Software Quality Journal (2024) 32:1175–1202
https://doi.org/10.1007/s11219-024-09684-2

RESEARCH

Enhancement and formal verification of the ICC mechanism with a sandbox approach in android system

Jiaqi Yin[1,3] · Sini Chen[2] · Yixiao Lv[2] · Huibiao Zhu[2]

Accepted: 30 May 2024 / Published online: 27 June 2024

Abstract
Inter-Component Communication (ICC) plays a crucial role in facilitating information exchange and functionality integration within the complex ecosystem of Android systems. However, the security and safety implications arising from ICC interactions pose significant challenges. This paper is an extended work building upon our previously published research that focuses on the verification of safety properties in the ICC mechanism. We address the previously observed issues of data leakage and privilege escalation by incorporating a sandbox mechanism and permission control. The sandbox mechanism provides an isolated and controlled environment in which ICC components can operate while permission control mechanisms are introduced to enforce fine-grained access controls, ensuring that only authorized entities have access to sensitive resources. We further leverage formal methods, specifically communicating sequential processes (CSP), to verify several properties of the enhanced ICC mechanism. By employing CSP, we aim to systematically model and analyze the flow of information, the behavior of components, and the potential vulnerabilities associated with the enhanced ICC mechanism. The verification results highlight the effectiveness of our approach in enhancing the security and reliability of ICC mechanisms, ultimately contributing to the development of safer and more trustworthy Android Systems.

Keywords Android · Inter-Component Communication (ICC) · Inter-App Communication (IAC) · Communicating Sequential Process (CSP) · PAT with C#

1 Introduction

The Inter-Component Communication (ICC) mechanism is a fundamental aspect of Android systems, facilitating communication and collaboration between different components within an application or across multiple applications. ICC enables the seamless

✉ Sini Chen
52265902002@stu.ecnu.edu.cn

✉ Huibiao Zhu
hbzhu@sei.ecnu.edu.cn

1 School of Software, Northwestern Polytechnical University, Xi'an, China

2 Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China

3 Yangtze River Delta Research Institute of NPU, Taicang, China

exchange of information, commands, and resources, thereby facilitating the integration and functionality of diverse app components (Samhi et al., 2021).

The security of ICC holds paramount importance, attributable to the inherent risks associated with unauthorized or malicious communication pathways (Bhandari et al., 2017). Improper implementation (Gadient et al., 2019) or insufficient security measures in ICC may expose vulnerabilities, leading to a range of security threats such as information disclosure, privilege escalation, and remote code execution (Biswas et al., 2018; Bugiel et al., 2012; Zhou et al., 2013). Exploitation of such weaknesses could grant attackers unauthorized access to sensitive data, enable manipulation of application behavior, or compromise the overall security posture of the system.

Building upon our previous research (Lv et al., 2023) using process algebra Communicating Sequential Processes (CSP), we identified significant security vulnerabilities inherent within the existing ICC mechanism, including data leakage and privilege escalation. Our previous work served as a foundation for this research, inspiring us to explore solutions that fortify the security posture of Android systems. Subsequently, a comprehensive analysis was conducted on the trace output from the model checker PAT, and we identified several primary reasons contributing to security vulnerabilities within the ICC mechanism. Henceforth, our verification in this paper aims to address its significance in enhancing security within the Android system. By verifying the improved ICC mechanism design, we aim to bolster the overall security posture of Android applications by mitigating risks associated with inter-component communication vulnerabilities.

First, the simplicity of the intent matching mechanism, which solely examines the Action and Category fields using a limited set of constants provided by the Android API, enables malicious entities to make educated guesses. Furthermore, the receiving process is overly straightforward, as the Android ecosystem does not require that an Intent must carry the sender's information. Consequently, recipients handle and process Intents immediately upon receipt, creating an opportunity for malware exploitation.

To this end, we propose an enhanced model that leverages a sandbox mechanism. This mechanism encapsulates ICC interactions within isolated containers, effectively partitioning components and limiting their access to sensitive resources (Neuner et al., 2014). By establishing these boundaries, we aim to mitigate the risk of unauthorized data leakages and curb the propagation of potential security threats across the system. Additionally, our extended architecture incorporates privilege control (Fang et al., 2014), enabling a fine-grained permission management framework, as is illustrated in Fig. 1. By implementing this mechanism, we provide administrators and application

