IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 6 September 2022, accepted 21 September 2022, date of publication 28 September 2022,
date of current version 17 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3210386

RESEARCH ARTICLE

A Formal Method for Description and Decision of Android Apps Behavior Based on Process Algebra

DONGKUI LIANG[1,2], LIMIN SHEN[1], (Member, IEEE), ZHEN CHEN[1], CHUAN MA[1,2], AND JIAYIN FENG[1]

[1]School of Information Science and Engineering, Yanshan University, Qinhuangdao 066004, China
[2]Engineering Training Center, Yanshan University, Qinhuangdao 066004, China

Corresponding author: Limin Shen (shenllmm@sina.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61772450, in part by the Hebei Natural Science Foundation under Grant F2019203287 and Grant F2017203307, in part by the Science and Technology Research Project of Colleges and Universities in Hebei Province under Grant QN2020183, and in part by the Hebei Postdoctoral Research Program under Grant B2018003009.

ABSTRACT Android is the most popular mobile platform, and it has become a primary malware target. Existing behavior-based Android malware detection methods suffer from false positive and false negative problems, which lead to low detection accuracy. Formal theory is crucial in studying the behaviors of Android applications characterized by high concurrency, interaction, and mobility. However, existing formal methods mainly focus on specific issues and lack the essential abstraction and high-level description of application behavior. In this study, we propose a formal method for the description and decision of application behavior based on process algebra. First, we propose a formal method for describing application behavior at a component level using process algebra. By extending π-calculus theory, we establish the mapping relationship from the Android application to process algebra, and present the semantics and evolution rules of behavior based on process algebra. Second, we describe the behavior of four types of components in applications and characterize concurrent interactions of components using process algebra expressions. Third, we define the behavior equivalence and simulation mechanism for application behavior analysis and propose the decision rules based on weak simulation. Finally, we discuss a demonstration case, which includes malicious behavior, to demonstrate the feasibility and effectiveness of the proposed method. The results show that our method can accurately describe and analyze application behavior, which provides theoretical support for technologies and methods of behavior-based detection.

INDEX TERMS Android malware detection, behavior formalization, process Algebra, process equivalence, simulation mechanism.

I. INTRODUCTION

Smartphones are being widely used with the development of Internet and IOT. Annual worldwide sales of smartphones stand at around 1.56 billion units, with Android accounting for 85.1% [1]. In the next few years, Android smartphones will remain more than 85% of the market [2]. Nowadays Android has become the most popular mobile platform, and the number of Android Apps is growing rapidly. However, due to the openness of free source, Android is highly vulnerable to malware attacks and has become the primary target of malware. Malware not only expose the private information and confidential data in smartphones to the risk of being stolen [3], but also impact the confidentiality, availability or integrity of system [4]. The efficient detection of Android malware has become a popular research topic.

The technologies of malware detection are divided into three categories: static analysis, dynamic analysis, and hybrid analysis [5]. Static analysis decomposes Android Application Package (APK) files by reverse engineering and extracts various features from the disassembly code without running source code [6], [7], [8]. It has high accuracy and efficiency in

The associate editor coordinating the review of this manuscript and approving it for publication was Eyuphan Bulut.

detecting known malicious code, but has high false negative in detecting unknown malicious code because it cannot deal with code confusion and dynamic code loading. Dynamic analysis runs Apps and analyzes them by monitoring runtime behaviors and data [9], [10], [11]. It performs well in detecting all types of code, but requires more resources and time costs. Moreover, some behaviors cannot be recognized correctly because the execution path of the application cannot be fully traversed. Hybrid analysis uses both static analysis and dynamic analysis by combining their advantages [12], [13], [14].

A variety of malware detection approaches are proposed based on these techniques. To further improve the detection of malware, researchers conducted a series of studies. On the one hand, technologies such as machine learning were widely applied to detect malware. Although the detection model based on machine learning had false positives in intrusion detection [15], Liu *et al.* [16] discussed the application and prospect of machine learning and suggested that it would be effective and promising in malware detection. On the other hand, researchers carried out theoretical research to promote the development and innovation of malware detection technology. Nowadays a series of formal theories have been proposed to analyze permission frameworks, component interactions, and application behavior.

By summarizing existing research, we observed that malware detection based on behavior performs satisfactorily but still suffers from false positives and false negatives. Formal theory is crucial in analyzing the behavior of Android Apps characterized by high concurrency, interaction, and mobility. However, existing formal methods mainly focus on modeling and validating specific issues, which lack the essential abstraction and modeling of behavior. In this paper, we use process algebra as an abstract language to analyze the behavior of applications, and propose a formal method for the description and decision of application behavior. Behaviors are described using process algebra expressions and operators, and can be analyzed by the inference and calculation mechanism of process algebra. The feasibility and effectiveness of the method in this study are verified by discussing a case containing malicious behavior. To the best of our knowledge, this is the first time process algebra has been employed to research behavior formalization and decision in behavior-based malware detection.

The formal method in this paper can help understand and reveal the essence and laws of application behavior. Our research will help any researcher who wants to carry out research in behavior-based Android malware detection in various domains such as static, dynamic, and hybrid, and guide them with theoretical support. The main contributions of this paper are summarized as follows:

1) A formal theory was proposed for describing the application behavior based on process algebra. By extending π-calculus theory, we abstract the behavior elements of Apps and map them to process algebra. In section III, we defined the basic behavior semantics of application behavior and proposed a formal definition for component behavior. Based on the inference and calculation mechanism, we presented the evolution rules for analyzing behavior.
2) Based on the formal theory proposed in this study, we described the behavior of four types of Android application components and formalized the behavior of application. The states and concurrent interactions of components are described using process algebra expressions.
3) On the basis of behavior formalization, we proposed a method for application behavior decisions based on weak simulation. According to process equivalence, we defined strong simulation and weak simulation for discussing the equivalence of behavior, and proposed decision rules for application behavior based on weak simulation.

The reminder of this paper is organized as follows. Section II discusses the related work. Section III establishes the formal theory for describing application behavior based on process algebra. Section IV describes the behaviors and concurrent interactions of the four types of components in applications. Section V defines strong simulation and weak simulation of behavior equivalence, and proposes decision rules based on weak simulation. Section VI discusses a demonstration case derived from real application that contains malicious behavior. Section VII concludes our work.

II. RELATED WORK

Currently, malware detection has been heavily researched. Researchers apply technologies such as machine learning to the analysis of the static and dynamic features and have proposed many methods from various perspectives.

Li *et al.* [17] proposed a framework based on association mining, which uses association rules derived from N-gram features mining to achieve efficient detection. In [18] and [19], deep learning technology was used to detect malware and performed well. In [20], a malware detection approach was proposed to analyze Apps at source code level by utilizing a deep traversal tree neural network. In [21], they converted the bytecodes of the ''classes.dex'' files to visual images, and proposed a vision-based detection model composed of 16 CNN algorithms. Zhang *et al.* [22] presented a hybrid representation learning approach to clustering weakly-labeled malware by preserving heterogeneous information from multiple sources. In [23], a novel framework was presented to improve the malware detection for Android IoT devices by combining the advantages of both machine learning and block chain technology. In [24], a signature-based framework was proposed to detect malware using API calls and other features. Zhang *et al.* [25] proposed a detection method based on the method-level correlation relationship of abstract API calls. The accuracy on malware datasets Drebin and AMD was 96%. Huang *et al.* [26] proposed a sequence-to-sequence neural network to investigate a sequence of Windows API calls recorded from malware

execution and produce tags to label their malicious behavior. Arora *et al.* [27] constructed graphs for malicious and benign applications by extracting permission pairs from manifest file, and detected malware by comparing these graphs. Xiao *et al.* [28] combined dynamic features and static features into composite features to detect malware and achieved an accuracy of 97.12%. In [29], they defined different corresponding behaviors and correlated features at four levels, and proposed a host-based detection system to classify behaviors of malware. Base on machine learning, [30] and [31] used permission as dynamic and static features to detect malware, respectively. Wang *et al.* [32] used seven feature selection algorithms to select permissions, API calls, and opcodes. The results of each algorithm were merged to obtain a new feature set to classify applications. Fatima *et al.* [33] used evolutionary genetic algorithm to construct the optimal feature subset and then used the subset to train classifier for malware detection. In [34], useful API calls were used as features to construct API subsets of malicious and benign applications to classify applications. In [35], they used permission and action repetition as static and dynamic features to identify malware by leveraging machine learning, and proved their efficiency and influential roles in detecting malware. Arslan *et al.* [36] designed a permission-based detection system. The system used hybrid analysis to detect malware and achieved an accuracy of 91.95%.

To reduce false positives and false negatives, researchers have introduced formal theory into the research of malware detection to promote the development of detection techniques and methods. In [37], π-calculus was used to analyze and validate security of software behavior. Chaudhuri [38] proposed a semantic-based formal description of Android Apps to help understand behavior security. Jia *et al.* [39] proposed a formal model for Android components based on process algebra, aiding developers in implementing least permission. Shen *et al.* [40] proposed a behavior detection method based on function and process algebra for the detection of privilege escalation attacks in Android Apps. To describe the interactions between Apps, [41] proposed a formal interoperability semantic to help understand and infer Android interoperations. In [42], a methodology based on formal methods was proposed to help understand and identify obfuscation codes. To help understand permission model of Android Apps, researchers have proposed a series of formal theories. He [43] presented a formal model of permission framework using high level Petri nets. It precisely defined relationships among different levels and could be used to analyze permissions and their combinations. In [44], formal methods were used to verify the security mechanisms of Android, and a comprehensive specification of permission model was developed to describe and justify the attributes of expected behavior in Apps. In [45], a formal approach was proposed to help identify the potential defects in Android permission protocol. Khan *et al.* [46] used theorem proving approach to analyze the security of Android permission, and proposed a language-based formal model for the analysis of Android security. In theoretical research, formal theory can be validated by theorem proving and existing tools. Scyther and Tamarin are automatic tools for the formal analysis and verification of security protocols [47], [48]. MWB (Mobility Workbench) is a tool for manipulating and analyzing mobile concurrent systems described in π-calculus or CCS. In [40], MWB was used to analyze formal expressions of application behavior to help detect collusion attacks.

Researchers have proposed various methods in malware detection and developed formal theories to support these techniques. However, existing detection methods based on behavior suffer from false positives and false negatives. Furthermore, existing formal research lacks the essential abstraction and description of application behavior. In view of the concurrency and interaction characteristics of Android Apps, we extend the π-calculus theory, which is suitable for mobile concurrent systems, and propose a formal method for the description and decision of application behavior. Based on the behavior semantics and rules proposed in this paper, application behavior is described using process expressions and operators, and then is determined according to behavior equivalence mechanism. In this study, the theory for the description and decision of application behavior is validated by analyzing a demonstration case derived from real Android Apps. The next section discusses the modeling of process algebra elements for application behavior, and present the semantics and evolution rules for application behavior.

III. SEMANTICS AND EVOLUTION RULES FOR ANDROID APPLICATION BEHAVIOR

An application is made up of four types of components: activity, service, content provider, and broadcast receiver. Components communicate with content provider through Uri, and communicate with other components through Intent. Intent mechanism is a run-time binding mechanism and is a communication mode of Android Apps. It is used to transfer information and data between components and its intentional or unintentional improper use may lead to security problems such as information leaks, malicious calls, and component hijacking [49], [50], [51]. Therefore, researchers had used Intent as important feature to detect malware [52], [53], [54]. In the modeling of application behavior, Intent becomes more important because it is used in components communication.

