity of the privilege escalation attack with multiapplication collusion. The following will be from the malware detection, and the privilege escalation detection methods are described.

2.1. Malware Detection Method. The malware detection method for Android has changed from the signature-based method to the application-based behavior feature method and then to the feature classification and detection based on machine learning and data mining theory. Androguard [14], a well-known malicious code early detection tool for Android, uses a signature-based method to detect a malicious code; however, it cannot detect unknown malicious applications.

Thus, many researchers are focusing on the detection method based on app behavior features. In [15–20], the researchers detected the behavior of Android malware by analyzing, extracting, and comparing the behavior features of the app such as permission, control flow, dataflow, and sensitive API calls. Furthermore, the probabilistic confidence value framework proposed in [20] can effectively reduce the detection cost. Although the detection method based on the application of behavioral features has achieved good results, it can still be strengthened for extracting and combining behavioral features.

Moreover, because of the maturity of machine learning and data mining theory, researchers are now introducing them to malicious application detection methods. In [21–26], the researchers used machine learning and data mining theory to extract, classify, evaluate, and detect known malicious features, and they provided a direction for further research on permission and API for detecting malicious applications. DroidCat [27] and SafeDroid v2.0 [28] contributed to query strategy, active learning, and simplifying malicious features, while DroidDeep [29] contributed to static feature collection and selection. Androdect [30] constructed the dataset of the component, key function call, and system call based on the feature extraction of the component, function call, and system call; moreover, it used the three-layer hybrid ensemble algorithm for detection. The system performed well for detection accuracy and execution efficiency but only considered three types of features. Unlike the previous studies, Amin et al. [31] proposed an anti-malware system based on a self-defined learning model that detects the end-to-end deep learning system of Android malware by extracting the operation code from the application bytecode; however, the system focused on using different deep learning models to improve the detection rate.

2.2. Privilege Escalation Detection Method. Because of the complexity of privilege escalation attacks, the aforementioned detection methods for malicious software are

relatively weak in detecting privilege escalation attacks. Some researchers have therefore conducted more in-depth research on privilege escalation attacks. In [32, 33], tracking tainted information and monitoring permission information were used to protect and detect the kernel-level privilege escalation attack. Two types of attacks on the application layer are confused deputy attacks and collusion attacks.

Researchers have proposed some good solutions for detection and prevention of confused deputy attacks. Bugiel et al. [34] proposed extending Android middleware and deploying a security framework of mandatory access control in the kernel, thus aiming to detect and prevent the application-level confused deputy attacks and focusing on the binder IPC, Internet sockets, and the file system of the kernel. Lee et al. [35] proposed protecting Android from privilege escalation attacks by monitoring important system calls of application processes. This method can detect and prevent new unknown malware; however, it only considers the feature of system calls. Xu et al. [36, 37] proposed a flexible and efficient security extension scheme for Android middleware. This security framework is used to prevent the expansion of permission lead vulnerabilities of third-party applications via confused deputy attacks. In this framework, the main consideration is how to restrict risky interapplication communications.

For the collusion attacks that we have studied, researchers have made in-depth exploration from data information and behavior features. Youn et al. [38] proposed a method for preventing privilege escalation attacks by verifying the usefulness of the protocol for requesting services, reviewing the applicability of requesting services, transmitting information to the system, and communicating risks to users. However, the primary consideration is the metadata and the context manager metadata. DroidAuditor [39] is a solution for application behavior analysis using interactive behavior graphs; it can detect application-layer privilege escalation attacks such as confused deputy and collusion attacks. Furthermore, it primarily uses the Android security module (ASM) access control architecture to analyze application behavior. Wang et al. [40] proposed a method for detecting privileged escalation attacks based on the component, application layer, and the defects of package management. For the component-based detection method, the researcher considered three features: permission, exported attribute, and Intent-filter. For the application-layer-based detection method, the component and permission information are considered. In the system update or package management-based detection method, the researcher considered three features: permission, UID sharing, and tainted data. To summarize, this study provided a good research for collusion attack detection; however, the number of features used in each method can still be strengthened.

Thus, for the application-layer collusion attacks, using the aforementioned detection method to detect a single application, the detection results must not be good. However, for the existing detection methods, only 1–3 types of behavior features are considered, and the detection of the dangerous information path between applications is insufficient. At the same time, Bhandari et al. [41] also proposed the importance of interapp communication detection. In fact, in addition to considering multiple features, it is necessary to detect the dangerous path between applications to effectively detect the multiple application collusion privilege escalation attacks in the application layer.

3. Extraction of Behavior Features

The behavior of applications can be expressed by their features [30]. Static methods are adopted to extract the behavior features on the basis of an analysis of attack-behavior features.

3.1. Analysis of Attack-Behavior Features. Android applications are composed of activity, service, broadcast receiver, and content provider, which communicate through Intent [2, 42]. The security architecture of Android allows an application with fewer (or no) permissions to access application components with more permission [2]. Therefore, malicious programs without any permission obtain the required privileges through third-party programs, thus constituting a multiapplication collusion privilege escalation attack, as shown in Figure 1 [43–45].

As can be seen from Figure 1,

(1) Three applications (A, B, and C) run independently and do not interfere with each other. Application A has components $ComA_1$ and $ComA_2$ and no permission. Application B has permission P_1 and components $ComB_1$ and $ComB_2$. Application C has no permissions, but component $ComC_1$ has permission P_1 and $ComC_2$ has permission P_2.

(2) $ComA_1$ without permission can access $ComB_1$ with P_1; $ComB_1$ can access $ComC_1$ with P_1 that is applied for component $ComC_1$ but cannot access $ComC_2$ with P_2.

(3) $ComA_1$ has P_1 without being applied for P_1. Three applications (A, B, and C) can constitute a privilege escalation attack based on P_1.

According to the attacking principle, the app that constitutes an attack must have the following behavioral features:

(1) Applied dangerous permission: privilege escalation attack occurs in the process of continuously refined permissions; dangerous permissions are an important factor of the privilege escalation attack.

(2) Component Intent communication: Android applications are component-based, and constituted attack involves transferring information between components.

(3) Sensitive dataflow pairs: privilege escalation attack can obtain user privacy data, and a hidden dangerous factor of sensitive dataflow pairs among components exists.

(4) Sensitive API calls: through tracking of sensitive API calls, we can know the dangerous behavior of the app.

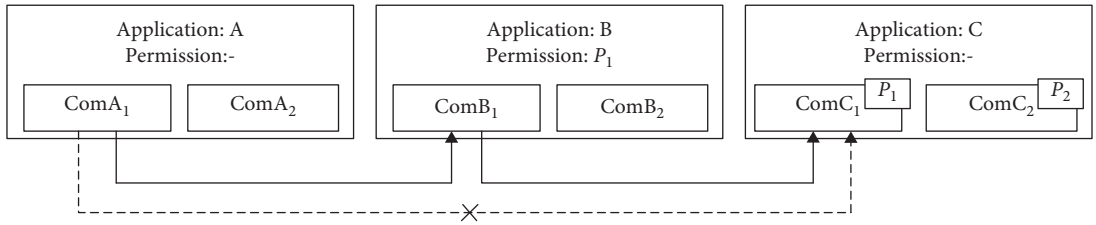

Figure 1: Principle of the privilege escalation attack.