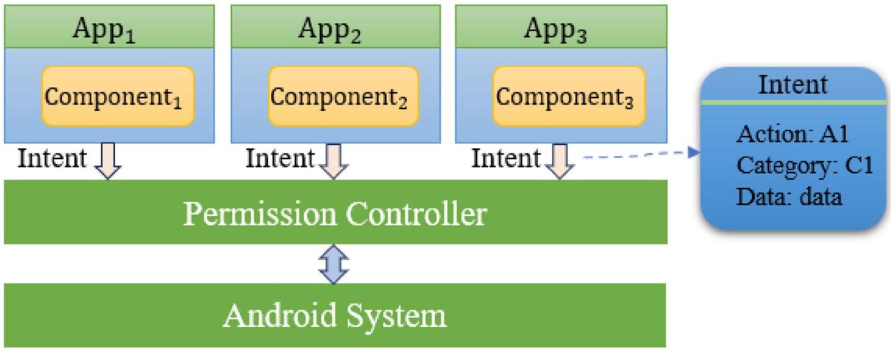


Fig. 1 The integration of permission control

developers with greater control over component access rights, ensuring that only authorized components can invoke critical functionalities and reducing the attack surface for potential exploits.

In line with our methodology, we continue to employ CSP as a formal modeling and verification technique for our extended ICC architecture. This enables us to rigorously assess the security properties of our proposed model, ensuring its soundness and resilience against potential threats.

By combining the advantages of the sandbox approach, privilege control, and formal verification through CSP, we demonstrate the potential to significantly enhance the security posture of the Android system's ICC mechanism. Our work contributes to a more robust and trustworthy mobile platform, empowering users, developers, and administrators alike to operate within a safer ecosystem.

In this paper, our primary extensions and contributions are as follows:

1. Proposing a simple yet effective sandbox approach to address data access security concerns, wherein different access permissions are allocated based on distinct data types.
2. Modelling and verifying the ICC mechanism integrated with the sandbox approach through introducing the relevant security properties to ensure the security of its data.
3. Providing technical guidance for security issues related to permission control and associated mechanisms.

The structure of this paper is organized as follows. Section 2 presents a brief introduction to the ICC mechanism in Android System, the process algebra CSP, and the verification tool PAT. In Section 3, we illustrate the modeling process using CSP. Section 4 shows the implementation of the model using PAT and verification of the properties. Finally, Section 5 concludes the paper and determines the goal of future work.

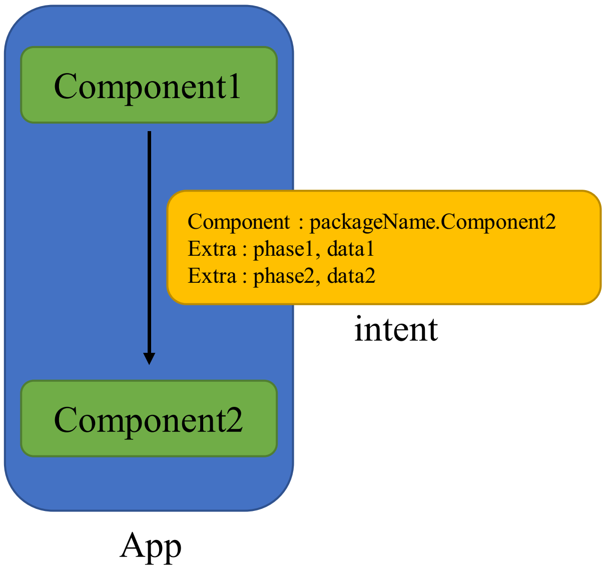
2 Background

In this section, we provide a brief introduction to ICC, IAC mechanism, permission granting mechanism, CSP, PAT, and the sandbox approach.

2.1 ICC and IAC mechanism

– ICC Mechanism: The ICC mechanism in Android allows components within an app or different apps to communicate with each other. It enables the exchange of data and messages, promoting interaction and collaboration. In Android, ICC comprises two types: Intra-app communication and Inter-app communication, as depicted in Figs. 2 and 3 respectively.
– IAC Mechanism: The IAC, or Inter-App Communication, refers specifically to communication between different apps running on an Android device. Android provides various mechanisms for IAC, including Content Providers, Broadcast Receivers, and Shared Preferences (Chin et al., 2011).

Fig. 2 The Communication Architecture of Intra-App Communication



Our research focuses on Intent-based communication, a high-level and loosely coupled mechanism utilized for both inter-component and inter-app communication. Intents (Developers, 2023) in Android are message objects that describe actions to be performed or data to be exchanged between components or apps. They enable one component to request an action from another component or transfer data. Intents facilitate actions such as starting activities, launching services, and broadcasting events. Additionally, Intents can carry data or parameters to be consumed by the receiving component (DiMarzio, 2008).