A. PROCESS ALGEBRA ELEMENTS FOR APPLICATION BEHAVIOR

In behavior analysis of Apps, analyzing all paths of behaviors is a huge and complex task. However, formal method is effective for studying complex system and plays an important role in behavior analysis. Process algebra, as the representative of formal methods, is suitable for analyzing concurrent system. Bekic proposed the basic semantics of process algebra consisted of at least three operators and seven operation rules [55]. In 1982 [56], Bergstra proposed the specific definition of process algebra. Nowadays, process algebra has developed many branches and extensions [57].

CCS [58] is a process algebra theory proposed by R. Milner and has been widely used in the analysis and validation of concurrent systems. Since CCS cannot describe the change of topology in mobile concurrent systems, R. Miller further extended CCS to proposed the π-calculus [59]. It is a named calculus and is especially suitable for mobile concurrent system. Its structured semantics can formally describe the concurrency and interaction of behaviors, and achieve the composition, decomposition and reduction of behavior. Additionally, π-calculus provides a mature and sound simulation theory to study behavior equivalence in the analysis of system behavior [60]. In π-calculus, an action or event is a behavior unit, which is called atomic behavior. Behavior is a series of actions or events, which is called a process. By introducing the concepts of name and channel from communication, messages, events, and actions are mapped to names, communication ports are mapped to channels. Names could be transmitted in channels. By adding prefix action to describe interactions, process expressions can be used to describe the behaviors of system.

The term "*process*" in process algebra is not the process of operating system, but the behavior mode of system. It describes system behavior through a finite set of actions, which can be further analyzed by inference and calculation mechanisms. Unless otherwise noted, the process mentioned below is "*process*". Android Apps are complex systems composed of collaborative and concurrent components, and attackers are no longer limited to a single attack mode, but implement collaborative collusion attacks through the communication of components. Therefore, we extend π-calculus suitable for studying mobile concurrent systems to the research of application behavior.

The behavior of Android application can be described by the behavior of the concurrent and interactive component instances which are instantiated in the process of operating system. At the code level, a statement or function call is a basic behavior, called behavior unit. Component behavior is the external representation of a series of program statements and function calls. The behavior unit is abstracted as an action. The actions in a component are divided into intra-action and inter-action according to whether they interact with the outside of the component. Intra-action which is independent of external environment can only describe the internal evolution of component, so it is simple and easy to describe. Inter-action can describe not only the internal behavior of the component, but also the interaction with the external environment, so the behavior description is very complicated. By introducing the semantics and rules of process algebra, we establish the mapping from Android Apps to the π-calculus of process algebra as follows:

- Modeling the component instance of Android Apps as process. The instance can be run in one or different processes of the operating system.
- Modeling the interaction of component instances as process communication.
- Modeling the behavior unit in program code, such as statement, function call and method call as action.
- Modeling variable, parameter, data, message, event, entity attribute, intent and other elements used in component interaction as name.

TABLE 1. Mapping relationship from Android Apps to π-calculus.

Android Apps	π-calculus
Android Components (Activity, Service, Broadcast Receiver, Content Provider)	Process
Parameter, Data, Intent, Message, etc.	Name
Function Call, Method Call, Statment	Action
Interaction	Communication

The mapping relationship between elements of Android Apps and π-calculus is established as shown in Table 1.

By extending π-calculus, we propose the process algebra elements in Android application behavior, and use process algebra as an abstract language to describe the behaviors of application. In the following, we propose the semantics and rules for application behavior formalization.

B. BASIC SEMANTICS OF APPLICATION BEHAVIOR

Components in Android Apps are treated as the subject and object of behavior. Application behavior is described with subject, object, action, input of subject, output of object, and state of component. The semantics of application behavior are given below.

Definition 1 (Name): The concepts of data, parameters, and communication channels, as well as behavior information and state information, are unified and abstracted as name, which can be denoted as $a, b, \ldots \in$ Name.

Name is the basic element of behavior and is transmitted as message in process. $\overline{a}$ is the complementary name of a, and $\overline{\overline{a}} \stackrel{\text{def}}{=} a$, $\overline{a}, \overline{b}, \ldots \in \overline{\text{Name}}$. An sequence of ordered names $a_1, \ldots a_n$ can be denoted as $\vec{a}$, then $P(\vec{a}) = P(a_1, \ldots a_n)$. If $\vec{a}$ and $\vec{b}$ are name-sequences of length n and P is a process expression, then $\{\vec{b}/\vec{a}\}$P means that each a_i in P are replaced by b_i separately, which is called α conversion. If the length of $\vec{a}$ and $\vec{b}$ are both one, $\{\vec{b}/\vec{a}\}P$ can be denoted as $\{b/a\}P$.

Definition 2 (Observation & Reaction): The actions are divided into observation and reaction according to whether they can be observed from outside.

- Action t is an observation if it can be observed through its interaction with component, or through the complementary action $\overline{t}$ after t interacts with component. Ation t is also called observation action.
- Action τ is a reaction if it cannot be observed outside, but can only be observed through the results of internal interaction.

In the concurrent execution of processes, the occurrence of actions in a process affects itself and the interaction between processes. The inter-process interactions can be observed, but

interactions between parts of a process are like occurring in a black box and cannot be observed externally.

If action t in process P can initiate an interaction with process Q, we use $\bar{t}$ to represent the action in Q that interacts with P, where $\bar{t}$ is a complementary action of t, and $\bar{\bar{t}} \stackrel{\text{def}}{=} t$. In this way, a pair of labels $(t, \bar{t})$ represents the inter-process interactions. We can observe t by observing the occurrence of $\bar{t}$. As a result, we can confirm interaction by observing the occurrence of t or $\bar{t}$. Action t is called observation and is an inter-action with a complementary action $\bar{t}$.

If Action τ in process P only affects the process itself, it will not directly affect the process interaction. As a result, τ cannot be observed directly, but only through the results after τ occurs. Action τ is called reaction and is an intra-action without complementary action.

Definition 3 (Prefix Action): Let P be a process, π be a action, then $\pi.P$ means that π must occur before P becomes active. Action π is a prefix action of P and "." is called prefix operator. There is a sequential relationship between π and P. The prefix action π is defined recursively as follows:

$$\pi ::\stackrel{\text{def}}{=} \begin{cases} t\,(x) & \text{receiving action} \\ \bar{t}\langle x\rangle & \text{sending action} \\ [x\ R\ y]\,\pi & \text{matching action} \\ \tau & \text{internal action} \end{cases} \tag{1}$$

$t\,(x)$ means action t receives x, where x is a restricted name. $\bar{t}\langle x\rangle$ means action t sends x, where x is a non-restricted name. $[x\ R\ y]\,\pi$ means action π could be executed when $[x\ R\ y]$ holds, where R is a logical operator. τ is the reaction defined in Definition 2.

An expression with the structure "$\pi.P$" is called a guardian expression, and P is guarded by π. A expression with the structure "$P = a.P$" is called a recursive guardian expression, and P is recursive. To simplify guardian expressions without affecting the semantics they represent, the expressions can be reduced to one expression according to the following rules:

- None of the expressions to be simplified are recursive.
- The expression obtained by reduction is not recursive.

For example, Let $P = a.P_1$, $P_1 = b.P_2$, where P and P_1 are not recursive. They could be reduced to $P = a.b.P_2$ if $P = a.b.P_2$ is not recursive. However, $P = a.Q$, $Q = b.R$, and $R = c.P$ are not recursive, but they cannot be reduced to $P = a.b.c.P$ because the result is recursive.

In communication, $(t, \bar{t})$ means action synchronization, where $\bar{t}$ is the complementary action of observation t. If $t\,(x)$ and $\bar{t}\langle y\rangle$ in expression $(t\,(x)\,.P + M)|(\bar{t}\langle y\rangle.Q + N)$ are not guarded by other actions, they constitute a sync-action-pair $(t\,(x)\,, \bar{t}\langle y\rangle)$. The firing of $(t\,(x)\,, \bar{t}\langle y\rangle)$ in process will lead to the occurrence of $(t\,(x)\,.P + M)|(\bar{t}\langle y\rangle.Q + N) \stackrel{t}{\rightarrow} y/x\}P|Q$. There is at least one channel in component communication, which means there is at least one sync-action-pair.

Definition 4 (Transition): $S = \{s_0, s_1, s_2, \ldots\}$ is the set of all possible states of system, and $\text{Act} = \{t | s \in S, \exists s' \in S$, and $s \stackrel{t}{\rightarrow} s'\}$ is the set of actions on S. The process of system state changing from s to s' under action t is called a transition of system, which is denoted as $s \stackrel{t}{\rightarrow} s'$. A transition can be represented as an ordered triple (s, t, s'), and the set of all system transitions is denoted as $T = \{(s, t, s') | s \in S, \exists t \in \text{Act}, \exists s' \in S$, and $s \stackrel{t}{\rightarrow} s'\}$.

Transitions $s_i \stackrel{a}{\rightarrow} s_{i+1}$ and $s_i \stackrel{b}{\rightarrow} s_{i+1}$ can be abbreviated as $s_i \stackrel{a;b}{\rightarrow} s_{i+1}$. Transitions $s_i \stackrel{a}{\rightarrow} s_{i+1} \stackrel{c}{\rightarrow} s_{i+2}$ can be abbreviated as $s_i \stackrel{<a,c>}{\longrightarrow} s_{i+2}$ without considering the intermediate state, where s_{i+2} is called a derived state of s_i under actions $\langle a, c\rangle$.

Definition 5 (Trace): A series of changes of the system state can be represented as $s_0 \stackrel{t_1}{\rightarrow} s_1 \stackrel{t_2}{\rightarrow} \ldots\ldots \stackrel{t_{n-1}}{\longrightarrow} s_{n-1} \stackrel{t_n}{\rightarrow} s_n$, the sequence of ordered actions $\langle t_1, t_2, \ldots, t_n\rangle$ is called a trace of system behavior. The set of traces of all system behaviors is represented as traces(S).

Definition 6 (Component Behavior): Component behavior is composed of names, processes, and symbols according to the BNF paradigm and the following syntax:

$$P ::= \pi.P \mid P_1 + P_2 \mid P_1 \mid P_2 \mid \text{new } a\ P \mid !P \mid \underline{0} \mid \surd \tag{2}$$

1) $\pi.P$ is a guardian expression, which means P will be active while action π occurs. π is the prefix action defined in Definition 3. "." is the prefix operator, which is another expression of sequence operator.
2) $P_1 + P_2$ is a selection structure, which means P_1 or P_2 will be selected and be active according to the context. "+" is the selection operator.
3) $P_1 \mid P_2$ is a parallel structure, which means that P_1 and P_2 are executed concurrently. The result depends on the context while there are guardian structures during concurrency. "|" is the parallel operator.
4) new aP means that name a is restricted within P and can only be used inside P. It can be represented as (new a)P, new $(a)P$ or $(v\ a)P$. "new" is the restriction operator, and a is a restricted name of P.
5) $!P$ means that P replicates itself and a copy of P is created. "!" is the replication operator.
6) $\underline{0}|\surd$ means the end of process, where $\underline{0}$ indicates that the process is forcibly terminated and $\surd$ indicates that the process ends successfully.

C. EVOLUTION RULES

All rules are based on the atomicity of action. Action a is atomic means a can be executed and terminated successfully, which is represented as $\overline{a \stackrel{a}{\rightarrow} \surd}$.