(5) Intent-filter protection: Intent-filter can help realize the communication between the same application components and applications.

Moreover, it is necessary to detect the interapplication path for the applications that can constitute the privilege attack. The permission mechanism of the Android platform has a feature that once the permission is granted, it will be permanently protected by the permission [42], and it can realize the call of components between applications, which is the best camouflage for privilege escalation attacks.

3.2. Privilege Escalation Attack Case. A privilege escalation attack case is presented, consisting of three normally independent apps (App1, App2, and App3) based on the permission of SEND_SMS. The key code for the three applications is shown in Table 1.

The key code is as follows:

(1) The component ComA of App1 can communicate with the component ComB of App2 without permission protection.

(2) App2 has applied for permission SEND_SMS so that its component ComB has SEND_SMS permission.

(3) ComC of App3 applied for SEND_SMS permission. ComB of App2 has the same permission as ComC, so ComB communicates with ComC.

(4) The three applications in SEND_SMS are refined and enhanced step by step. The three conspired applications constitute the privilege escalation attack. They use SMS to send the bank code to the specified phone number.

Figure 2 shows a diagram of a constituted privilege escalation attack.

3.3. Application-Behavior Feature Extraction. Feature extraction was performed to construct an application feature tree (AFT). AFT is a tree that has three depths. The root node stands for the name of the application, the root node of each subtree stands for the component of the application, and the leaf node of the subtree stands for the features of the corresponding component. Features include dangerous permissions, component Intent communication, sensitive dataflow pairs, Intent-filter, and sensitive API calls.

3.3.1. Component and Dangerous Component Permission Extraction. Permission and component information were extracted from an AndroidManifest.XML file. A dangerous permissions' list was built based on Google's official documents, as shown in Figure 3.

3.3.2. Dangerous Application Permission Extraction. According to the permission mechanism of the Android system, each component is protected by an application's permissions, as shown in Figure 4.

Several random APKs were used to extract permissions and components. The statistical data are shown in Table 2.

3.3.3. Component Intent Communication Extraction. Based on the conversion of the APK to the Smali file, it is combined with the Intent tag of the component in the AndroidManifest.XML for analysis, and then the information of application components was extracted, as shown in Figure 5.

In the attack case in Section 3.2, ComB of App2 was extracted, and its Intent communication information is shown in Table 3.

3.3.4. Sensitive API Call Extraction. Au et al. [46] studied the corresponding relationship between the permissions of multiple versions of the Android system and API correspondence. With the help of their findings, sensitive API calls are obtained by using the strace tool to extract the sequence of system calls, as shown in Figure 6.

3.3.5. Sensitive Dataflow Pair Extraction. FlowDroid [47] was used to extract sensitive data stream pairs, that is, <source, sink>. Source is the source of sensitive information acquisition, and sink is the source of sensitive information transmission. The extracted sensitive dataflow pairs are added to AFT.

In the attack case in Section 3.2, the component ComC of App3 was extracted, and its sensitive dataflow pairs are shown in Table 4.

3.3.6. Intent-filter Extraction. <Intent-filter> protection and Intent's action name are used to track the transmission of information flow, as shown in Figure 7.

4. Application-Behavior Modeling Based on Process Algebra

4.1. Syntax and Semantic of Process Algebra. Process algebra can be applied to accurately describe the interaction between

Table 1: Key code.

	App1	App2	App3
Applied for dangerous permission	—	App:<uses-permission android: name = “android.permission.SEND_SMS”/>	ComC:<android: permission = “android.permission.SEND_SMS>
Intent-filter	—	ComB:<action android:name = “IntentKeyOne”/>	ComC:<action android: name = “IntentKeyTwo”/>
Component Intent commutation	ComA:Intent intentA = new Intent(“IntentKeyOne”); bundleA.putString(“userBankCode”, txtBankCode.getText().toString());	ComB:Intent intentB = new Intent(“IntentKeyTwo”);	—
Sensitive API calls	—	—	void enforceReceiveAndSend(Java.lang.String)
Sensitive dataflow pairs	<TextView android:id = ”@+id/ txtBankCode”. . .. . ./>	Obtain: strBankCode = budle.getString(“userBankCode”); Sending:bundleB.putString(“strBankCodeKey”, strBankCode);	Obtain:sendBankCode = bundleC.getString(“strBankCodeKey”); Sending by SMS: smsManager.sendTextMessage(phoneNo, null, sendBankCode, null, null);

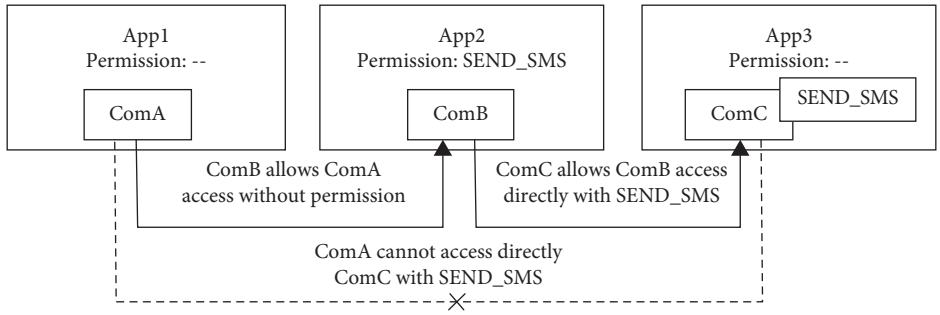

Figure 2: Diagram of a constituted privilege escalation attack.

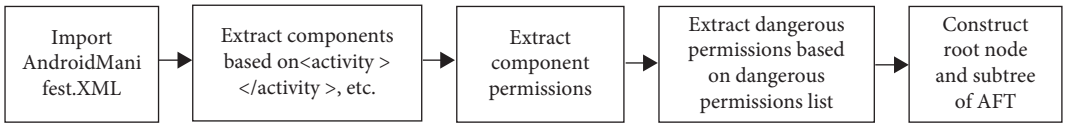

Figure 3: Process of component and dangerous component permission extraction.

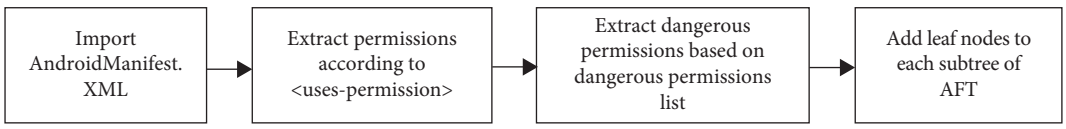

Figure 4: Process of dangerous application permission extraction.

Table 2: Statistical data of permission and component extraction.

App package name	Permission/dangerous	Repeat permission	Activity	Service	Receiver	Provider	No use component
jjszjgsgl.kaoshi.namespace	7/2	1	41	0	0	0	1
Protect.eye	24/4	0	50	6	5	2	12
com.example.healthmonitor	76/16	36	118	6	5	0	11
com.mtscrm.pahd2	9/4	0	2	1	0	0	0
...	...	...	...	...	...	...	...