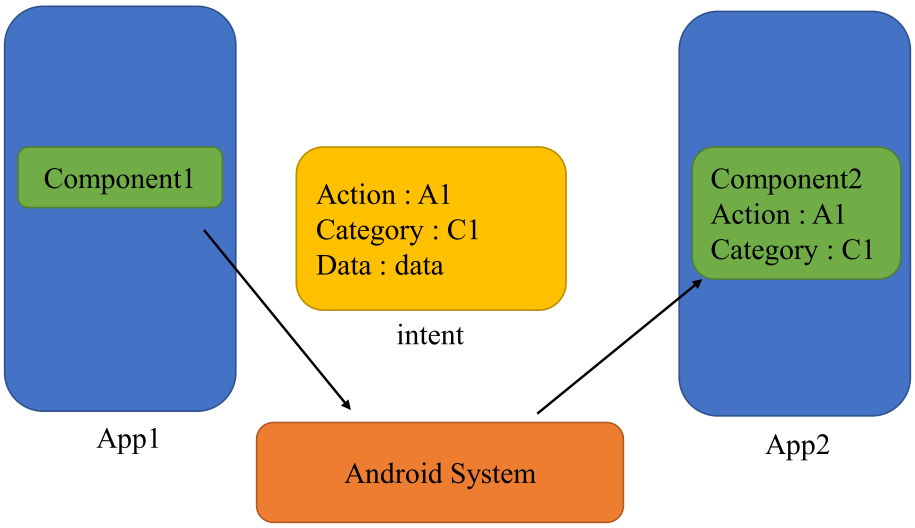


Fig. 3 The Communication Architecture of Inter-App Communication

Overall, the ICC and IAC mechanisms in Android play pivotal roles in enabling seamless communication and collaboration between app components as well as different apps. The choice of mechanism depends on specific needs and requirements, determining the level of interaction between components within apps or across apps. These mechanisms facilitate the development of complex and interconnected systems, leading to rich and interactive user experiences.

2.2 Permission granting mechanism

The permission granting mechanism functions as a user-centric control system, enabling users to determine which permissions an application can access on their devices. Permissions are access rights to specific resources or functionalities, such as accessing device sensors, reading contacts, or using the camera. By granting or denying permissions, users have the ability to protect their sensitive information and ensure that applications function within desired boundaries (Au et al., 2012). The permission granting mechanism comprises several steps, inlucding:

- Permission Request: When an Android application requires access to a sensitive resource or functionality that requires a permission, it must declare the necessary permissions in its manifest file. For example, if an application needs access to the device's camera, it must include the CAMERA permission in its manifest.
- Permission Check: At runtime, when the application attempts to access the requested resource or functionality, the Android system checks if the necessary permission has been granted by the user.
- User Consent: The permission prompt displayed to the user includes information about the permission being requested and the context in which it will be used. The user has the discretion to either grant the permission or deny it based on their judgment and trust in the application.
- Permission Grant: If the user grants the permission, the Android system marks it as granted for the specific application. Subsequently, the application can freely access the requested resource or functionality during its runtime. In this paper, we referred to the permission control method proposed by Almomani and Al Khayer (2020) and combined it with our original ICC architecture, as shown in Fig. 4.

2.3 CSP and PAT

The process algebra is a formal method that studies the communication between concurrent processes. In this paper, we use Communicating Sequential Process (CSP), proposed by Turing Award winner Hoare (1985) to model the ICC mechanism in Android System. It is a kind of process algebra, which has been successfully applied to verify a lot of parallel systems (Lowe & Roscoe, 1997; Xu et al., 2021).

The syntax and definitions of CSP statements used in this paper are presented in Table 1, where *a* represents an atomic action or event, *b* denotes a Boolean expression, *c* stands for a channel, *P* and *Q* represent the processes.

PAT is a model checker suitable for analyzing and verifying various protocols and systems. It can check the deadlock freedom, reachability, and Linear temporal logic (LTL) properties of the system, by automatically traversing the state of the system and providing a counterexample path when the system does not meet the properties.