1) RULES FOR PREFIX OPERATOR

- $a.P \stackrel{a}{\rightarrow} P$
  While observation a occurs, process will transfer to P and P becomes active.
- $[\tau]\,.P \stackrel{[\tau]}{\rightarrow} P$
  While reaction τ occurs, process will transfer to P and P becomes active.
- $\frac{a \stackrel{[\tau]}{\rightarrow} \surd}{a.P \stackrel{[\tau]}{\rightarrow} P}$

Observation a can execute successfully while reaction τ occurs. While reaction τ occurs, process will transfer to P and P becomes active.

2) RULES FOR SELECTION OPERATOR

- $a.P_1 + P_2 \xrightarrow{a} P_1$
  While observation a occurs, $a.P_1$ is executed. Then process will transfer to P_1 and P_1 becomes active.
- $\frac{P_1 \xrightarrow{a} P_1'}{P_1+P_2 \xrightarrow{a} P_1'}$
  While observation a occurs, P_1 will be executed and terminated successfully. Then process will transfer to P_1' and P_1' becomes active.
- $\frac{P_1 \xrightarrow{a} P_1' P_2 \xrightarrow{a} P_2'}{P_1+P_2 \xrightarrow{a} P_1'+P_2'}$
  While observation a occurs, P_1 and P_2 can be executed and terminated successfully. Then process will transfer to another selection structure $P_1' + P_2'$.

These rules also hold for reaction τ.

3) RULES FOR PARALLEL OPERATOR

- $\frac{P_1 \xrightarrow{a} P_1'}{P_1|P_2 \xrightarrow{a} P_1'|P_2}$ While observation a occurs, P_1 will transfer to P_1' and P_2 has no change. Then process will transfer to another parallel structure $P_1'|P_2$. The rule holds for reaction τ.
- $\frac{P_1 \xrightarrow{a} P_1' P_2 \xrightarrow{\overline{a}} P_2'}{P_1|P_2 \xrightarrow{a} P_1'|P_2'}$
  $(a, \overline{a})$ is a pair of synchronous actions used to represent communication between P_1 and P_2, which is discussed in Definition 3. $P_1 \xrightarrow{a} P_1'$ indicates P_1 transfer to P_1' after a occurs, expressed as $P_1 = a.P_1'$; $P_2 \xrightarrow{\overline{a}} P_2'$ indicates $\overline{a}$ in P_2 occurs as the complementary action of a and P_2 transfer to P_2', expressed as $P_2 \xrightarrow{\overline{a}} P_2'$. According to the first rule for prefix operator, there are expressions $a.P_1' \xrightarrow{a} P_1'$ and $\overline{a}.P_2' \xrightarrow{\overline{a}} P_2'$. While a occurs, P_1 initiates interaction with P_2 and $P_1|P_2$ will transfer to another parallel structure $P_1'|P_2'$.
- $(a.P_1 + M)|(\overline{a}.P_2 + N) \xrightarrow{a} P_1|P_2$
  $(a, \overline{a})$ means that actions are synchronous, which is discussed in Definition 3. While observation a occurs, process will select $a.P_1$ and $\overline{a}.P_2$ to execute and can be expressed as $a.P_1|\overline{a}.P_2$. According to the first rule for prefix operator, there are $a.P_1 \xrightarrow{a} .P_1$ and $\overline{a}.P_2 \xrightarrow{\overline{a}} P_2$. According to the second rule for parallel operator, the process will transfer to another parallel structure $P_1|P_2$.
- $(a(\vec{x}).P_1 + M)|(\overline{a}\langle\vec{y}\rangle.P_2 + N) \xrightarrow{a} \{\vec{y}/\vec{x}\}P_1|P_2$
  $\vec{x}$ and $\vec{y}$ are name-sequences of length n, which is discussed in Definition 1. $(a, \overline{a})$ is discussed in Definition 3. While observation a occurs, process will select $a(\vec{x}).P_1$ and $\overline{a}\langle\vec{y}\rangle.P_2$ to excute and then transfer to another parallel structure $\{\vec{y}/\vec{x}\}P_1|P_2$.

TABLE 2. Examples of ICC commands.

sendBroadcast(Intent intent) sendOrderedBroadcast(Intent intent, String receiverPermission) sendStickyBroadcast(Intent intent)
startActivity(Intent intent) startActivityForResult(Intent intent, int requestCode)
startService(Intent intent) stopService(Intent intent) bindService(Intent intent, ServiceConnection conn, int flags) unbindService(Intent intent, ServiceConnection conn, int flags)
query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) insert(Uri uri, ContentValues values) update(Uri uri, ContentValues values, String selection, String[] selectionArgs) delete(Uri uri, String arg1, String[] arg2)

IV. BEHAVIOR AND INTERACTION DESCRIPTION OF COMPONENTS IN ANDROID APPLICATION

Application behavior consists of the behavior of components that are executed concurrently. In the concurrent system S, the size of traces(S) increases too rapidly as the actions in the interaction increase. Consider two processes with n actions, where actions can be executed interactively in any correct order. Using S(n) to represent the scale of concurrency, the following conclusions are obtained after calculation:

$$\mathrm{S}(1) = 2, \mathrm{S}(2) = 6, \mathrm{S}(3) = 20, \ldots$$

It can be predicted that $S(n)$ will increase geometrically with n, which is not conducive to further analysis.

In application behavior analysis, considering all actions to fully characterize and analyze application behavior will bring an unbearable burden. While studying application behavior at the component level, intra-action can affect the state, but has little impact on transitions. Therefore, we ignore invisible intra-actions and describe behaviors by inter-actions which can be observed and can trigger component interaction. The commands of inter-component communication (ICC) are abstracted as actions to describe the behavior of application. Some ICC commands are shown in Table 2.

A. ACTIVITY

Activity provides a visual interface for interacting with users. It has four states: running, paused, stopped, and destroyed. The state transitions are implemented by life cycle functions, and the processes are described as follows:

$$\mathrm{Activity}^{\mathrm{null}} \xrightarrow{<onCreat, onStart, onResume>} \mathrm{Activity}^{\mathrm{running}},$$

$$\mathrm{Activity}^{\mathrm{running}} \xrightarrow{onPause} \mathrm{Activity}^{\mathrm{paused}},$$

$$\mathrm{Activity}^{\mathrm{running}} \xrightarrow{<onPause, onStop>} \mathrm{Activity}^{\mathrm{stopped}},$$

$$\mathrm{Activity}^{\mathrm{running}} \xrightarrow{<onPause, onStop, onDestroy>} \mathrm{Activity}^{\mathrm{destroyed}},$$

$$\mathrm{Activity}^{\mathrm{paused}} \xrightarrow{onResume} \mathrm{Activity}^{\mathrm{running}},$$

$$\mathrm{Activity}^{\mathrm{stopped}} \xrightarrow{<onRestart, onStart, onResume>} \mathrm{Activity}^{\mathrm{running}}.$$

In Android Apps, an activity is presented as a page, which is the carrier of information. An activity can be the initiator or receiver of the interaction between activities, in which intent is used to encapsulate data to exchange information. Behaviors of activity include the initiation and acceptance of request, the return and reception of result, and the closure of activity, which are defined as follows:

$$\overline{startActivity}\langle x\rangle.\text{Activity},$$
$$startActivity\ (y)\ .\text{Activity},$$
$$\overline{setResult}\langle m\rangle.\text{Activity},$$
$$setResult\ (n)\ .\text{Activity},$$
$$finish.\text{Activity}.$$

The state transitions of activity vary depending on the behavior model and are described as follows:

$$\overline{startActivity}\langle x\rangle.\text{Activity}^{\text{running}} \xrightarrow{startActivity} \text{Activity}^{\text{stopped}},$$
$$startActivity\ (y)\ .\text{Activity}^{\text{null}} \xrightarrow{startActivity} \text{Activity}^{\text{running}},$$
$$finish.\text{Activity}^{\text{running}} \xrightarrow{finish} \text{Activity}^{\text{destroyed}},$$
$$\overline{setResult}\langle m\rangle.finish.\text{Activity}^{\text{running}} \xrightarrow{<setResult,finish>} \text{Activity}^{\text{destroyed}},$$
$$setResult\ (n)\ .\text{Activity}^{\text{paused}} \xrightarrow{setResult} \text{Activity}^{\text{running}}.$$

In the interaction between activities, action $\overline{setResult}$ must wait for the action *finish* to execute before forming a sync-action-pair with *setResult* to return data to the initiator. The actions in this process always appear in the form of $\langle\overline{setResult}, finish, setResult\rangle$. Therefore, we replace it by $\langle setResult, setResult\rangle$ and describe the process as follows:

$$\overline{setResult}\langle m\rangle.\text{Activity}^{\text{running}} \xrightarrow{setResult} \text{Activity}^{\text{destroyed}},$$
$$setResult\ (n)\ .\text{Activity}^{\text{paused}} \xrightarrow{setResult} \text{Activity}^{\text{running}}.$$

A typical interaction between activities is ''The display page switches from page A_1 to page A_2. After time T, page A_2 is closed and page A_1 returns to the foreground''. The actions executed by A_2 in the duration T is represented as e, and the interaction process is described as follows:

$$\overline{startActivity}\langle x\rangle.setResult\ (n)\ .A_1 \mid startActivity\ (y)\ .$$
$$e.\overline{setResult}\langle m\rangle.A_2.$$

Page A_1 is stopped and page A_2 is running while the display page is switched from A_1 to A_2. The behaviors and state transitions are described as follows:

$$\overline{startActivity}\langle x\rangle.A_1^{\text{running}} \mid startActivity\ (y)\ .A_2^{\text{null}} \xrightarrow{startActivity} A_1^{\text{stopped}} \mid \{x/y\}A_2^{\text{running}},$$
$$((onPause.A_1^{\text{paused}}).onCreate.onStart.onResume.A_2^{\text{running}}).onStop.A_1^{\text{stopped}}.$$

Page A_2 is closed and page A_1 returns to the foreground after time T. At this time, A_1 is running and A_2 is destroyed. The behaviors and state transitions are described as follows:

$$\overline{setResult}\langle m\rangle.A_2^{\text{running}} \mid setResult\ (n)\ .A_1^{\text{paused}} \xrightarrow{setResult} A_2^{\text{destroyed}} \mid \{m/n\}A_1^{\text{running}},$$
$$((onPause.A_2^{\text{paused}}).onRestart.onStart.onResume.A_1^{\text{running}}).onStop.onDestroy.A_2^{\text{destroyed}}.$$

In the interaction between activity and other components, the process description and state transitions are as follows:

$$\overline{startActivity}\langle x\rangle.\text{Component} \mid startActivity\ (y)\ .\text{Activity} \xrightarrow{startActivity} \text{Component} \mid \{x/y\}\text{Activity}^{\text{running}},$$
$$onCreate.onStart.onResume.\text{Activity}^{\text{running}}.$$

B. SERVICE

Service provides the services required by users and business logic. It has two states: running and destroyed. The state transitions are described as follows:

$$\text{Service}^{\text{null}} \xrightarrow{<onCreat,onStartCommand>} \text{Service}^{\text{running}},$$
$$\text{Service}^{\text{null}} \xrightarrow{<onCreat,onBind>} \text{Service}^{\text{running}},$$
$$\text{Service}^{\text{running}} \xrightarrow{onDestroy} \text{Service}^{\text{destroyed}}.$$

Service has two startup modes: starting and binding. Components use intent to encapsulate data to communicate with service. Service is responsible for responding to service requests of components, including the start, bind, unbind, and stop of the service. The behaviors are defined as follows:

$$startS\ (y)\ .\text{Service},$$
$$stopS\ (y)\ .\text{Service},$$
$$bindS\ (y)\ .\text{Service},$$
$$unbindS\ (y)\ .\text{Service}.$$