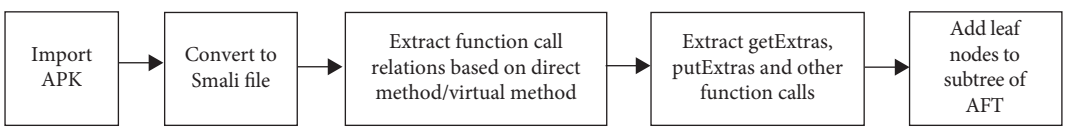

Figure 5: Process of component communication extraction.

Table 3: Example of component communication extraction.

Application's name	Component's name	Permission	Receiving	Sending
App2	ComB	Android.permission.SEND_SMS	getIntent().getExtras	putString()/putExtra

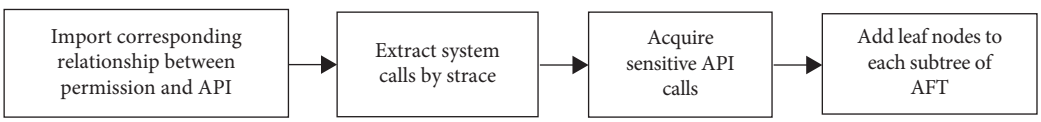

Figure 6: Sensitive API call extraction.

Table 4: Example of sensitive dataflow pair extraction.

Application	Component	Source	Sink	(source, sink)
ExampleC	ComC	Bundle.getString()/ sendActivity.onCreate()	Log/ sendActivity. sendSMSMessage	(Bundle.getString()/sendActivity.onCreate(),Log/ sendMessage.sendSMSMessage)

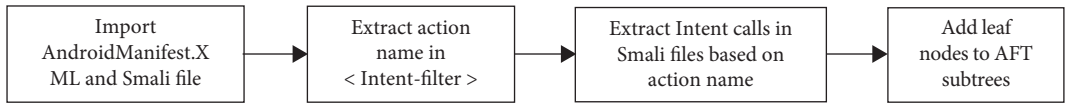

Figure 7: Intent-filter extraction.

two systems and determine the equivalence in their behavior [48]. In process algebra, the behavior of a system is defined using actions or events that it can perform. Actions represent abstract activity or behavior. Therefore, process algebra is used to define interactions between processes based on actions. It describes the behavior of a system in a combinatorial way. The Android application is built on components. The action set of all the components constitutes the overall behavior of the app [49]. Process algebra can be used to effectively describe the Android architecture and features of message communication. Therefore, we propose that process algebra must be used to model the behavior and attack behavior of the app. The syntax and semantic specifications of the process algebra are given as follows:

$$P::=A(y_1,y_2,\ldots,y_n)\Big|\sum_{i\varepsilon I}a_i.P_i\Big|P_1\Big|P_2 \cdot\Big|\left(\overline{X}\langle y_1,y_2,\ldots,y_n\rangle \mid X(y_1,y_2,\ldots,y_n)\right)\Big|(\nu\chi)P|!P|0. \tag{1}$$

In the formula,

(1) $A(y_1,y_2,\ldots,y_n)$, represents that each process P has a unique process identifier, where y_i represents the free name in P.

(2) $\sum_{i\in I}a_i.P_i = a_1P_1 + a_2P_2 + \cdots + a_nP_n$; it is a summation, where I is any finite indexing set. If $I = \varnothing$, then $\sum_{i\in I}a_i.P_i$ is the empty summation, written as 0, indicating that the process is successfully terminated. P_i is protected by a_i because P_i must start activities after the action represented by a_i occurs.

(3) $P1|P2$ represents P_1 and P_2 running concurrently.

(4) $(\overline{X}\langle y_1,y_2,\ldots,y_n\rangle \mid X(y_1,y_2,\ldots,y_n))$, where $\overline{X}\langle y_1,y_2,\ldots,y_n\rangle$ represents the action of message sending; $X(y_1,y_2,\ldots,y_n)$ represents the action of message receiving.

(5) $(\nu\chi)P$, the scope of χ is limited to the subsequent process P, where υ is a qualified identifier.

(6) $!P$ represents the process replication, which defines a recursive process.

(7) 0 represents an empty process.

4.2. Application-Behavior Model. We used process algebra to create an application-behavior model based on AFT. The syntax and semantic specifications of the definitions are given as follows.

Definition 1 (application behavior). In the Android architecture, applications are composed of components. On the basis of the permission mechanism of Android, component action is a set of information sending, receiving, and executing operations under permission protection. The sensitive behavior of a component is under permission protection. Therefore, app behavior can be obtained by the actions of all components. Application behavior is defined as follows:

$$\text{AppBehavior}::=\sum_{i=1}^{n}\text{Com}_i\sum_{j=1}^{m}\text{Action}_j,\quad (0<i\le n, 0<j\le m), \tag{2}$$

where n represents the total number of components in the app and m represents the total number of actions of the current component. The behavior model of any component is as follows.

The application behavior model is created by using the process algebra based on AFT. Each composition part of formula (1) is expressed using the key features of the Android architecture. For example, $y_1,y_2,\ldots,y_n$ in $\overline{X}\langle y_1,y_2,\ldots,y_n\rangle$ is expressed using sensitive data. Therefore, the behavior model of any component is constructed by using formulas (1) and (2) as follows:

$$\forall\text{ComAction}::=\text{Com(id)}\Big|\sum_{\substack{i\varepsilon v\\ j\varepsilon w}}P_i.\text{Feature}_j\Big|\text{Feature}_1\Big| \cdot\text{Feature}_2\Big|\cdots|\text{Feature}_W\Big|(\overline{X}\langle\text{data}\rangle \mid \cdot X(\text{data}))|(\text{Feature})P|!\text{Feature}_j. \tag{3}$$

In the formula,

(1) Com (id) represents that ComAction has a unique id, and it can be omitted.

(2) $\sum_{i\varepsilon v}^{j\varepsilon w} P_i.\text{Feature}_j$, where, w represents the number of features in AFT, v represents the number of permissions, and P represents permissions. $P_i.\text{Feature}_j$ represents Feature_j is protected by P_i and the order in which they occur; Feature_j must occur before P_i can start the activity.

(3) $\text{Feature}_1|\text{Feature}_2|\text{Feature}_w$ represents an application has w features at the same time.

(4) $(\overline{X}\langle\text{data}\rangle \mid X(\text{data}))$, where $\overline{X}\langle\text{data}\rangle$ represents the action of message sending, and $X(\text{data})$ represents the action of message receiving.

(5) $(\text{Feature})P$ represents the behavior in which an application is protected by permission P.

(6) $!\text{Feature}_j$ represents Feature_j being replicated.

Formula (3) describes the set of behaviors of any component of the app. This includes the following: any single component can have a unique identity, actions of the components can be performed under the permission protection, and these actions must be performed after the permission is granted; each component can have multiple features, the components can have the ability to send and receive information, and any feature of the component can be reused.