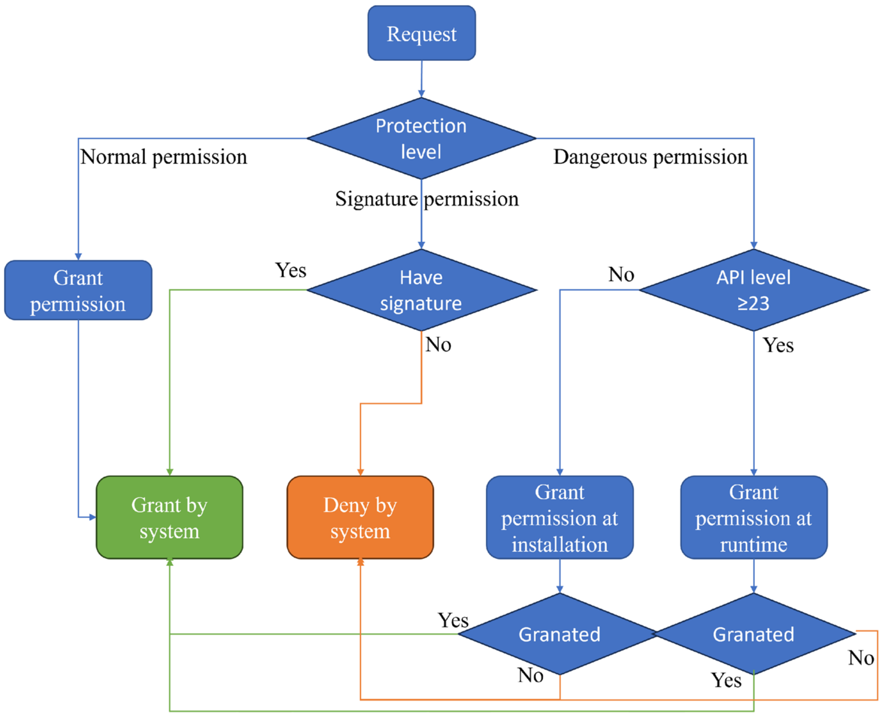


Fig. 4 The permission granting based on the protection level

Table 1 The syntax and its corresponding explanations used in this paper

Syntax	Explanations
$SKIP$	The process terminates successfully
$STOP$	The process reaches a deadlock, making it unable to successfully terminate
$a \rightarrow P$	The process P can only be executed after the execution of the event a
$c?x \rightarrow P$	Before the execution of process P, this process expects to receive a message on the channel c and assign it to the variable x
$c!x \rightarrow P$	The process first sends a variable x through the channel c to another process, and then executes the process P
$P \lhd b \rhd Q$	A process is executed as the process P or the process Q based on the value of the Boolean expression b
$P \square Q$	The external environment determines whether the process executes P or Q
$P \parallel Q$	The process P and Q are executed concurrently
$P \quad Q$	The process Q is sequentially executed after the process P
$P[\|c\|]Q$	The process P and Q are executed in parallel through channels in set c
$P[[a \leftarrow b]]$	The event a in the process P is replaced by another event b

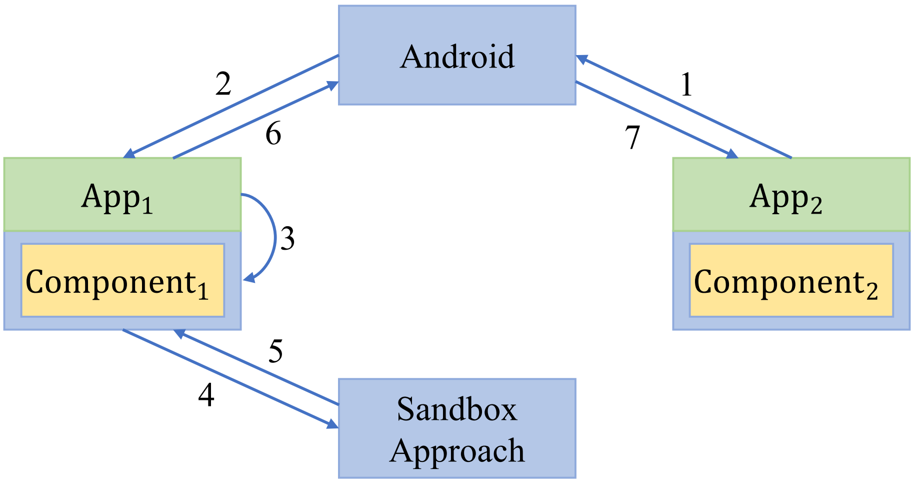


Fig. 5 The Workflows of the Sandbox Approach

2.4 Sandbox approaches

Security sandbox approach (da Costa et al., 2022) involves isolating and compartmentalizing different software components, processes, or applications to ensure minimal mutual interference, whose workflow is shown in Fig. 5. By employing techniques such as process isolation, user permissions and virtualization, controlled environments are created, prohibiting unauthorized accesses, data tempering and other unintended interactions. To further enhance the security, memory protection, isolated network communication, code signing, and runtime permission control mechanisms can also be employed. These combined measures establish a multilayered defense against potential security threats, safeguarding user data, and maintaining system integrity.

By isolating each app within its own sandboxed environment, sandbox methods prevent unauthorized access and interference between applications. This isolation is crucial for protecting user data and privacy, ensuring that malicious apps cannot compromise the system, and maintaining the overall stability and reliability of the Android ecosystem (Vasilescu et al., 2014). Security sandboxing also helps enforce the principle of least privilege, where apps only have access to the resources and data they genuinely need, reducing the risk of security breaches and enhancing user trust in the platform (Sammler et al., 2019).