A running service can be bound by multiple components. To reflect the changes of the component instances connected to service, the collection of currently connected instances is represented as *clients*. Then, $clients\ (c)\ .\text{Service}$ indicates that instance c is added to *clients*, $\overline{clients\ (c)}.\text{Service}$ indicates that instance c is removed from *clients*. In addition, Service provides a method *stopSelf* () which is used to stop service instance when *clients* is empty. Consequently, there are some more behaviors defined as follows:

$$(\text{v}\ clients)\ \text{Service},$$
$$clients\ (c)\ .\text{Service},$$
$$\overline{clients\ (c)}.\text{Service},$$
$$stopSelf\ .\text{Service}.$$

Calling *startService*() to communicate with service will first create an instance of service by calling *onCreate*(), and then call the callback method *onStartCommand*(). If the instance already exists, only the callback method is executed. It is necessary to call *stopService*() or *stopSelf* () to stop

a running service. The behaviors and state transitions of starting/stopping the service are described as follows:

$$\overline{startS}\langle x\rangle.\text{Component} \mid startS\ (y)\ .\text{Service}^{\text{null}} \xrightarrow{startS} \text{Component} \mid \{x/y\}Service^{\text{running}},$$
$$onCreat.onStartCommand.\text{Service}^{\text{running}},$$
$$\overline{stopS}\langle x\rangle.\text{Component} \mid stopS\ (y)\ .\text{Service}^{\text{running}} \xrightarrow{stopS} \text{Component} \mid \text{Service}^{\text{destroyed}},$$
$$onDestroy.\text{Service}^{\text{destroyed}},$$
$$stopSelf\ .\text{Service}^{\text{running}} \xrightarrow{stopSelf} \text{Service}^{\text{destroyed}},$$
$$onDestroy.\text{Service}^{\text{destroyed}}.$$

Calling *bindService*() to communicate with service will bind component to the service instance. If the service is not running, *onCreate*() is executed first to create the instance, and then callback method *onBind*() is executed. Otherwise, only method *onRebind*() is executed. To unbind component from service, it is necessary to call *unbindService*() unless the component bound to the service has been destroyed. A running service can be stopped until all components bound to service are unbound. The behaviors and state transitions of binding/unbinding the service are described as follows:

$$\overline{bindS}\langle x\rangle.\text{Component} \mid bindS\ (y)\ .\text{Service}^{\text{null}} \xrightarrow{bindS} \text{Component} \mid clients\ (x)\ .\text{Service}^{\text{running}},$$
$$onCreat.onBind.\text{Service}^{\text{running}},$$
$$\overline{unbindS}\langle x\rangle.\text{Component} \mid unbindS\ (y)\ .\text{Service}^{\text{running}} \xrightarrow{unbindS} \text{Component} \mid \overline{clients\ (x)}.([clients \neq \emptyset]\ \text{Service}^{\text{running}} + [clients = \emptyset]\ \text{Service}^{\text{destroyed}}),$$
$$onUnbind.([clients \neq \emptyset]\ \text{Service}^{\text{running}} + [clients = \emptyset]\ onDestroy.\text{Service}^{\text{destroyed}}).$$

Components can call *startService*() and *bindService*() to communicate with service and the behaviors are different according to the calling order. Generally, service is started by calling *startService*(), and then component can be bound to the service by calling *bindService*(). The behaviors and state transitions are described as follows:

$$\overline{startS}\langle x\rangle.\text{Component} \mid startS\ (y)\ .\text{Service}^{\text{null}} \xrightarrow{startS} \text{Component} \mid \{x/y\}\text{Service}^{\text{running}},$$
$$\overline{bindS}\langle m\rangle.\text{Component} \mid bindS\ (\text{n})\ .\text{Service}^{\text{running}} \xrightarrow{bindS} \text{Component} \mid clients\ (m)\ .\text{Service}^{\text{running}},$$
$$(onCreat.onStartCommand.\text{Service}^{\text{running}}).onBind.\ \text{Service}^{\text{running}}.$$

While communicating with service, *bindService*() can be called before *startService*(), but it is not recommended. The behaviors and state transitions are described as follows:

$$\overline{bindS}\langle m\rangle.\text{Component} \mid bindS\ (n)\ .\text{Service}^{\text{null}} \xrightarrow{bindS} \text{Component} \mid \{m/n\}\text{Service}^{\text{running}},$$
$$\overline{startS}\langle x\rangle.\text{Component} \mid startS\ (y)\ .\text{Service}^{\text{running}} \xrightarrow{startS} \text{Component} \mid clients\ (x)\ .\text{Service}^{\text{running}},$$
$$(onCreat.onBind.\text{Service}^{\text{running}}).onStartCommand.\ \text{Service}^{\text{running}}.$$

In order to stop a running service, regardless of the order in which *startSevice*() and *bindService*() were called, *unbindService*() and *stopService*() should be both called to ensure that both of *onUnbind*() and *onDestroy*() are executed. Moreover, *onUnbind*() should be executed before *onDestroy*(). The behaviors and state transitions of service are described as follows:

$$\overline{unbindS}\langle x\rangle.\text{Component} \mid unbindS\ (y)\ .\text{Service}^{\text{running}} \xrightarrow{unbindS} \text{Component} \mid \overline{client\ (x)}.\text{Service}^{\text{running}},$$
$$\overline{stopS}\langle m\rangle.\text{Component} \mid stopS\ (n)\ .\text{Service}^{\text{running}} \xrightarrow{stopS} \text{Component} \mid ([clients = \emptyset]\ \text{Service}^{\text{destroyed}} + [clients \neq \emptyset]\ \text{Service}^{\text{running}}),$$
$$(onUnbind.\text{Service}^{\text{running}}).onDestroy.\text{Service}^{\text{destroyed}}.$$

C. BROADCAST RECEIVER

Broadcast receiver is the receiver in Android broadcasts and is responsible for responding to the broadcasts of system and component. The behavior is defined as follows:

$$sendBroadcast\ (y)\ .\text{BroadcastReceiver}.$$

Each component of application can initiate a broadcast as a sender, where the broadcast is handled by *onReceive*() in the receiver. Broadcast has a short life cycle, which begins at creation and ends after successfully execution or termination due to timeout. The duration of the broadcast is represented as *timeCost*, and the timeout of the broadcast is represented as *timeOut*. Then, the behavior is described as follows:

$$\overline{sendBroadcast}\langle x\rangle.\text{Component} \mid sendBroadcast\ (y)\ .\ \text{BroadcastReceiver} \xrightarrow{sendBroadcast} \text{Component} \mid ([timeCost \leq timeOut]\ onReceive.\text{BroadcastReceiver} + [timeCost > timeOut]\ \underline{0}).$$

D. CONTENT PROVIDER

Content provider is used to store data and provide data sharing in applications. It can be accessed by external process if android:exported="true" is set in declaration. Components manipulate the data in content provider by using the methods of content resolver. The behaviors are defined as follows:

$$insertCP\ (u)\ .\text{ContentProvider},$$
$$deleteCP\ (u)\ .\text{ContentProvider},$$
$$updateCP\ (u)\ .\text{ContentProvider},$$
$$queryCP\ (u)\ .\text{ContentProvider}.$$

Components first create an instance of content resolver, and then call *insert*(), *delete*(), *update*(), *query*() with uri as

the parameter in the instance to manipulate data. The behavior of inserting data is described as follows:

$$(newr)(RegistContentResolver\ (r)\ .\overline{insertCP}\langle u\rangle.\ \text{Component}) \mid insertCP\ (u)\ .\text{ContentProvider} \xrightarrow{insert} \text{Component} \mid \text{ContentProvider}.$$

Action $RegistContentResolver\ (r)$ is the intra-action of caller component for creating instance r of content resolver, where r can call class methods to manipulate data. Therefore, the behavior can be simplified and described as follows:

$$(\overline{insertCP}\langle u\rangle.\text{Component}) \mid insertCP\ (u)\ .\text{ContentProvider} \xrightarrow{insert} \text{Component} \mid \text{ContentProvider}.$$

The processes of deleting, updating, and querying data are consistent with the process of inserting data, therefore there is no longer to repeat the description.

E. COMPONENT INTERACTION WITH DATABASE

The methods for data operations on database are predefined. In order to maintain the consistency of logic and formal descriptions, data operations are regarded as interactions and expressed as $\overline{op}\ (data)\ .\text{Component}|op\ (data)\ .\text{DataBase}$. The behaviors of data operations are defined as follows:

$$insertDB\ (data)\ .\text{DataBase},$$
$$deleteDB\ (data)\ .\text{DataBase},$$
$$queryDB\ (data)\ .\text{DataBase},$$
$$updateDB\ (data)\ .\text{DataBase}.$$

In this section, we describe the behavior of components using process algebra expression and achieve the behavior formalization of application. These expressions conform to the semantics and rules in this paper and have been validated using MWB. Since the validations are not the focus of study and require many pages, they are not included in this paper.

Based on behavior formalization, application behavior can be analyzed using the process equivalence mechanism of process algebra theory. In the following section, we define the simulation and mutual simulation of behavior to analyze the similarity between behaviors, and then propose decision rules based on weak simulation.

V. BEHAVIOR DECISION BASED ON BEHAVIOR EQUIVALENCE-WEAK SIMULATION

Expected behavior refers to the combination of queues and actions necessary to achieve application functions and meet user requirements. Application behaviors can be categorized as follows according to whether they are expected:

- **Credible behavior.** Behaviors can be monitored and identified, and are expected.
- **Malicious behavior.** Behaviors can be monitored and identified, and are unexpected.
- **Suspicious behavior.** Behaviors can be monitored, but can only be partially identified and cannot be identified as expected or unexpected.

An application is benign if all its behaviors are expected, which means that all behaviors are credible. An application is malicious if there are some unexpected behaviors in it, which means that it has at least one malicious behavior. Therefore, in Android malware detection, the focus should be whether there is malicious behavior in the application. Based on the relationship between application behavior and expected or unexpected behavior, application behavior could be analyzed according to the behavior equivalence mechanism.

Definition 7 (Behavior Equivalence): Let P and Q be two different processes, P and Q are trace equivalent if and only if traces(P)= traces(Q). The relationship between P and Q is process equivalence, also called behavior equivalence.

The term *process* describes the behavior of system by using the elements discussed in Section III. Processes are trace equivalent means that they have the same behavior pattern.

To describe the equivalence between behaviors, we extend the simulation theory of π-calculus and define the behavior simulation and behavior mutual simulation. In the following section, the two concepts are represented with simulation and mutual simulation for ease of description.

A. BEHAVIOR EQUIVALENCE: SIMULATION AND BISIMULATION

Simulation is an one-way description between behaviors. "P simulates Q" indicates that the behavior pattern of P is at least as rich as that of Q. Mutual simulation is a two-way simulation between behaviors, also known as bisimulation. "P and Q are mutual simulated" indicates that their behavior patterns are equivalent to some extent.

To accurately express the degree of similarity between behaviors, simulation is divided into strong simulation and weak simulation.

Definition 8 (Strong Simulation): Let P and Q be two behaviors, and let S be a binary relation. The relationship between P and Q is expressed as QSP. Then, we say that P strongly simulates Q if the following conditions hold:

(1) For each action a in Q and its transition $q \xrightarrow{a} q'$, where a is defined in formula (1), there exists action a and its transition in P such that $p \xrightarrow{a} p'$, where p' is a derived state of p.
(2) p' strongly simulates q'.