Definition 2 (privilege escalation attack model)**.** According to the analysis of the attacking principle in Section 3.1, the application that constitutes the privilege escalation attack (PEApp) must have dangerous permissions, sensitive API calls, component Intent communication, and sensitive information flow sending. According to formula (2), attack behavior must be completed by a component in the application, that is, $\exists\text{Com}_k\,(k \in n)$, and the component must have AFT. According to formulas (1) and (3), the model of privilege escalation attacks is as follows:

$$\begin{aligned}\text{PEApp}::&= \exists\text{Com}_k \sum_{j=1}^{m} \text{feature}_j \\ &::= \sum_{j=1}^{m} P_1.\text{feature}_j|(\overline{X}\langle y\rangle \mid X(y))|_1(\text{Feature})P| \\ &\quad !\text{Feature}_j,\end{aligned} \tag{4}$$

where

(1) $\sum_{j=1}^{m} P_1.\,\text{feature}_j$ represents all actions of Com_k

(2) P_1 represents dangerous permissions of the current component

(3) y represents sensitive information of this component

Formula (4) is used to describe the principle that a component of an app has a set of attack behaviors. Actions of the component can be performed under a series of dangerous permission protections. These actions must be performed after permission is granted, after which the component can send and receive information, where any feature of the component can be reused. Therefore, by using formulas (3) and (4), we can complete the modeling of app and attack behaviors.

5. Behavior Equivalence and Interapplication Path Detection

The strong equivalence of process algebra is used to determine the equivalence relationship between the behavior and attack models of the app. Therefore, according to the concept of labelled transition system, strong simulation, and strong equivalence in process algebra, the following are defined: behavior-labelled transition system, behavior-strong simulation, and behavior-strong equivalence based on AFT.

5.1. Related Concepts of Behavior Equivalence

Definition 3 (behavior-labelled transition system (LTS))**.** Suppose the application action is under the protection of P permission; the app action set $\text{Act} = \{a_1, a_2, \ldots, a_t, a_1', a_2', \ldots, a_t'\}$ is a pair (Q, T) which is LTS, where $Q = \{(a_1, P), (a_2, P), \ldots, (a_t, P), (a_1', P), (a_2', P), \ldots, (a_t', P)\}$ is a state set; $T \subseteq (Q \times \text{Act} \times Q)$ is a ternary relation known as a transition relation. If $\forall(a_i, P) \in Q$, $\forall(a_i', P) \in Q, \exists a_i \in \text{Act}\,(\text{or}\,\exists a_i' \in \text{Act})$, and $((a_i, P), a_i, (a_i', P)) \in T\,(\text{or}\,((a_i, P), a_i', (a_i', P)) \in T)$, we write $(a_i, P) \xrightarrow{a_i} (a_i', P)\,(\text{or}\,(a_i, P) \xrightarrow{a_i'} (a_i', P))$. Therefore, LTS for app component behavior and attack behavior can be constructed to determine the equivalent relationship between component behavior and attack behavior.

Based on formulas (3) and (4), the attack behavior of an app's component must be included in the overall behavior of the component; therefore, the simulation relationship between the attack behavior of the component and the behavior of the component belongs to the same LTS. Therefore, based on the concept of strong simulation in process algebra, the concept of behavior-strong simulation is given.

Definition 4 (behavior-strong simulation)**.** Based on Definition 3, let (Q, T) be an LTS, and let $S = \{((a_1, P), (a_1', P)), ((a_2, P), (a_2', P)), \ldots, ((a_t, P), (a_t', P))\}$ be a binary relation over Q. Then, S is called a strong simulation over (Q, T) if whenever $(a_i, P)S(a_i', P)$; if $\forall(a_i, P), (a_i, P) \longrightarrow^{a_i} (a_i, P)'$, then there exists $(a_i', P)'$ such that $(a_i', P) \longrightarrow^{a_i} (a_i', P)'$ and $(a_i, P)'S(a_i', P)'$. It can verify whether the LTS consisting attack behavior and component behavior has strong simulation S. S includes attack behavior and part of the component behavior.

Definition 5 (behavior-strong equivalence)**.** Based on Definitions 3 and 4, a binary relation S over Q is said to be a strong bisimulation over the LTS (Q, T) if both S and its converse are simulations. We say that (a_i, P) and (a_i', P) are strongly equivalent, written as $(a_i, P) \sim (a_i', P)$, if a strong bisimulation S exists such that $(a_i, P)S(a_i', P)$, where $(a_i, P) \in \text{ComAction}$, $(a_i', P) \in \text{PEApp}$. Definition 5 is used to verify that the strong simulation S with attack

behavior and part of component behavior is strong mutual simulation. There are two states, namely, $(a_i, P) \in$ ComAction, $(a_i', P) \in$ PEApp and $((a_1, P), (a_1', P)) \in S$, which are strong equivalences. Therefore, the definition can verify the equivalence between the application component behavior and attack behavior models.

5.2. Interapplication Path Detection

Definition 6 (dangerous information flow path between applications (DIPA)). Since the application-layer collusion privilege escalation attack occurs between multiple applications, it is necessary to detect the dangerous data flowing between multiple applications. Therefore, on the basis of applying for dangerous permissions, sensitive information transmission paths exist between multiple apps, which are composed of "applicationName.componentName⟶ applicationName.componentName⟶⋯⟶application Name.componentName."

Based on Definition 6, we construct an algorithm of the interapplication dangerous information flow path.

6. Experiment

6.1. Application Feature Extraction. According to the discussion in Section 3.3, the features of App1, App2, and App3 are obtained as shown in Table 5. Text in italics indicates the abbreviations of features, and text in boldface denotes the storage of feature (Intent) value.

According to Table 5, AFT is constituted for ComA, ComB, and ComC, as shown in Figures 8(a)–8(c), respectively.

6.2. Attack Behavior Model. For the AFT of the privilege escalation attack case given in Section 6.1, the application's attack behavior models were modeled by formula (4).

(1) App1 model is shown in the following formula:

$$\text{App1}:: = \left(\overline{X}\langle y_1\rangle \mid X(y_1)\right). \tag{5}$$

(2) App2 model is shown in the following formula:

$$\begin{aligned}\text{App2}:: &= \sum_{j=1}^{m} P_1.\text{feature}_j\left(\overline{X}\langle y_1, y_2\rangle \mid X(y_1, y_2)\right)\Big|(\nu\chi)P_1 \\ &:: = P_1.\text{feature}_1 + P_1.\text{feature}_2 \\ &\quad + P_1.\text{feature}_3\big|(x(y_1) \mid \overline{x}\langle y_1\rangle).P_1.\end{aligned} \tag{6}$$

(3) App3 model is shown in the following formula:

$$\begin{aligned}\text{App3}:: &= \sum_{j=1}^{m} P_1.\text{feature}_j\left(\overline{X}\langle y_1, y_2\rangle \mid X(y_1, y_2)\right)\Big|(\nu\chi)P_1 \\ &:: = P_1.\text{feature}_1 + P_1.\text{feature}_2 + P_1.\text{feature}_3 \\ &\quad + P_1.\text{feature}_4\big|\left(x(y_1) \mid \overline{x}\langle y_1\rangle\right).P_1.\end{aligned} \tag{7}$$

6.3. App Behavior Model. Formula (3) is used to model the app behavior of the case as follows:

(1) The behavior model of AppModel1 is shown in the following formula:

$$\text{AppModel1}:: = (X(\text{data1}) \mid \overline{X}\langle\text{data1}\rangle). \tag{8}$$

(2) The behavior model of AppModel2 is shown in the following formula:

$$\begin{aligned}\text{AppMode2}:: = {} & P_1.\text{Intent1} + P_1.\text{APIApp2} \\ & + P_1.\text{filterApp2}\big|(X(\text{data1}) \mid \overline{X}\langle\text{data2}\rangle).P_1.\end{aligned} \tag{9}$$

(3) The behavior model of AppModel3 is shown in the following formula:

$$\begin{aligned}\text{AppMode3}:: = {} & P_1.\text{Intent1} + P_1.\text{APIApp3} + P_1.\text{filterApp3} \\ & + P_1.\text{SFPApp3}\big|(X(\text{data2}) \mid \overline{X}\langle\text{data3}\rangle).P_1.\end{aligned} \tag{10}$$

6.4. Equivalence Relation Verification and Path Detection. MWB (Mobility Workbench) is a tool used for manipulating, analyzing, and verifying mobile concurrent systems described in process algebra. According to Definition 5, MWB is used to normalize the app behavior model and the attack model. Then, the equivalence relationship between them is verified. Table 6 is the basic syntax transformation of normalization.