Henceforth, based on our prior verification work (Lv et al., 2023) and the specific requirements outlined, in this paper, we are considering the implementation of permission control within a sandbox methodology to achieve data isolation and protection among Android apps. To elaborate, we initially classify the data types transferred between apps into two main categories: sensitive data and non-sensitive data. Sensitive data includes information such as contacts, location, address book, and voice recordings. For instance, as illustrated in Fig. 5, when components from App2 seek to access components within App1 (step 1), App2 is required to submit a request containing its unique ID. App1, serving as the recipient, assesses the request (step 2) and distinguishes whether the data requested is sensitive or non-sensitive (step 3). In cases involving non-sensitive data, App1 can securely transmit the requested information directly to App2, as validated in our prior research (Lv et al., 2023). Hence, we do not delve into the details here. In the case of sensitive data, App1 employs the sandbox technology, which is on the basis of the computational logic to generate new values to ensure non-repudiation of the requested party (step 4). This logic may involve applying functions that generate new values, with the actual parameters of these functions including App2's ID, App1's ID, the sensitive data within App1 to be accessed, and the timestamp when the requested computation is carried out by App1 (step 5). Consequently, App2 receives the data that has undergone processing within the sandbox technology (steps 6 & 7), ensuring that sensitive data is transmitted without security leaks.

3 Modeling

In this section, we present a formal model of the ICC mechanism integrated with the sandbox approach in the Android System using CSP. To streamline the model, we designate activities as instances of components in this paper, as activities are the most common components in Apps. To aid comprehension of the modeling process, we provide a sequence diagram depicted in Fig. 6.

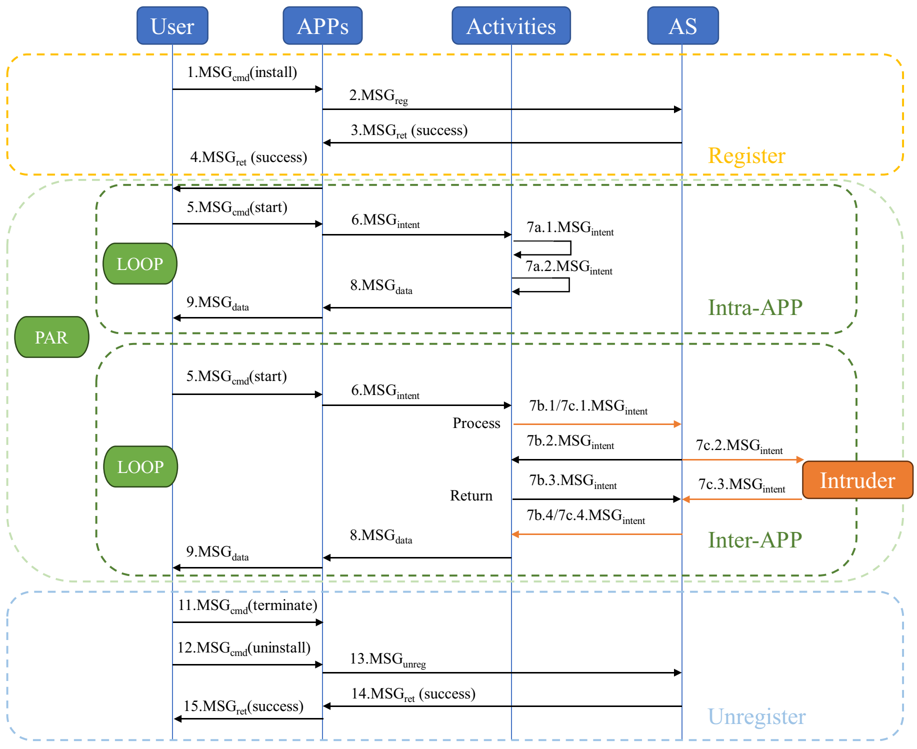


Fig. 6 The sequence diagram of the modeling

From Fig. 6, it can be observed that the entire basic process is divided into register (steps 1-4), unregister (steps 11-15), as well as parallel communication within intra-app and inter-app (steps 5-9). Details are delineated in the following modeling.

3.1 Sets, messages, and channels

To model the ICC mechanism in Android System, we first define the sets, messages, and channels which are used in our model.

First, we define all sets as follows:

- ***UserCMD*** contains all commands that the User sends to the APPs including *install*, *start*, *terminate*, and *uninstall*.
- ***IntentCMD*** involves all commands that intents take, including *startAc*, *processData*, *retData*.
- ***Ret*** includes two types of returned messages, including *success* and *fail*.
- ***Data*** is composed of all data content during communication.
- ***Src*** is comprised of all source types which are determined by the channel that starts the activity, including *SRC_APP*, *SRC_AS*, *ac0*, *ac1*.
- ***Intent*** holds all intents that may be sent through the channels.
- ***Permission*** encompasses all permissions that may be assigned to the data, including 0 (passing without any check), 1 (calling the authentication verification algorithm), and 2 (checking the permission level to determine whether the data needs to ask for dynamic authentication from the user). Note that this Permission category is designed to cater for the permission granted mechanism in the sandbox approach we proposed in this paper.
- ***MsgFlag*** incorporates four flags used to label the type of message.
- ***AppPackage*** contains all names of App packages.
- ***ClassName*** depicts all class names of activities.
- ***Action*** contains all actions that are used in the system.
- ***Category*** embraces all categories that are used in the system.