The binary relation S is called a strong simulation. QSP means that for any transition of Q, P has a path that contains all the actions of Q to match it.

Definition 9 (Strong Bisimulation): Let P and Q be two behaviors, and let S be a binary relation. The relationship between P and Q is expressed as QSP. Then, we say that P and Q are strongly bisimulated if the following conditions hold:

(1) For each action a in Q and its transition $q \xrightarrow{a} q'$, where a is defined in formula (1), there exists action a and its transition in P such that $p \xrightarrow{a} p'$, where p' is a derived state of p. Additionally, p' strongly simulates q' and q' strongly simulates p'.
(2) For each action b in P and its transition $p \xrightarrow{b} p'$, where b is defined in formula (1), there exists action b and its

transition in Q such that $q \xrightarrow{b} q'$, where q' is a derived state of q. Additionally, q' strongly simulates p' and p' strongly simulates q'.

The binary relation S is called a strong bisimulation. QSP means that P and Q are strongly equivalent, written as $P \sim Q$. From definition 9, it can be concluded that QSP is equivalent to QSP.

Kindly note that "P and Q are strongly equivalent" is not equal to "P strongly simulates Q, and Q strongly simulates P". The former is a stricter condition and includes the latter.

It is extremely difficult to use strong simulation to analyze the behavior of complex systems because considering all the actions and transitions of the system will bring an unbearable burden. However, the focus of behavior analysis of Android Apps should be on whether there is malicious behavior in the application, rather than whether the application behavior and malicious behavior are isomorphic or homomorphic. Android Apps are complex systems with a high degree of concurrency and interaction, so it is a feasible solution to ignore irrelevant actions to analyze the behavior of application. Therefore, we present the definition of weak simulation of behavior.

Based on the semantics of behavior proposed in Definition 4 and Definition 5, we use symbol e to represent a sequence of actions. It is a sequence of ordered actions, denoted as $e = \langle t_1, t_2, t_3, \ldots \rangle$, which can contain any type and any number of actions. $(new\ a)\ e$ means there is at least one action a in e. The execution of e is denoted as $\stackrel{e}{\Longrightarrow}$, where there can be any number of interactions.

Definition 10 (Weak Simulation): Let P and Q be two behaviors, and let S be a binary relation. The relationship between P and Q is expressed as QSP. Then, we say that P weakly simulates Q if the following conditions hold:

(1) For each action a in Q and its transition $q \xrightarrow{a} q'$, where a is defined in formula (1), there exists action a and $(new\ a)e$ in P such that $p \stackrel{(new\ a)e}{\Longrightarrow} p'$, where p' is a derived state of p.
(2) p' weakly simulates q'.

The binary relation S is called a weak simulation. QSP means that for any transition of Q, P has a path to cover it.

Definition 11 (Weak Bisimulation): Let P and Q be two behaviors, and let S be a binary relation. The relationship between P and Q is expressed as QSP. Then, we say that P and Q are weakly bisimulated if the following conditions hold:

(1) For each action a in Q and its transition $q \xrightarrow{a} q'$, where a is defined in formula (1), there exists action a and $(new\ a)e$ in P such that $p \stackrel{(new\ a)e}{\Longrightarrow} p'$. Additionally, p' weakly simulates q'.
(2) For each action b in P and its transition $p \xrightarrow{b} p'$, where b is defined in formula (1), there exists action b and $(new\ b)\ e$ in Q such that $q \stackrel{(new\ b)e}{\Longrightarrow} q'$. Additionally, q' weakly simulates p'.

The binary relation S is called a weak bisimulation. QSP means that P and Q are weakly equivalent, written as $P \approx Q$. Weak bisimulation is also called weak equivalence or observation equivalence.

The restrictions of strong bisimulation, strong simulation, weak bisimulation, and weak simulation are gradually reduced. Researchers can apply the corresponding simulation mechanism according to their actual situation.

In application behavior analysis, using strong simulation to analyze the equivalence between behaviors must consider all actions and trasitions. This will be an intolerable burden because it is extremely complicated. Fortunately, we have found that weak simulation is suitable and sufficient for studying suspicious and uncertain behavior. The next section proposes decision rules based on weak simulation according to behavior equivalence.

B. DECISION RULES BASED ON WEAK SIMULATION

Rule 1 (Credible Behavior): Behavior P is a credible behavior if expected behavior Q weakly simulates P.

The rule indicates that P is contained in expected behavior Q. The sequence of actions in P is an ordered subset of the sequence of actions in Q, where actions can be continuous or discontinuous. The sequence of actions in Q is credible, and so is its ordered subset. Therefore, P is credible.

Rule 2 (Malicious Behavior): Behavior P is a malicious behavior if P weakly simulates unexpected behavior R.

The rule indicates that unexpected behavior R is contained in P. The sequence of actions in R is an ordered subset of the sequence of actions in P, where actions can be continuous or discontinuous. Since the sequence of actions in R is malicious, there is at least one path of malicious behavior in P. Therefore, P is malicious.

*Rule 3 (Suspicious Behavior):*Behavior P is a suspicious behavior if P weakly simulates expected behavior Q, or unexpected behavior R weakly simulates P.

P weakly simulates expected behavior Q indicates that traces(P) contains part of the sequence of actions in Q. However, it is impossible to decide whether P is credible or malicious according to this condition alone, because there is no guarantee that all traces in P are credible. Therefore, P is suspicious and should be further analyzed.

Unexpected behavior R weakly simulates P indicates that traces(P) contains part or all of the sequence of actions in R. If traces(P) contains only part of R's actions, it does not necessarily constitute a malicious behavior. It is impossible to decide whether P is credible or malicious according to this condition alone. Therefore, P is suspicious and should be further analyzed.

Based on the relationship between weak simulation and weak bisimulation, the following inferences can be drawn from Rule 1 and Rule 2.

Inference 1 (Credible Behavior): P is credible if it weakly mutual simulates expected behavior Q.

Inference 2 (Malicious Behavior):P is malicious if it weakly mutual simulates unexpected behavior R.

To make decision of application behavior P, it is necessary to construct formal descriptions of expected and unexpected behaviors to form priori rules, and make decision according to these rules. The decision process is as follows:

(1) P is malicious if P has some malicious behaviors.
(2) P is credible if all the behaviors of P are credible.
(3) P is suspicious if it contains suspicious behaviors that cannot be determined as malicious or credible. It should be further analyzed according to the rules based on weak simulation and weak bisimulation.
(4) P is malicious if it weakly simulates unexpected behavior R, or it is weakly mutual simulated by R.
(5) P is credible if it weakly simulates expected behavior Q, or it is weakly mutual simulated by Q.

In this process, the decisions of suspicious behaviors are abstracted into new decision rules for expected or unexpected behaviors, and other subsequent behaviors can be directly determined by continuously improving rules. On the basis of behavior formalization, according to the proposed rules, application behavior can be classified and determined by the inference and calculation mechanism of process algebra.

According to Definition 8 and Definition 10, it can be inferred that P weakly simulates Q from the condition "P strongly simulates Q" or "P and Q are strongly equivalent". Therefore, the rules and inferences based on weak simulation also hold for strong simulation, as shown below:

- Behavior P is credible if expected behavior Q strongly simulates P, or P strongly mutual simulates Q.
- Behavior P is malicious if P strongly simulates or strongly mutual simulates unexpected behavior R.
- Behavior P is suspicious if P strongly simulates expected behavior Q, or unexpected behavior R strongly simulates P.

In the behavior analysis of Android Apps, using strong simulation to study equivalence is not only complex and difficult, but also flawed. For example, let Q be malicious behaviors, let P weakly simulates Q but does not strongly simulate Q. According to the decision rules based on weak simulation, it can be concluded that P is malicious. However, P cannot be analyzed by using strong simulation because there is no such relationship between P and Q. So P cannot be considered malicious when using the rules based on strong simulation. Therefore, weak simulation is more suitable than strong simulation in studying the equivalence between behaviors. Actully, if there is a strong simulation relationship between behaviors, which indicates that there is a higher equivalence than weak simulation, the same conclusion can be drawn as using weak simulation.

In the following section, we discuss a demonstration case derived from a real Android application containing malicious behavior to demonstrates the feasibility and effectiveness of the method in this paper.

VI. CASE ANALYSIS

Fig. 1 shows component interactions of the demonstration case. The default page is LogActivity and registered users can log into the page directly. New users should first register on RegActivity and then return to LogActivity to log into MainActivity, which provides functions such as information modification. Users register on RegActivity using a mobile

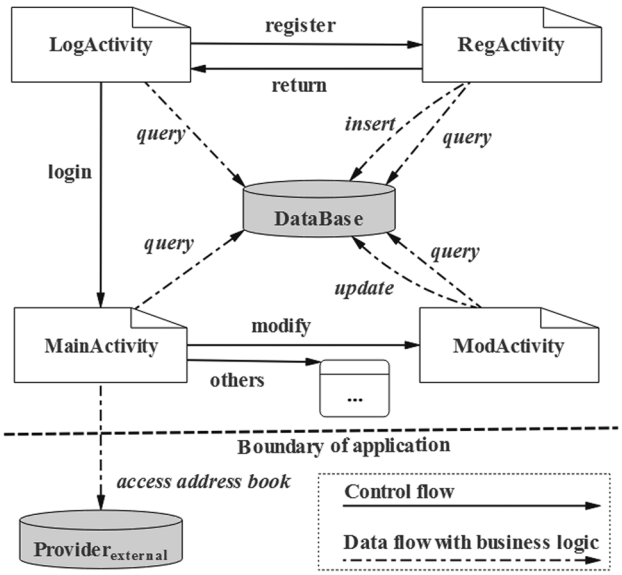


FIGURE 1. Schematic diagram of the interactions between components.

number and authorize the application to access the mobile phone address book and send SMS. This application can leak information through component interactions.

The process of analyzing application behavior using the process equivalence mechanism of process algebra is divided into two phases:

Phase 1: Establish the instances of application component and achieve the behavior formalization of application.

Phase 2: Analyze the relationship between application behavior and malicious behavior by using simulation mechanism, and then make the decision.

A. FORMAL DESCRIPTION OF COMPONENT BEHAVIOR AND COMPONENT INTERACTION

According to Fig. 1 and Table 2, we construct the component instances and illustrate there actions in Fig. 2.