For example, $(\overline{X}\langle y_1\rangle \mid X(y_1))$ in formula (5) is converted according to MWB grammar, and the conversion result is $X(y_1).'X\langle y_1\rangle$.

Formulas (5)–(10) were transformed into MWB language. The attack behavior models of App1 and App2 (representation in MWB with APP2), and the app behavior models of AppModel1 and AppModel2 were validated to have a strong equivalence relationship, as shown in Figure 9.

Similarly, we can verify the strong equivalence between App3 and AppModel3. According to Algorithm 1, we can see that

(1) App1, App2, and App3 can constitute a privilege escalation attack.

(2) The attacking path of the constituted privilege escalation attack is

$$\begin{aligned}\text{App1.ComA.IntentApp1} &\longrightarrow \text{App2.ComB.IntentApp2} \\ &\longrightarrow \text{App3.Com3. Intent3.}\end{aligned} \tag{11}$$

7. Evaluation and Validity Analysis

7.1. Experiment Evaluation. The key steps of our detection method are sensitive dataflow detection and dangerous path detection. In the worst case, the search time complexity of the algorithm is $O((n-m+1)m)$. Fifty-five Android APK

Table 5: Results of behavior feature extraction.

Application	App1	App2	App3
Application's component	ComA	ComB	ComC
Dangerous permission	—	SEND_SMS	SEND_SMS
Component Intent communication	*Intent1*: putString("data1")/ putExtras() new Intent("IntentApp1")	*Intent2*:getInent().getExtras/ getString("data1") putString("data2")/ putExtras()new Intent("**IntentApp2**")	*Intent3*:getInent().getExtras() getString("data2")/getExtras()
Sensitive API call	—	*APIApp2*:void enforceReceiveAndSend(Java.lang.String)	*APIApp3*:void enforceReceiveAndSend(Java.lang.String)
Sensitive dataflow pairs	—	—	*SFPApp3*:(Bundle.getString()/ sendActivity.onCreate(),Log/ sendMessage.sendSMSMessage)
Intent-filter	—	*FilterApp2*: Intent.ActionName = **IntentApp1**;	*FilterApp3*: Intent.ActionName = "**IntentApp2**";

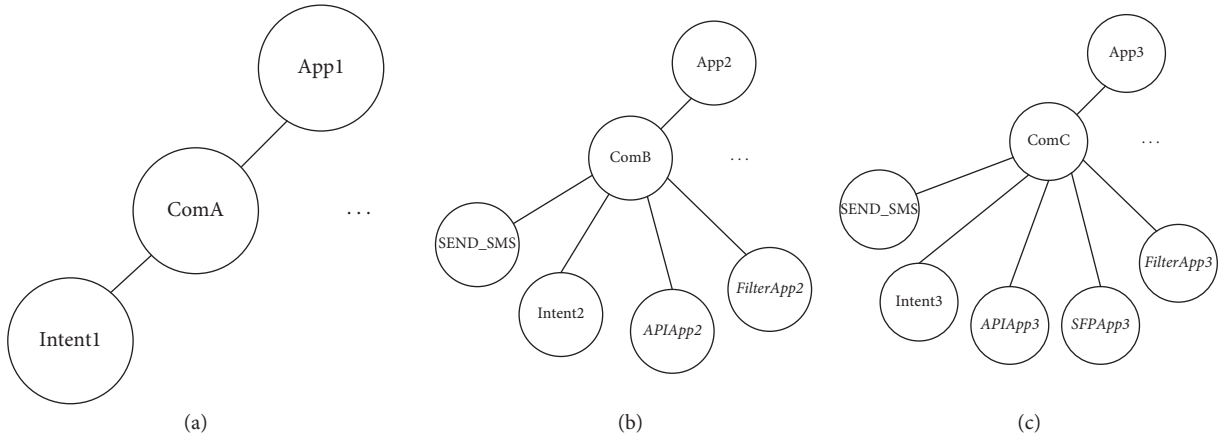

Figure 8: Constituting AFT: (a) AFT of App1; (b) AFT of App2; (c) AFT of App3.

Table 6: Process algebra and MWB syntax conversion rules.

MWB	Process algebra	Syntax
‘α	$\overline{\alpha}$	input action
agent	—	define identifier
Eq	~	strong equivalence
0	0	an empty process
T	T	internal action
.....	.....	.....

samples were tested, and two APKs failed. The time and space cost analysis data are shown in Table 7.

7.2. Benchmark Test and Comparison Analysis. This method was used to examine some test sets of the DroidBench benchmark [50] and comparisons with [40]. The detection method that is proposed in [40] is used to detect the components, dangerous permissions, component communication, sensitive API calls, and the Intent-filter in the benchmark test. However, this method does not detect the flow of sensitive data. Table 8 summarizes the obtained data of the benchmark test and comparison results. The expression was as follows: the test result of "this method/the test result of [40]." The test results show that a threat was expressed by *T*, no threat was expressed by NT, a false alarm was expressed by F, a missed alarm was expressed by *L*, a security call (registration) was expressed by A, insecurity was expressed by NA, and nondetection was expressed by N, as shown in Table 8.

The benchmark test showed that this method can accurately detect the security risks of the privilege escalation attack such as sensitive information and component communication. Compared with [40], two points were shown:

(a) (b)

Figure 9: Validation of the strong equivalence relationship: (a) validation of App1's strong equivalence relationship; (b) validation of App2's strong equivalence relationship.

Input: AppBechaver$_i$, AFT$_i$, PEApp$_i$, P_1
Output:DIPA
(1) App$_1$ = getStartPoint(AppBechaver)
(2) For $i = 0$ to $m - 1$
(3) Construction PEApp$_i$
(4) If App$_i$ ~ PEApp$_i$ Then
(5) Insert into PEAppString+"/"//Use/to split application which constituted privilege escalation attack
(6) temp$_j$ = Split(PEAppString,"/")
(7) for $j = 0$ to w//w represents the number of applications that can constituted privilege escalation attack
(8) Construction AFT$_j$
(9) for $j = 0$ to w
(10) Construction AFF//Construct Application Feature Forest (AFF) of suspicious application based on AFT
(11) Intent$_k$ points to leaf-AFF$_k$
(12) postorder-traversal save as forestString
(13) n = forestString.length, m = Intent$_k$.length
(14) for $s = 0$ to $n - m$
(15) if Intent$_k$[1...m] = = forestString ($s + 1$,$s + m$) And P$_1$Intent$_k$ = = forestString ($s - 1$, s)
(16) save rootNode.Com.Intent$_k$ to resultArray
(17) forestString[$s - 2$] as startPoint goto (12) until $s = = n - 1$
(18) print resultArray as DIPA

Algorithm 1: Interapplication dangerous information flow path algorithm.