Next, we define the following messages based on the above definitions:

$$MSG = MSG_{intent} \cup MSG_{cmd} \cup MSG_{ret} \cup MSG_{data} \cup MSG_{reg} \cup MSG_{unreg}$$

$$MSG_{intent} = \{f.intent \mid f \in MsgFlag,\ intent \in Intent\}$$

$$MSG_{permission} = \{f.permission \mid f \in MsgFlag,\ permission \in Permission\}$$

$$MSG_{cmd} = \{cmd \mid cmd \in UserCMD\}$$

$$MSG_{ret} = \{ret \mid ret \in Ret\}$$

$$MSG_{data} = \{data \mid data \in Data\}$$

$$MSG_{reg} = \{a.b.c.d \mid a \in AppPackage,\ b \in ClassName, c \in Action,\ d \in Category\}$$

$$MSG_{unreg} = \{a \mid a \in AppPackage\}$$

MSG_{intent} represents the set of messages carrying intents, where flags indicate the direction of intent, and similarly $MSG_{permission}$ shows the set of messages containing permissions. Msg1 indicates that it is sent from the APPs to the Activities. Msg2 refers to communications between Activities. Msg3 is used to label the intents from the Activities to the AS, which is exactly opposite to Msg4. MSG_{cmd} represents the set of messages from the User. MSG_{ret} represents the set of returned messages. MSG_{data} represents the set of

Table 2 The channels and its relevant explanations adopted in this paper

Channels	Explanations
$User\_APP_i$	the channel between the user and the applications (apps) with the number i
APP_i_AS	the channel between the applications with the number i and the android system AS
$APP_i\_Ac_j$	the channel between the applications with the number i and the activity with the number j
Ac_{ij}_AS	the channel between the activity with the sequence ij and the android system AS
$AC_{ij}\_Ac_{ki}$	the channel between the activity with the sequence ij from the first application and the activity with the sequence ki
Sec_User_AS	the secure channel between the user and the android system AS
$Sec\_APP_i\_User$	the secure channel between the application (app) with the number i and the user
$Fake\_AI$	the intruder channel between the Android system and the intruder

messages that transmit data. MSG_{reg} and MSG_{unreg} are the sets of registration and deregistration requests sent by the APPs to the AS, respectively. MSG includes all messages defined above.

Finally, we define the channels to simulate communications between processes. These channels can be divided into two categories, COM_PATH and INTRUDER_PATH, which are defined as follows and their homologous explanations are shown in Table 2:

– *COM_PATH* contains the normal channels without the intruder, including $User\_APP_i$, APP_i_AS, $APP_i\_Ac_j$, Ac_{ij}_AS, $AC_{ij}\_Ac_{ki}$, Sec_User_AS, and $Sec\_APP_i\_User$. Here, note that in the channel set *COM_PATH*, Sec_User_AS and $Sec\_APP_i\_User$ are assumed to be the secure channels, i.e., these two channels cannot be intruderd or intercepted.
– *INTRUDER_PATH* contains the fake channel *Fake_AI* used when an intruder is present.

3.2 Overall modeling

We first separate the intruder from our system and establish a model named *System_total*, which is composed of the User, the APPs, the Activities, and the Android System (AS). In this paper, the APPs and the AS specifically refer to the processes in the system, which have different meanings from Apps and Android System.

The process User uses the channel $User\_APP_i$ to send commands to the process APPs, and receives messages from them through the same channel. The APPs sends registration and deregistration requests to the process AS, and receives success or fail messages from the AS by the channel APP_i_AS. It sends the start intent to the process Activities to start it, and receives data from them via the channel $APP_i\_Ac_j$. Usually, activities in the process Activities communicate with each other using the channel $Ac_{ij}\_Ac_{ki}$ when they are in the same App. However, if they are in different Apps, they send intents to the AS, and receive them from the AS by the channel Ac_{ij}_AS. The AS can process registration and deregistration requests, and work as a bridge for communication between different Apps. The communication model of *System_total* is shown in the square frame of Fig. 7.