In Fig. 2, *otheractions* represent reactions and $action_i$ represent observations, which are discussed in Definition 2. $intent_i$ are used to transmit information in component communicaion and are defined as follows:

$$\text{intent}_1 = \text{newIntent(LogActivity.this, RegActivity.class);}$$
$$\text{intent}_2 = \text{newIntent(LogActivity.this, ModActivity.class);}$$
$$\text{intent}_3 = \text{newIntent(MainActivity.this, ModActivity.class).}$$

There are four component instances that can initiate interactiions and act as the subject or object of interaction behavior, as shown in Fig. 2. $\text{Provider}_{\text{external}}$ is an external component outside application. It cannot initiate interactions, but only responds to the interaction of other components. The behaviors of components are defined as follows:

$$\overline{startActiviy}\langle regist\rangle.\text{LogActivity},$$
$$setResult\ (info)\ .\text{LogActivity},$$

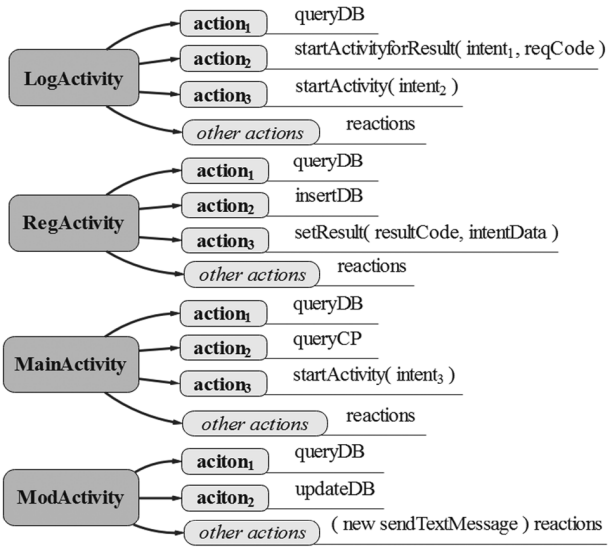


FIGURE 2. Behavior actions of component instances.

$$\overline{queryDB}\langle phone\rangle.\text{LogActivity},$$
$$\overline{startActiviy}\langle login\rangle.\text{LogActivity},$$
$$startActiviy\,(regist)\,.\text{RegActivity},$$
$$\overline{queryDB}\langle phone\rangle.\text{RegActivity},$$
$$\overline{insertDB}\langle newUser\rangle.\text{RegActivity},$$
$$\overline{setResult}\langle info\rangle.\text{RegActivity},$$
$$startActiviy\,(login)\,.\text{MainActivity},$$
$$\overline{queryDB}\langle phone\rangle.\text{MainActivity},$$
$$\overline{queryCP}\langle phoneBook\rangle.\text{MainActivity},$$
$$\overline{startActivity}\langle modify\rangle.\text{MainActivity},$$
$$startActiviy\,(modify)\,.\text{ModActivity},$$
$$\overline{queryDB}\langle phone\rangle.\text{ModActivity},$$
$$\overline{updateDB}\langle newInfo\rangle.\text{ModActivity},$$
$$queryCP\,(phoneBook)\,.\text{Provider}_{\text{external}}.$$

LogActivity can initiate the interaction with MainActivity and RegActivity according to user choice. Users should log in again if the authentication fails. User choice is represented as *choice*, and the result of authentication is represented as *isChecked*. The component behavior is described as follows:

$$[choice = regist]\,(\overline{startActiviy}\langle regist\rangle.$$
$$setResult\,(info)\,.\text{LogActivity}\,|startActivity\,(regist)\,.$$
$$\overline{setResult}\langle info\rangle.\text{RegActivity})+[choice = login]$$
$$((\overline{queryDB}\langle phone\rangle.\text{LogActivity}\mid queryDB\,(phone)\,.$$
$$\text{DataBase}).([isChecked = True](\overline{startActiviy}\langle login\rangle.$$
$$\text{LogActivity}\mid startActivity\,(login)\,.\text{MainActivity})$$
$$+[isChecked = Flase]\,!\text{LogActivity})).$$

RegActivity will be terminated if the new user cancels the registration. If the registration is successful, LogActivity will be active after saving user information to database, otherwise user should fill in registration information again. Use *isExsited* to indicate whether the user exists, the component behavior is described as follows:

$$[operate = Regist](\overline{queryDB}\langle phone\rangle.\text{RegActivity}\mid$$
$$queryDB\,(phone)\,.\text{DataBase}).([isExsited = False]$$
$$(\overline{insertDB}\langle newUser\rangle.\overline{setResult}\langle info\rangle.\text{RegActivity}\mid$$
$$insertDB\,(newUser)\,.\text{DataBase}\mid setResult\,(info)\,.$$
$$\text{LogActivity})+[isExsited = True]\,!\text{RegActivity})$$
$$+[operate = Cancel]\,finish.\text{RegActivity}.$$

MainActivity can display user information and provides functions such as user information modification. In addition, it can access the mobile phone address book. The component behavior is described as follows:

$$(\overline{queryDB}\langle phone\rangle).\text{MainActivity}\mid queryDB\,(phone)\,.$$
$$\text{DataBase}).(\overline{queryCP}\langle phoneBook\rangle.\text{MainActivity}\mid$$
$$queryCP\,(phoneBook)\,.\text{Provider}_{\text{external}}).$$
$$([choice = modify]\overline{startActivity}\langle modify\rangle.$$
$$\text{MainActivity}\mid startActiviy\,(modify)\,.\text{ModActivity}$$
$$+[choice = otherChoice]\,\text{Process}_{\text{others}}).$$

Users can modify personal information on ModActivity. If user confirms the modification, the new information will be updated to the database, otherwise ModActivity will be terminated. The component behavior is described as follows:

$$(\overline{queryDB}\langle phone\rangle.\text{ModActivity}\mid queryDB\,(phone)\,.$$
$$\text{DataBase}).([confirm = Yes]\,\overline{updateDB}\langle newInfo\rangle.$$
$$\text{ModActivity}\mid updateDB\,(newInfo)\,.\text{DataBase}$$
$$+[confirm = No]\,finish.\text{ModActivity})\mid$$
$$(\text{new}\ sendTextMessage)\,\text{ModActivity}.$$

The behavior of the case is composed of the concurrent execution of these formal expressions above. The results of validation by MWB show that these formal expressions can accurately describe the behavior of components and the interactions with other components.

B. BEHAVIOR ANALYSIS USING STRONG SIMULATION AND WEAK SIMULATION

As shown in Fig. 1, MainActivity has the behavior of accessing address book and ModActivity has the behavior of sending text message. Then, they form a path for information leakage through interaction and achieve a collusion attack, as shown in Fig. 3.

MainActivity implements collaborative behavior and obtains data from mobile address book through *queryCP*. It initiates communication through action *startActivity* and transmits data to ModActivity. Then, ModActivity leaks data through *sendTextMessage*. Without considering the intermediate states and interactions with other components, the

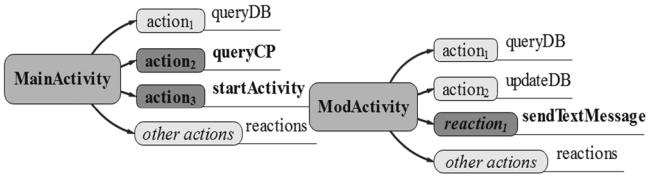


FIGURE 3. Collusion attack in components.

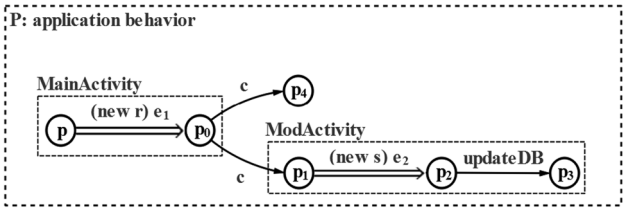


FIGURE 4. Schematic diagram of the collusion attack in the application.

collusion attack is described as follows:

$$(\overline{queryCP}\langle x\rangle\overline{startActivity}\langle i\rangle, \text{MainActivity} \mid startActiviy\,(i)\,.\,(new\ sendTextMessage)\,(e_2.queryDB.e_3.updateDB.\text{ModActivity}).$$

The symbol e_i is a sequence of actions, which can contain any number and any type of actions. We first represent $\overline{queryCP}\langle x\rangle.e_1$ as $(newquery\ CP)\,e_1$, and then represent $(new$sendTextMessage$)(e_2.queryDB.e_3.$ModActivity) as $(newsendTM)\,e_2.updateDB.$ModActivity. Therefore, the collusion attack containing harmful sequence of actions $\langle queryCP, sendTM\rangle$ can be described as follows:

$$(newquery\ CP)\,e_1.startActivity.\,(newsendTM)\,e_2.$$

The action of obtaining private data is represented as r, the action of initiating component interaction is represented as c, and the action of leaking data is represented as s. In MainActivity, r is the action *queryCP* that accesses the mobile address book. Using e_1 to represent the actions of obtaining data, the behavior of obtaining private data can be expressed as $p \xRightarrow{(new\ r)e_1} p_0$, and there is some private data in p_0. In ModActivity, s is the action *sendTM* that sends text message. Using e_2 to represent the actions of leaking data, the behavior of leaking data is expressed as $p_1 \xRightarrow{(new\ s)e_2} p_2$, and data in p_2 has been leaked. Therefore, the collusion attack implemented in the interaction between MainActivity and ModActivity is represented as P, as shown in Fig. 4.

The harmful sequence of actions $\langle queryCP, sendTM\rangle$ in application can be expressed as $\langle r, s\rangle$, which constitutes collusion attack Q with a series structure, as shown in Fig. 5.

P is the information leakage behavior in the demonstration case. Q is a typical collusion attack behavior in Android applications. Using strong simulation and weak simulation to analyze the relationship between P and Q, we can draw the following conclusions:

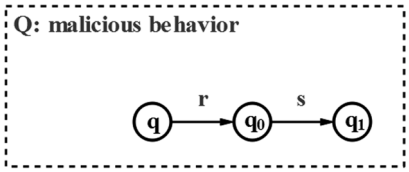


FIGURE 5. Collusion attack with series structure.

1) P weakly simulates Q.

- For $s \in Q$ and $q_0 \xrightarrow{s} q_1$, then $\exists p_0 \xrightarrow{c} p_1 \xRightarrow{(news)e_2} p_2$ in P, which can be abbreviated as $p_0 \xRightarrow{<c,(new\ s)e_2>} p_2$ according to Definition 4. Obviously, p_2 weakly simulates q_1 because there is no action in q_1. Therefore, p_0 weakly simulates q_0.
- For $r \in Q$ and $q \xrightarrow{r} q_0$, then $\exists p \xRightarrow{(new\ r)e_1} p_0$ in P. Since p_0 weakly simulates q_0 has been proven, p weakly simulates q.

According to Definition 10, we draw the conclusion that P weakly simulates Q.

2) Q cannot weakly simulate P.

- For $c \in$ P and $p_0 \xrightarrow{c} p_1$, there are no actions and transitions to match them in Q. According to Definition 10, we draw the conclusion that Q cannot weakly simulate P.

3) P cannot strongly simulate Q.

- For $r \in$ Q and $q \xrightarrow{r} q_0$, we first assume that there is a transition $p \xRightarrow{(new\ r)e_1} p_0$ in P. However, p_0 cannot strongly simulate q_0 because there are no actions and transitions to match $s \in$ Q and $q_0 \xrightarrow{s} q_1$. According to Definition 8, we draw the conclusion that P cannot strongly simulate Q.
- If the assumption is not valid, there are no actions and transitions to match $r \in$ Q and $q \xrightarrow{r} q_0$. According to Definition 8, we draw the same conclusion.

4) Q cannot strongly simulate P.

- It has been proven that Q cannot weakly simulate P in **2)**. According to the relationship between strong simulation and weak simulation, it is obvious that Q cannot strongly simulate P.

From the above discussion, it can be concluded that application behavior P weakly simulates malicious behavior Q, and P is malicious, but P cannot strongly simulate Q.

In the behavior analysis, even if the strong simulation relationship between P and Q is invalid, we can draw the conclusion that P is malicious based on the condition "P weakly simulates malicious behavior Q". However, behavior M cannot be considered malicious if malicious behavior Q weakly simulates M. For example, let Q be a malicious behavior containing transitions $q \xrightarrow{r} q_0 \xrightarrow{s} q_1$, M be a behavior containing only one transition $m \xrightarrow{r} m_0$. Obviously, Q weakly simulates behavior M, but M cannot be considered malicious because M only contains one action of obtaining data.