Table 7: Time cost and space cost.

	Minimum	Maximum	Average
Time cost (S)	94.947	380.497	131.51
Space cost (M)	857.118	917.932	888.35

① We proposed the method detected sensitive API calls and sensitive dataflow pairs and considered more comprehensive dangerous factors than [40]
② Two methods have high accuracy in components, dangerous permissions, component communication, and Intent-filter

Table 9 shows the accuracy rate, false positive rate, and missing rate of our method in benchmark test.

7.3. Experiment Validity

7.3.1. Composition of the Test Set. Fifty-two APKs were selected from an Android application market such as Google Play, and three APKs were developed by our research team to build the test sample set [51]. According to the classification of APK in major application markets, 22 classes of APK were selected in sample extraction, and 1 to 3 typical APKs were selected for each type. The specific

Table 8: Data of benchmark test and comparison results.

Test sets	Components	Dangerous permissions	Component communication	Sensitive API call	Sensitive dataflow pairs	Intent-filter
ActivityCommunication1	T/T	T/T	NT/NT	T/N	NT/N	A/A
ActivityCommunication2	T/T	T/T	T/T	T/N	NT/N	A/A
FieldSensitivity1	T/T	T/T	NT/NT	L/N	NT/N	A/A
FieldSensitivity2	T/T	T/T	NT/NT	T/N	NT/N	A/A
InactiveActivity	T/T	T/T	NT/NT	F/N	NT/N	A/A
LogNoLeak	T/T	NT/NT	NT/NT	NT/N	NT/N	A/A
ObjectSensitivity1	T/T	T/T	NT/NT	T/N	NT/N	A/A
ObjectSensitivity2	T/T	T/T	NT/NT	T/N	NT/N	A/A
PrivateDataLeak1	T/T	T/T	NT/NT	T/N	L/N	A/A
PrivateDataLeak2	T/T	T/T	NT/NT	T/N	L/N	A/A
PrivateDataLeak3	T/T	T/T	NT/NT	T/N	L/N	A/A
ServiceCommunication1	T/T	T/T	L/NT	NT/N	NT/N	A/A
StartActivityForResult1	T/T	T/T	T/NT	T/N	NT/N	A/A

classification and the number of APKs are shown in Table 10.

For Android users, they want the app to be powerful but smaller; therefore, two rules should be followed when selecting a typical APK: (1) app is at the top of this kind of app; (2) the app takes up less space. However, the small size of an app does not mean that it has fewer components, which is the smallest unit for launching an attack. Therefore, the statistical figure of the app size and the number of components is shown in Figure 10.

Because collusion attacks easily occur in the combination form of multiple apps of the same company and the same developer, the test set selection considers the situation of the same company or the same developer. Furthermore, there are 13 apps developed by four companies and one developer; the detail is shown in Table 11.

7.3.2. Analysis of Experiment Validity. The validity of this method was verified by the test sample set. The test results showed that 5.5% of the applications constitute the privilege escalation attack. Some applications are equivalent to the attack model, which can be used as part of the collusion attack. No dataflow path with multiple applications is detected; thus, it is called the application that can constitute the privilege escalation attack, accounting for 32.7%. However, APK encryption or code obfuscation hinders the detection of our method. Thus, 3.6% of the APKs were failed, and 96.4% is the efficiency. The component-based detection method that is proposed in [40] is used to detect the test set. Both test set detection results and comparison of the test set are shown in Table 12. The expression method is the test results of this method/the test results of [40].

From Table 12, it can be observed that the component-based detection method proposed in [40] lacks the detection of dataflow between the applications. The detection result of collusion constitutes that a privilege escalation attack is invalid. The method of Wang et al. [40] does not distinguish between the apps that can constitute the privilege escalation attack and the ones that have hidden dangerous. Therefore, 54.6% of the apps can be detected as privilege escalation attacks, but using our method, there are 71.0% of them which can be detected as privilege escalation attacks and hidden dangerous. Since Wang et al.'s [40] method only detects activities and service components, there is a 16.4% missing rate. Compared with the component-based detection method proposed in [40], the test results in Tables 8 and 12 indicate that the method proposed in this paper has the following advantages: (1) it detects sensitive dataflow pairs, making the detection of attack behavior features more effective; (2) it detects the dataflow path between applications, making the method more conducive to detect collusion attacks; (3) it can distinguish between an app that can constitute a privilege escalation attack and an app that is hidden dangerous, which makes the detection results more accurate; and (4) it detects the four components that lead to reduction in the missing rate.

As shown in Figure 10, components are the basic units of detection, while 1,679 components are detected in this study.

Among them, 272 components have Intent communication. The components of the app use Intent to send sensitive data; however, no components receiving sensitive information can be detected in the app. In this case, there may be a vulnerability in passing sensitive information or encoding, which is termed a dangerous component, represented by a total number of 88 components. In the encoding process, some components are registered in the AndroidManifest.XML file, but in the app, these components have never been used. It is called a registered but unused component, with 53 components in total. The detailed statistical results are shown in Table 13.

For example, in the APK with the package name "com.example.healthmonitor," it is detected that there are 12 components registered, such as ".Uploadimage,.Ecgdactivity,.Ecgdatactivity, and.Shakeactivity," but they are not used. In the APK with the package name of "com.yuntongxun.ecdemo1," there is a dangerous component ".Main.index." In this component, the sending action with Intent name of "SendTo" is detected, but the receiving component of the Intent data cannot be found in the APK.

As per Table 11, there are 13 apps of the same company or developer in the test set. Among them, there are three apps that constitute privilege escalation attacks, two apps

Table 9: Analysis of our method in benchmark test results.

	Component (%)	Dangerous permission (%)	Component Intent communication (%)	Sensitive API calls (%)	Sensitive dataflow pairs (%)	Intent-filter (%)
Accuracy rate	100	100	92.3	84.6	76.9	100
False positive rate	0	0	0	7.7	0	0
Missing rate	0	0	7.7	7.7	23.1	0

Table 10: APK sample classification.

Type	Working	Daily life	Shopping	Home control	Medical treatment	Finance	Examination	Browser	Tourism	Beauty	Social networks
Number	3	3	3	2	3	3	3	1	2	1	3
Type	Picture browsing	Reading	System tools	News	Home-based elderly care	Study	Exercise	Wallpaper	Plug-in unit	Entertainment	Research group
Number	3	1	3	3	3	3	3	1	3	2	3

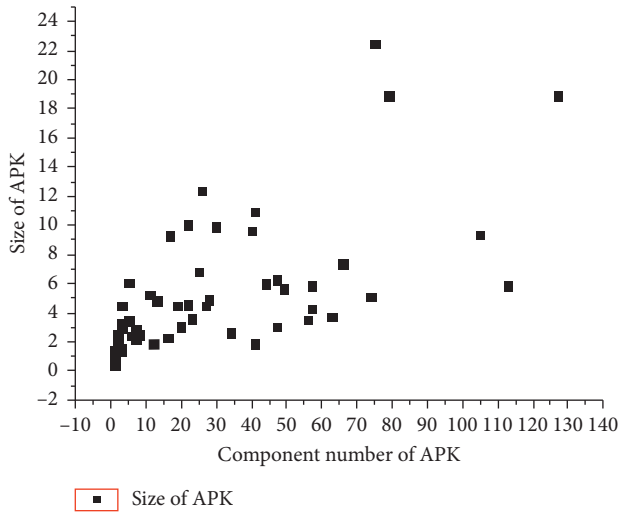

Figure 10: App size and the number of components.