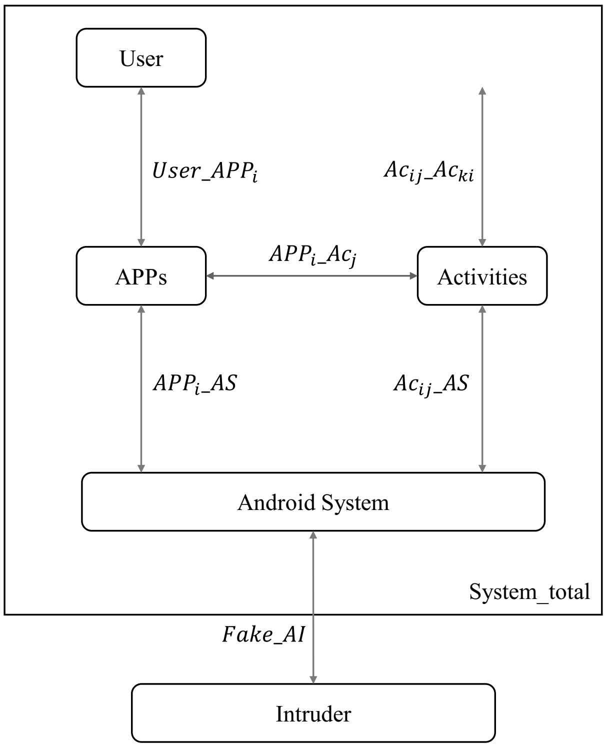


Fig. 7 The communication model of the whole system

When an intruder invades the system, it first forges fake registration information and sends it to the AS via the channel *Fake_AI*. Then, on the one hand, it sends fake intents to the AS through this channel to send spam messages to user's Apps. On the other hand, it can also receive intents from the AS, which should have been sent to the other Apps, so that it can steal user's privacy information.

Thus, *System_total* executing parallel to the process Intruder constructs the whole System. *System_I* indicates the whole system with the intruder. The communication model of the whole System is presented in Fig. 7.

In summary, we get the overall model.

$$System\_total =_{df} User[|COM\_PATH|]APPs[|COM\_PATH|]$$
$$Activities[|COM\_PATH|]AS$$
$$System\_I =_{df} System\_total[|INTRUDER\_PATH|]Intruder$$

3.3 User modeling

Since the user can either send commands to the Apps or receive messages from it, we define three processes named *User_reg*, *User_IntraApp*, and *User_InterApp* to show the registration, communications within applications, and communications among applications respectively. These subprocesses are defined as below.

$$
\begin{aligned}
User\_reg = {} & User\_APP1!install.permission1 \rightarrow User\_APP1?msg41 \rightarrow User\_reg \\
& \square User\_APP2!install.permission2 \rightarrow User\_APP2?msg42 \rightarrow User\_reg \\
User\_IntraApp = {} & User\_APP1!start.permission1 \rightarrow User\_APP1?msg411\{ \\
& retValue = msg411\} \rightarrow User\_IntraApp \\
& \square User\_APP2!start.permission2 \rightarrow User\_APP2?msg422\{ \\
& retValue = msg422\} \rightarrow User\_IntraApp \\
User\_InterApp = {} & Sec\_APP1\_User?app\_11.token\_app11 \rightarrow \\
& if(decide\_to\_send\_token == true)\{ \\
& \quad Sec\_User\_AS!app\_11.token\_app11 \rightarrow \\
& \quad User\_APP1!start.permission1 \rightarrow \\
& \quad User\_APP1?data\{retValue = data\} \rightarrow User\_InterApp\} \\
& else\{ \\
& \quad Sec\_User\_AS!app\_11.null \rightarrow \\
& \quad User\_APP1!start.permission1 \rightarrow \\
& \quad User\_APP1?data\{retValue = data\} \rightarrow User\_InterApp\}
\end{aligned}
$$

The above *User_reg* represents a process where a user concurrently registers with two different applications, *User_APP*1 and *User_APP*2, for installing permissions. In the first part, the user interacts with channel *User_APP*1 by sending a message to request permission installation (*install.permission*1), receives a response message (*msg*41), and then returns to the registration process. Simultaneously, in the second part, the user follows a similar process with *User_APP*2, continuing the registration process with both applications in parallel. For brevity, we omit explanations for *User_IntraApp* and *User_InterApp* as they are similar.

3.4 APP modeling

The APPs process involves multiple Apps running in parallel, with each App exhibiting five states, *i.e.*, "initial", "installing", "installed", "running", and "uninsalling", during its operation (as illustrated in Fig. 8). When an App is not installed, it can receive the user's installation command and send a registration request to the AS. Subsequently, it transitions to the next state, referred to as "installing". During the "installing" state, the App can become the state

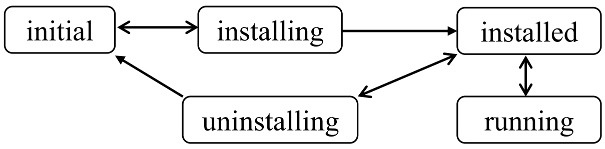


Fig. 8 The State Diagram of the App

“installed" if it receives a success message, which it then forwards to the User. Otherwise, it returns to its initial state “initial" after sending a failure message. Once installed, the user can start or uninstall the App at any time by issuing the corresponding instructions. If the user starts the App, it sends a start intent to the main activity and transitions to the subsequent state named “running". In the “running" state, the App receives at least one message from its activities before eventually receiving a terminate command, which returns it to the “installed" state. If the user opts to uninstall the App, it sends a deregistration request to the AS, transitioning to the “uninstalling" state. Here, the App can revert to its initial state upon receiving a success message, which it then communicates to the user. Otherwise, it returns to the “installed" state after conveying a failure message.