By discussing the behavior P in Fig. 3 and the behavior Q in Fig. 4, we have demonstrated that the weak simulation

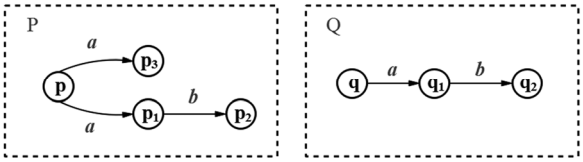


FIGURE 6. Schematic diagram of the behavior of P and Q.

relationship between P and Q can be used to make decision of application behavior. Unfortunately, there is no strong simulation relationship between P and Q. Thus, although strong simulation represents higher behavior equivalence, it sometimes has limitations and flaws in behavior analysis. Moreover, strong simulation requires more work than weak simulation. Therefore, weak simulation is more suitable than strong simulation in application behavior analysis.

In order to illustrate the application of strong simulation in behavior equivalence, we further abstract and simplify the actions in P and Q to generate two more general behaviors, which are still represented as P and Q, as shown in Fig. 6.

Using strong simulation to analyze the relationship between P and Q, we draw the following conclusions:

1) P strongly simulates Q.

- For $b \in Q$ and $q_1 \xrightarrow{b} q_2$, then $\exists p_1 \xrightarrow{b} p_2$ in P. It is obvious that p_2 strongly simulates q_2 because there is no action in q_2. Therefore, p_1 strongly simulates q_1.
- For $a \in Q$ and $q \xrightarrow{a} q_1$, then $\exists p \xrightarrow{a} p_1$ in P. Since p_1 strongly simulates q_1 has been proven, p strongly simulates q.

According to Definition 8, we draw the conclusion that P strongly simulates Q.

2) Q strongle simulates P.

- For $b \in P$ and $p_1 \xrightarrow{b} p_2$, then $\exists q_1 \xrightarrow{b} q_2$ in Q. It is obvious that q_2 strongly simulates p_2 because there is no action in p_2. Therefore, q_1 strongly simulates p_1.
- For $a \in P$ and $p \xrightarrow{a} p_1$, then $\exists q \xrightarrow{a} q_1$ in Q, and q_1 strongly simulates p_1 has been proven; for $a \in P$ and $p \xrightarrow{a} p_3$, then $\exists q \xrightarrow{a} q_1$ in Q, and q_1 strongly simulates p_3 because there is no action in p_3. Therefore, q strongly simulates p.

According to Definition 8, we draw the conclusion that Q strongly simulates P.

3) P and Q are not strongly equivalent.

- For $b \in P$ and $p_1 \xrightarrow{b} p_2$, then $\exists q_1 \xrightarrow{b} q_2$ in Q. Since p_1 strongly simulates q_1 and p_2 strongly simulates q_2 were proven in **1)**; q_1 strongly simulates p_1 and q_2 strongly simulates p_2 were proven in **2)**. Therefore, q_1 and p_1 are strongly equivalent, denoted as $q_1 \sim p_1$.
- P has two branches at action a. Consider the bottom branch alone, for $a \in P$ and $p \xrightarrow{a} p_1$, then $\exists q \xrightarrow{a} q_1$, and $q_1 \sim p_1$ is proven, such that $q \sim p$. However, for $a \in P$ and $p \xrightarrow{a} p_1$ in the top branch, then $\exists q \xrightarrow{a} q_3$. Since q_1 strongly simulates p_3 but p_3 cannot strongly simulate q_1, p and q are not strongly equivalent.

According to Definition 9, we draw the conclusion that P and Q are not strongly equivalent.

From the discussion above, we concluded that P and Q in Fig. 6 are not strongly equivalent, but P strongly simulates Q and Q strongly simulates P. The proof of these conclusions shows that ''P strongly simulates Q and Q strongly simulates P'' is not equal to ''P and Q are strongly equivalent''. In fact, according to Definition 8 and Definition 10, it can be inferred that P weakly simulates Q and Q weakly simulates P. Therefore, both P and Q are malicious or credible according to the rules based on weak simulation.

In this section, we achieve the formal description of the behavior of demonstration application, and then use strong simulation and weak simulation to analyze the equivalence between application behavior and malicious behavior. The results show that the method for the description and decision of application behavior can accurately and effectively modle and analyze application behavior. Furthermore, the analysis process prove that weak simulation is suitable compared with strong simulation and the rules and inferences based on weak simulation are sufficient and effective for the analysis and decision of application behavior.

VII. CONCLUSION

In this study, we proposed a formal method for studying application behavior, which uses process algebra expressions and operators to achieve behavior formalization of an application, and use the inference and calculation mechanism to achieve the analysis and decision of application behavior. In view of the concurrency and interaction characteristics of Android Apps, we extended the π-calculus theory, which is suitable for analyzing mobile concurrent interaction systems, to the study of application behavior. Based on this study's semantics and rules, the behavior of four types of application components was described using process algebra expressions. Further, we presented definitions for analyzing behavior equivalence and proposed decision rules based on weak simulation by discussing the application of strong simulation and weak simulation in behavior analysis. The formal method for the description and decision of application behavior was validated in case analysis. This study will help scholars understand the relationships and laws of application behavior with collaborations and interactions, and it will provide theoretical support for the analysis and detection of application behavior based on various dynamic and static behavior features.

In future research, we will focus on the construction of behavior rules and the simulation detection of behavior to conduct automatic analysis and decision of application behavior. In this study, we have proved that the rules based on weak simulation are effective and sufficient in behavior analysis, and weak simulation is more suitable than strong simulation. However, strong simulation represents a higher degree of similarity in behavior equivalence and the rules based on weak simulation also hold for strong simulation. Therefore, combining inter-action and intra-action to analyze

application behavior using strong simulation is a potential research field.

REFERENCES

[1] *Share of Smartphone Shipments Worldwide by Operating System From 2014 to 2023, 2021*. Accessed: Mar. 12, 2021. [Online]. Available: https://www.statista.com/ statistics/277048/global-market-share-forecast-of-smartphone-perating-systems/

[2] (2021). *Smartphone Market Share*. Accessed: Mar. 26, 2021. [Online]. Available: https://www. idc.com/promo/smartphone-market-share

[3] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 1–41, Dec. 2017, doi: 10.1145/3017427.

[4] F. Farivar, M. S. Haghighi, A. Jolfaei, and M. Alazab, "Artificial intelligence for detection, estimation, and compensation of malicious attacks in nonlinear cyber-physical systems and industrial IoT," *IEEE Trans. Ind. Informat.*, vol. 16, no. 4, pp. 2716–2725, Apr. 2020, doi: 10.1109/TII.2019.2956474.

[5] T. Sharma and D. Rattan, "Malicious application detection in Android—A systematic literature review," *Comput. Sci. Rev.*, vol. 40, May 2021, Art. no. 100373, doi: 10.1016/j.cosrev.2021.100373.

[6] S. Hr, "Static analysis of Android malware detection using deep learning," in *Proc. Int. Conf. Intell. Comput. Control Syst. (ICCS)*, May 2019, pp. 841–845, doi: 10.1109/ICCS45141.2019.9065765.

[7] D. O. Sahin, S. Akleylek, and E. Kilic, "LinRegDroid: Detection of Android malware using multiple linear regression models-based classifiers," *IEEE Access*, vol. 10, pp. 14246–14259, 2022, doi: 10.1109/ACCESS.2022.3146363.

[8] L. Pan, B. Cui, J. Yan, X. Ma, J. Yan, and J. Zhang, "Androlic: An extensible flow, context, object, field, and path-sensitive static analysis framework for Android," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2019, pp. 394–397, doi: 10.1145/3293882.3339001.

[9] J. Zhang, X. Zhuang, and Y. Chen, "Android malware detection combined with static and dynamic analysis," in *Proc. the 9th Int. Conf. Commun. Netw. Secur.*, Nov. 2019, pp. 6–10, doi: 10.1145/3371676.3371685.

[10] J. Gajrani, V. Laxmi, M. Tripathi, M. S. Gaur, A. Zemmari, M. Mosbah, and M. Conti, "Effectiveness of state-of-the-art dynamic analysis techniques in identifying diverse Android malware and future enhancements," *Adv. Comput.*, vol. 119, pp. 73–120, Jan. 2020, doi: 10.1016/bs.adcom.2020.03.002.

[11] H. Alshahrani, H. Mansourt, S. Thorn, A. Alshehri, A. Alzahrani, and H. Fu, "DDefender: Android application threat detection using static and dynamic analysis," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2018, pp. 1–6, doi: 10.1109/ICCE.2018.8326293.

[12] M. Choudhary and B. Kishore, "HAAMD: Hybrid analysis for Android malware detection," in *Proc. Int. Conf. Comput. Commun. Informat. (ICCCI)*, Jan. 2018, pp. 1–4, doi: 10.1109/ICCCI.2018.8441295.

[13] J. Tang, R. Li, K. Wang, X. Gu, and Z. Xu, "A novel hybrid method to analyze security vulnerabilities in Android applications," *Tsinghua Sci. Technol.*, vol. 25, no. 5, pp. 589–603, Oct. 2020, doi: 10.26599/TST.2019.9010067.

[14] W. Wang, C. Ren, H. Song, S. Zhang, and P. Liu, "FGL_Droid: An efficient Android malware detection method based on hybrid analysis," *Secur. Commun. Netw.*, vol. 2022, pp. 1–11, Apr. 2022, doi: 10.1155/2022/8398591.

[15] P. Mishra, V. Varadharajan, U. Tupakula, and E. S. Pilli, "A detailed investigation and analysis of using machine learning techniques for intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 686–728, 1st Quart., 2019, doi: 10.1109/COMST.2018.2847722.

[16] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of Android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124579–124607, 2020, doi: 10.1109/ACCESS.2020.3006143.

[17] B. Li, Y. Zhang, J. Yao, and T. Yin, "MDBA: Detecting malware based on bytes n-gram with association mining," in *Proc. 26th Int. Conf. Telecommun. (ICT)*, Apr. 2019, pp. 227–232, doi: 10.1109/ICT.2019.8798828.

[18] R. Vinayakumar, K. P. Soman, and P. Poornachandran, "Deep Android malware detection and classification," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2017, pp. 1677–1683, doi: 10.1109/ICACCI.2017.8126084.

[19] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," *Digit. Invest.*, vol. 24, pp. S48–S59, Mar. 2018, doi: 10.1016/j.diin.2018.01.007.

[20] C. Zhang, Q. Zhou, Y. Huang, K. Tang, H. Gui, and F. Liu, "Automatic detection of Android malware via hybrid graph neural network," *Wireless Commun. Mobile Comput.*, vol. 2022, pp. 1–11, May 2022, doi: 10.1155/2022/7245403.

[21] I. Almomani, A. Alkhayer, and W. El-Shafai, "An automated vision-based deep learning model for efficient detection of Android malware attacks," *IEEE Access*, vol. 10, pp. 2700–2720, 2022, doi: 10.1109/ACCESS.2022.3140341.

[22] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, "Familial clustering for weakly-labeled Android malware using hybrid representation learning," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 3401–3414, 2020, doi: 10.1109/TIFS.2019.2947861.

[23] R. Kumar, X. Zhang, W. Wang, R. U. Khan, J. Kumar, and A. Sharif, "A multimodal malware detection technique for Android IoT devices using various features," *IEEE Access*, vol. 7, pp. 64411–64430, 2019, doi: 10.1109/ACCESS.2019.2916886.