Table 11: Same company or same developer rate.

	Same company or same developer	Different company or different developer
Number	13	42
Percentage (%)	23.6	76.4

Table 12: Test set detection results and comparison.

Category	Number	Percentage (%)
Constitute attack	18/30	32.7/54.6
Hidden dangerous app	18/0	32.7/0
Privilege escalation attack	3/0	5.5/0
Nondangerous app	14/23	25.5/41.8
Method failed	2/2	3.6/3.6
Method efficiency	53/53	96.3/96.3

Table 13: Component communication threat statistics.

	Number	Percentage (%)
Component has Intent communication	272	16.2
Drogues component	88	5.2
Components were registered but not used	53	3.2
Nondangerous component	1,266	75.4

Table 14: Statistics of the app by the same company or developer.

	Number	Percentage (%)	Percentage of the same samples (%)
App constitutes privilege escalation attacks	3	23.1	100
App may constitute privilege escalation attacks	2	15.4	11.1
App has hidden dangers	5	38.4	27.8
Safe app	3	23.1	21.4

Table 15: Dangerous factors.

Factors	Number	Percentage (%)
Repeatedly applied for permission	20	36.4
Applied for nonexistent permission	10	18.2
Applied for dangerous permission	38	69.1
Registered component but not used	11	20
Dangerous components	29	52.7

that can constitute privilege escalation attacks, and five apps that have hidden dangers, as shown in Table 14.

The statistics in Table 14 show that the percentage of the applications of the same developer that can constitute the privilege escalation attacks and carry security risks is relatively high. This is because

(1) The same developer's ideas and methods are the same, which makes it easy to produce the same software vulnerabilities and coding irregularities

(2) The same developers have the convenience of using the application developed by them to constitute privilege escalation attacks

Therefore, the joint detection of multiapp in the same development is particularly important.

Our test results show that the dangerous factors in the app that can constitute the privilege escalation attack included dangerous permission abuse and inaccurate use of component Intent methods, as shown in Table 15.

For example, in the APK with the package name "com.example.healthmonitor," 36 components that repeatedly applied for permissions were detected. Among them, "android.permission.WRITE_EXTERNAL_STORAGE" was applied for a dangerous permission six times, "android.permission.READ_PHONE_STATE" five times, and "android.permission.ACCESS_FINE_LOCATION" four times, with more examples occurring but not listed herein.

8. Conclusion

On the basis of feature extraction of applications, our method proposes that application behavior and attack behavior can be modeled through process algebra and combined with a strong simulation to detect a single application and then uses interapplication dangerous information flow path algorithm to detect the interapplication calls so as to determine the multiapps that constituted the privilege escalation attacks. In our method, we fully consider the different roles of multifeatures in the application-layer collusion attacks. The proposed approach was tested by using the DroidBench benchmark; the results show its good accuracy. Through the detection of the test set, 32.7% of the APK can constitute privilege escalation attacks, 32.7% of the APK have security risks, and the efficiency of the method is 96.4%. Based on the test results of this study, the improper use of permission, intent communication, and the intentional design of the same developer are the biggest factors that cause the collusion attack of the application layer. In the future work, we will continue to study the privilege escalation attack model and dynamic feature extraction technology to refine attack behavior features and improve the attack model.

Data Availability

The measurement data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare no conflicts of interest.

Acknowledgments

This work was partly financially supported through grants from the National Natural Science Foundation of China (no.

61772450), the Natural Science Foundation of Hebei Province (no. F2017203307), and Science and Technology Project of Hebei Province (no. 17210701D).

References

[1] L. Khalid, L. Mohamed, and E. Khalid, "Porting mobile apps from iOS to Android: a practical experience," *Mobile Information Systems*, vol. 2019, Article ID 4324871, 29 pages, 2019.

[2] Y. P. Xu, Z. F. Ma, Z. H. Wang et al., "Survey of security for Android smart terminal," *Journal on Communications*, vol. 37, pp. 169–184, 2016.

[3] Y. Q. Zhang, K. Wang, H. Yang et al., "Survey of Android OS security," *Journal of Computer Research and Development*, vol. 37, pp. 1385–1396, 2014.

[4] L. Sui, "Strategy analytics: android captures record 88 percent share of global smartphone shipments in Q3 2016" 2019, https://www.strategyanalytics.com/strategy-analytics/news/strategy-analytics-press-releases/strategy-analytics-press-release/2016/11/02/strategy-analytics-android-captures-record-88-percent-share-of-global-smartphone-shipments-in-q3-2016?slid=838714&spg=1.

[5] Y. Xu, J. Ren, G. Wang, C. Zhang, J. Yang, and Y. Zhang, "A blockchain-based nonrepudiation network computing service scheme for industrial IoT," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 6, pp. 3632–3641, 2019.

[6] J. S. Shi and R. Li, "Survey of blockchain access control in Internet of things," *Journal of Software*, vol. 30, no. 6, pp. 1632–1648, 2019.

[7] Y. Xu, J. Ren, Y. Zhang, C. Zhang, B. Shen, and Y. Zhang, "Blockchain empowered arbitrable data auditing scheme for network storage as a service," *IEEE Transactions on Services Computing*, vol. 13, no. 2.

[8] X. L. Wang, X. Z. Jiang, and Y. Li, "Model for data access control and sharing based on blockchain," *Journal of Software*, vol. 30, no. 6, pp. 1661–1669, 2019.

[9] Y. Xu, Q. Zeng, G. Wang, C. Zhang, J. Ren, and Y. Zhang, "An efficient privacy-enhanced attribute-based access control mechanism," *Concurrency and Computation Practice & Experience*, vol. 32, no. 5.

[10] Nokia Threat Intelligence Lab, "Nokia threat intelligence report," 2019, http://networks.nokia.com/solutions/threat-intelligence.

[11] 360 Internet Security Center, "Android system security ecoenvironment research report in 2018," 2019, http://zt.360.cn/1101061855.php?dtid=1101061451&did=610082749.

[12] 360 Internet Security Center, "China mobile phone security ecology research report in 2018," 2019, http://zt.360.cn/1101061855.php?dtid=1101061451&did=610107232.

[13] 360 Internet Security Center, "Android malware special report in 2018," 2019, http://zt.360.cn/1101061855.php?dtid=1101061451&did=610100815.

[14] "Androguard ," 2019, https://github.com/androguard/androguard.

[15] X. Su, M. Chuah, and G. Tan, "Smartphone dual defense protection framework: detecting malicious applications in Android markets," in *Proceedings of the 8th Annual International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pp. 14–16, Chengdu, China, December 2012.

[16] S. Lee and D. Y. Ju, "Assessment of malicious applications using permissions and enhanced user interfaces on Android," in *Proceedings of the 11th IEEE International Conference on Intelligence and Security Informatics (IEEE ISI)*, June 2013.

[17] P. Zegzhda, D. Zegzhda, E. Pavlenko et al., "Detecting Android application malicious behaviors based on the analysis of control flows and data flows," in *Proceedings of the 10th International Conference on Security of Information and Networks (SIN)*, October 2017.

[18] J. X. Xiao, L. Z. C. Cong, and Q. H. Xu, "A new Android malicious application detection method using feature importance score," in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence CSAI'18*, Shenzhen, China, December 2018.

[19] X. J. Liu, X. F. Dong, and Q. Lei, "Android malware detection based on multi-features," in *Proceedings of the 8th International Conference on Communication and Network Security ICCNS 2018*, Qingdao, China, November 2018.

[20] T. G. Kim, B. J. Kang, and E. G. Im, "Runtime detection framework for android malware," *Mobile Information Systems*, vol. 2018, Article ID 8094314, 15 pages, 2018.

[21] Q. Jerome, K. Allix, R. State et al., "Using opcode-sequences to detect malicious android applications," in *Proceedings of the IEEE International Conference on Communications (ICC)*, pp. 10–14, Sydney, Australia, June 2014.

[22] D. Geneiatakis, R. Satta, I. N. Fovino, and R. Neisse, "On the efficacy of static features to detect malicious applications in Android," *Trust, Privacy and Security in Digital Business*, vol. 9264, pp. 87–98, Springer, Cham, Switzerland, 2015.

[23] T. Chen, Q. Y. Mao, Y. M. Yang et al., "TinyDroid: a lightweight and efficient model for Android malware detection and classification," *Mobile Information Systems*, vol. 2018, Article ID 4157156, 9 pages, 2018.

[24] S. Badhani and S. K. Muttoo, "CENDroid-a cluster-ensemble classifier for detecting malicious android applications," *Computers & Security*, vol. 85, pp. 25–40, 2019.

[25] M. Kakavand, M. Dabbagh, and A. Dehghantanha, "Application of machine learning algorithms for Android malware detection," in *Proceedings of the CIIS 2018*, Phuket, Thailand, November 2018.

[26] A. Mehtab, W. B. Shahid, T. Yaqoob et al., "AdDroid: rule-based machine learning framework for Android malware analysis," *Mobile Networks and Applications*, vol. 25, no. 1, pp. 180–192, 2019.

[27] B. Rashidi, C. Fung, and E. Bertino, "Android malicious application detection using support vector machine and active learning," in *Proceedings of the 13th International Conference on Network and Service Management (CNSM)*, pp. 26–30, Tokyo, Japan, November 2017.

[28] M. Argyriou, N. Dragoni, and A. Spognardi, "Analysis and evaluation of SafeDroid v2.0, a framework for detecting malicious Android applications," *Security and Communication Networks*, vol. 2018, Article ID 4672072, 15 pages, 2018.

[29] X. Su, W. Shi, X. Qu et al., "DroidDeep: using deep belief network to characterize and detect Android malware," *Soft Computing*, vol. 24, no. 8, 2020.

[30] H. Yang, Y. Q. Zhang, Y. P. Hu et al., "A malware behavior detection system of android application based on multi-class features," *Chinese Journal of Computers*, vol. 37, 2014.

[31] M. Amin, T. A. Tanveer, M. Tehseen, F. A. Khan, F. A. Khan, and S. Anwar, "Static malware detection and attribution in android byte-code through an end-to-end deep system," *Future Generation Computer Systems*, vol. 102, pp. 112–126, 2020.

[32] W. M. Zhou, Y. Q. Zhang, and X. F. Liu, "POSTER: a new framework against privilege escalation attacks on Android," in *Proceedings of the 2013 ACM SIGSAC Conference on*

Computer & Communications Security CCS'13, pp. 1411–1413, Berlin, Germany, November 2013.

[33] T. Yamauchi, Y. Akao, R. Yoshitani et al., "Additional kernel observer to prevent privilege escalation attacks by focusing on system call privilege changes," in *Proceedings of the IEEE Conference on Dependable and Secure Computing (DSC)*, pp. 10–13, Kaohsiung, Taiwan, December 2018.

[34] S. Bugiel, L. Davi, A. Dmitrienko et al., "Poster: the quest for security against privilege escalation attacks on Android," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 741–744, Toronto, Canada, October 2011.

[35] H.-T. Lee, D. Kim, M. Park, and S.-J. Cho, "Protecting data on android platform against privilege escalation attack," *International Journal of Computer Mathematics*, vol. 93, no. 2, pp. 401–414, 2016.

[36] Y. Xu, J. Ren, Y. X. Zhang et al., "An adaptive Android security extension against privilege escalation attacks," in *Proceedings of the 15th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)/ 16th IEEE International Conference on Ubiquitous Computing and Communications (IUCC)*, Guangzhou, China, December 2017.

[37] Y. Xu, G. Wang, J. Ren, and Y. Zhang, "An adaptive and configurable protection framework against android privilege escalation threats," *Future Generation Computer Systems*, vol. 92, pp. 210–224, 2019.

[38] M. Youn-A, C. Tae-Mu, and J.-M. Kim, "A study on android privilege escalation attack by binder drive metadata management," *Advanced Science Letters*, vol. 23, no. 10, pp. 9926–9929, 2017.

[39] S. Heuser, M. Negro, P. K. Pendyala et al., "DroidAuditor: forensic analysis of application-layer privilege escalation attacks on Android," in *Financial Cryptography and Data Security*, pp. 260–268, Springer, Berlin, Germany, 2017.

[40] C. Wang, R. B. Zhang, and G. Li, "Technology of detection for privilege escalation attack on android," *Transducer and Microsystem Technologies*, vol. 36, pp. 146–148, 2017.

[41] S. Bhandari, W. B. Jaballah, V. Jain et al., "Android inter-app communication threats and detection techniques," *Computers & Security*, vol. 70, 2017.

[42] A. P. Felt, E. Chin, S. Hanna et al., "Android permission demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 627–638, Toronto, Canada, October 2011.

[43] L. Davi, A. Dmitrienko, A. R. Sadeghi et al., "Privilege escalation attacks on Android," *Information Security-International Conference*, pp. 346–360, Springer, Berlin, Germany, 2011.

[44] S. H. Qing, "Research progress on Android security," *Journal of Software*, vol. 27, pp. 45–71, 2016.

[45] P. Wang, *Research on Attack Protection Method of Privilege Escalation Attack for Android Platform*, Harbin Engineering University, Harbin, China, 2016.

[46] K. W. Y. Au, Y. F. Zhou, Z. Huang et al., "PScout: analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 217–228, Raleigh, NC, USA, October 2012.

[47] S. Arzt, S. Rasthofer, C. Fritz et al., "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 259–269, June 2014.

[48] R. Milner, *Communicating and Mobile Systems the Pi-Calculus*, United Kingdom at the University Press, Cambridge, UK, 1999.

[49] W. Z. Yang, C. H. Huang, and Z. Y. Zhang, "A software trustworthiness metric model based on process algebra," *Journal of WUT (Information & Management Engineering)*, vol. 33, no. 6, 2011.

[50] Secure Software Engineering at Paderborn University and TU Darmstadt, "DroidBench-benchmarks," 2019, https://blogs.uni-paderborn.de/sse/tools/droidbench/.

[51] WT Software Innovation Lab, "Test sample set," 2019, https://pan.baidu.com/s/1Xg6viFPIza68GU89sYfnGg.