[24] N. Xie, X. Wang, W. Wang, and J. Liu, "Fingerprinting Android malware families," *Frontiers Comput. Sci.*, vol. 13, no. 3, pp. 637–646, 2019, doi: 10.1007/s11704-017-6493-y.

[25] H. Zhang, S. Luo, Y. Zhang, and L. Pan, "An efficient Android malware detection system based on method-level behavioral semantic analysis," *IEEE Access*, vol. 7, pp. 69246–69256, 2019, doi: 10.1109/ACCESS.2019.2919796.

[26] Y.-T. Huang, Y. S. Sun, and M. C. Chen, "TagSeq: Malicious behavior discovery using dynamic analysis," *PLoS ONE*, vol. 17, no. 5, May 2022, Art. no. e0263644, doi: 10.1371/journal.pone.0263644.

[27] A. Arora, S. K. Peddoju, and M. Conti, "PermPair: Android malware detection using permission pairs," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 1968–1982, 2020, doi: 10.1109/TIFS.2019.2950134.

[28] J. Xiao, K. Xu, and J. Duan, "Malicious Android application detection based on composite features," in *Proc. 3rd Int. Conf. Comput. Sci. Appl. Eng. (CSAE)*, 2019, pp. 1–6, doi: 10.1145/3331453.3361664.

[29] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018. doi: 10.1109/TDSC.2016.2536605.

[30] A. Mahindru and P. Singh, "Dynamic permissions based Android malware detection using machine learning techniques," in *Proc. 10th Innov. Softw. Eng. Conf.*, Feb. 2017, pp. 202–210, doi: 10.1145/3021460.3021485.

[31] S. Y. Yerima and S. Khan, "Longitudinal performance analysis of machine learning based Android malware detectors," in *Proc. Int. Conf. Cyber Security and Protection of Digital Services (Cyber Security)*, Jun. 2019, pp. 1–8, doi: 10.1109/CyberSecPODS.2019.8885384.

[32] X. Wang, L. Zhang, K. Zhao, X. Ding, and M. Yu, "MFDroid: A stacking ensemble learning framework for Android malware detection," *Sensors*, vol. 22, no. 7, p. 2597, Mar. 2022, doi: 10.3390/s22072597.

[33] A. Fatima, R. Maurya, M. K. Dutta, R. Burget, and J. Masek, "Android malware detection using genetic algorithm based optimized feature selection and machine learning," in *Proc. 42nd Int. Conf. Telecommun. Signal Process. (TSP)*, Jul. 2019, pp. 220–223, doi: 10.1109/TSP.2019.8769039.

[34] J. Jung, H. Kim, D. Shin, M. Lee, H. Lee, S.-J. Cho, and K. Suh, "Android malware detection based on useful API calls and machine learning," in *Proc. IEEE 1st Int. Conf. Artif. Intell. Knowl. Eng. (AIKE)*, Sep. 2018, pp. 175–178, doi: 10.1109/AIKE.2018.00041.

[35] A. S. Shatnawi, A. Jaradat, T. B. Yaseen, E. Taqieddin, M. Al-Ayyoub, and D. Mustafa, "An Android malware detection leveraging machine learning," *Wireless Commun. Mobile Comput.*, vol. 2022, pp. 1–12, May 2022, doi: 10.1155/2022/1830201.

[36] R. S. Arslan, İ. A. Doğru, and N. Barişçi, "Permission-based malware detection system for Android using machine learning techniques," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 1, pp. 43–61, Jan. 2019, doi: 10.1142/S0218194019500037.

[37] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson, "Control flow analysis for the pi-calculus," *Proc. CONCUR Concurrency Theory*, 1998, pp. 84–98, doi: 10.1007/BFb0055617.

[38] A. Chaudhuri, "Language-based security on Android," in *Proc. ACM SIGPLAN 4th Workshop Program. Lang. Anal. Secur.*, 2009, pp. 1–7, doi: 10.1145/1554339.1554341.

[39] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-time enforcement of information-flow properties on Android," in *Proc. 18th Eur. Symp. Res. Comput. Secur.*, vol. 8134, 2013, pp. 775–792, doi: 10.1007/978-3-642-40203-6_43.

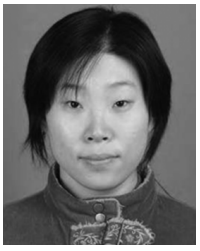
[40] L. Shen, H. Li, H. Wang, and Y. Wang, "Multifeature-based behavior of privilege escalation attack detection method for Android applications," *Mobile Inf. Syst.*, vol. 2020, pp. 1–16, Jun. 2020, doi: 10.1155/2020/3407437.
[41] S. Bae, S. Lee, and S. Ryu, "Towards understanding and reasoning about Android interoperations," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 223–233, doi: 10.1109/ICSE.2019.00038.
[42] A. Cimitile, F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone, "Formal methods meet mobile code obfuscation identification of code reordering technique," in *Proc. IEEE 26th Int. Conf. Enabling Technol., Infrastructure Collaborative Enterprises (WETICE)*, Jun. 2017, pp. 263–268, doi: 10.1109/WETICE.2017.23.
[43] X. He, "Modeling and analyzing the Android permission framework using high level Petri nets," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2017, pp. 232–239, doi: 10.1109/QRS.2017.34.
[44] G. Betarte, J. Campo, M. Cristia, F. Gorostiaga, C. Luna, and C. Sanz, "Towards formal model-based analysis and testing of Android's security mechanisms," in *Proc. 43rd Latin Amer. Comput. Conf. (CLEI)*, Sep. 2017, pp. 1–10, doi: 10.1109/CLEI.2017.8226404.
[45] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "A formal approach for detection of security flaws in the Android permission system," *Formal Aspects Comput.*, vol. 30, no. 5, pp. 525–544, Sep. 2018, doi: 10.1007/s00165-017-0445-z.
[46] W. Khan, M. Kamran, A. Ahmad, F. A. Khan, and A. Derhab, "Formal analysis of language-based Android security using theorem proving approach," *IEEE Access*, vol. 7, pp. 16550–16560, 2019, doi: 10.1109/ACCESS.2019.2895261.
[47] N. Z. Almuzaini and I. Ahmad, "Formal analysis of the signal protocol using the scyther tool," in *Proc. 2nd Int. Conf. Comput. Appl. Inf. Secur. (ICCAIS)*, May 2019, pp. 1–6, doi: 10.1109/CAIS.2019.8769532.
[48] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *Computer Aided Verification*, vol. 8044. Berlin, Germany: Springer, 2013, pp. 696–701, doi: 10.1007/978-3-642-39799-8_48.
[49] B. Khadiranaikar, P. Zavarsky, and Y. Malik, "Improving Android application security for intent based attacks," in *Proc. 8th IEEE Annu. Inf. Technol., Electron. Mobile Commun. Conf. (IEMCON)*, Oct. 2017, pp. 62–67, doi: 10.1109/IEMCON.2017.8117149.
[50] M. W. Afridi, T. Ali, T. Alghamdi, T. Ali, and M. Yasar, "Android application behavioral analysis through intent monitoring," in *Proc. 6th Int. Symp. Digit. Forensics Secur. (ISDFS)*, Mar. 2018, pp. 1–8, doi: 10.1109/ISDFS.2018.8355359.
[51] A. Tiwari, S. Gross, and C. Hammer, "IIFA: Modular inter-app intent information flow analysis of Android applications," in *Security and Privacy in Communication Networks*, vol. 305. Cham, Switzerland: Springer, 2019, pp. 335–349, doi: 10.1007/978-3-030-37231-6_19.
[52] H. Yang and J. Xu, "Android malware detection based on improved random forest," *J. Commun.*, vol. 38, no. 4, pp. 8–16, 2017, doi: 10.11959/j.issn.1000-436x.2017073.
[53] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "AndroDialysis: Analysis of Android intent effectiveness in malware detection," *Comput. Secur.*, vol. 65, pp. 121–134, Mar. 2017, doi: 10.1016/j.cose.2016.11.007.
[54] B. Kim, J. Jung, S. Han, S. Jeon, S.-J. Cho, and J. Choi, "A new technique for detecting Android app clones using implicit intent and method information," in *Proc. 11th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Jul. 2019, pp. 478–483, doi: 10.1109/ICUFN.2019.8806121.
[55] H. Bekic, "Towards a mathematical theory of processes," in *Programming Languages and Their Definition* (Lecture Notes in Computer Science), vol. 177. Berlin, Germany: Springer, 1984, doi: 10.1007/BFb0048944.
[56] J. A. Bergstra and J. W. Klop, "Fixed point semantics in process algebra," Math. Centre, Amsterdam, The Netherlands, Tech. Rep. N8318363, 1982, pp. 1–24. [Online]. Available: https://ntrl.ntis.gov/NTRL/
[57] J. C. M. Baeten, "A brief history of process algebra," *Theor. Comput. Sci.*, vol. 335, no. 2, pp. 131–146, 2005, doi: 10.1016/j.tcs.2004.07.036.
[58] R. Milner, *A Calculus of Communicating Systems*. Berlin, Germany: Springer, 1980.
[59] R. Milner, *Communicating and Mobile Systems: The Π-Calculus*. Cambridge, U.K.: Cambridge Univ. Press, 1999.
[60] A. C. Esterline and T. Rorie, "Using the Π-calculus to model multiagent systems," in *Proc. Int. Workshop Formal Approaches Agent-Based Syst.*, 2000, pp. 164–179, doi: 10.1007/3-540-45484-5_14.

DONGKUI LIANG was born in Hebei, China. He received the B.S. degree in information and computing science and the M.S. degree in computer software and theory from Yanshan University, Qinhuangdao, China, in 2003 and 2010, respectively, where he is currently pursuing the Ph.D. degree in computer application science. He worked at the School of Information Science and Engineering, Yanshan University, from 2003 to 2020. He is also working at the Engineering Training Center, Yanshan University. His current research interests include flexible software technology, information security, and software formal methods.

LIMIN SHEN (Member, IEEE) received the M.S. degree in computer application from the Hefei University of Technology, China, in 1987, and the Ph.D. degree in electronic circuit and system from Yanshan University, China, in 2005. He worked at the Department of Computer Science, Illinois Institute of Technology, USA, from 2005 to 2007, as a Visiting Scholar. He is currently a Professor and a Ph.D. Supervisor at the School of Information Science and Engineering, Yanshan University. His main research interests include flexible software technology, information security, service computing, and cooperative defense.

ZHEN CHEN received the B.S. and Ph.D. degrees in computer science and technology from Yanshan University, China, in 2010 and 2017, respectively. He is currently an Associate Professor at the School of Information Science and Engineering, Yanshan University. He is also working on service computing, cloud computing, and collaborative computing.

CHUAN MA was born in Hebei, China. He received the B.S. degree in information and computing science and the M.S. degree in computer application science from Yanshan University, Qinhuangdao, China, in 2003 and 2009, respectively, and the Ph.D. degree from the School of Information Science and Engineering, Yanshan University, in 2017. He worked at the School of Information Science and Engineering, Yanshan University, from 2003 to 2020. He is currently an Associate Professor at the Engineering Training Center, Yanshan University. His main research interests include information security and software formal methods.



JIAYIN FENG was born in Hebei, China. She received the B.S. degree in computer science and the M.S. degree in computer application science from Yanshan University, Hebei, in 2005 and 2008, respectively, where she is currently pursuing the Ph.D. degree. She has more than ten years of teaching experience at the Computer Science Department, Hebei Normal University of Science and Technology. Her current research interests include mobile network security, deep learning, and information security.