According to the above description, the three relevant subprocesses *APPs_reg*, *APPs_IntraApp* and *APPs_InterApp* are defined as below:

$$
\begin{aligned}
\mathit{APPs\_reg} = {} & \mathit{User\_APP}1?\mathit{cmd}11.p1 \rightarrow \mathit{if}(\mathit{cmd}11 == \mathit{install})\{\\
& \mathit{APP}1\_\mathit{AS}!\mathit{install.permission}1 \rightarrow \mathit{APP}1\_\mathit{AS}?\mathit{ret}41 \rightarrow \mathit{if}(\mathit{ret}41 == \mathit{success})\{\\
& \mathit{User\_APP}1!\mathit{success} \rightarrow \mathit{APPs\_reg}\}\mathit{else}\{\mathit{fail}1 \rightarrow \mathit{APPs\_reg}\}\\
\square & \mathit{User\_APP}2?\mathit{cmd}12.p2 \rightarrow \mathit{if}(\mathit{cmd}12 == \mathit{install})\{\\
& \mathit{APP}2\_\mathit{AS}!\mathit{install.permission}2 \rightarrow \mathit{APP}2\_\mathit{AS}?\mathit{ret}42 \rightarrow\\
& \mathit{if}(\mathit{ret}42 == \mathit{success})\{\ \mathit{User\_APP}2!\mathit{success} \rightarrow \mathit{APPs\_reg}\}\\
& \mathit{else}\{\mathit{fail}1 \rightarrow \mathit{APPs\_reg}\}\ \}\\
\mathit{APPs\_IntraApp} = {} & \mathit{User\_APP}1?\mathit{cmd}11.p1 \rightarrow \mathit{if}(\mathit{cmd}11 == \mathit{start})\{\\
& \{\mathit{intent\_in}\_11 = \mathit{newIntent}(0, 0, \mathit{startAc}, 0, 5, 4, 0);\} \rightarrow\\
& \mathit{APP}1\_\mathit{Ac}1!\mathit{Msg}1.\mathit{intent\_in}\_11 \rightarrow \mathit{APP}1\_\mathit{Ac}1?\mathit{data} \rightarrow\\
& \mathit{User\_APP}1!\mathit{data} \rightarrow \mathit{APPs}_{I}\mathit{ntraApp}\\
& \square\{\mathit{intent\_in}\_12 = \mathit{newIntent}(0, 0, \mathit{startAc}, 0, 5, 4, 0);\} \rightarrow\\
& \mathit{APP}1\_\mathit{Ac}2!\mathit{Msg}1.\mathit{intent\_in}\_12 \rightarrow\\
& \mathit{APP}1\_\mathit{Ac}2?\mathit{data} \rightarrow \mathit{User\_APP}1!\mathit{data} \rightarrow \mathit{APPs\_IntraApp}\}\\
\square & \mathit{User\_APP}2?\mathit{cmd}12.p2 \rightarrow \mathit{if}(\mathit{cmd}12 == \mathit{start})\{\\
& \{\mathit{intent\_in}\_21 = \mathit{newIntent}(0, 0, \mathit{startAc}, 0, 5, 4, 0);\} \rightarrow\\
& \mathit{APP}2\_\mathit{Ac}1!\mathit{Msg}1.\mathit{intent\_in}\_21 \rightarrow \mathit{APP}2\_\mathit{Ac}1?\mathit{data} \rightarrow\\
& \mathit{User\_APP}2!\mathit{data} \rightarrow \mathit{APPs}_{I}\mathit{ntraApp}\\
& \square\{\mathit{intent\_in}\_22 = \mathit{newIntent}(0, 0, \mathit{startAc}, 0, 5, 4, 0);\} \rightarrow\\
& \mathit{APP}2\_\mathit{Ac}2!\mathit{Msg}1.\mathit{intent\_in}\_22 \rightarrow \mathit{APP}2\_\mathit{Ac}2?\mathit{data} \rightarrow\\
& \mathit{User\_APP}2!\mathit{data} \rightarrow \mathit{APPs}_{I}\mathit{ntraApp}\}\\
\mathit{APPs\_InterApp} = {} & \{\mathit{token\_app}1 = \mathit{intent\_from}\_1\_\mathit{to}\_2.\mathit{GenerateRandomNumber}()\} \rightarrow\\
& \mathit{Sec\_APP}1\_\mathit{User}!\mathit{app}\_1.\mathit{token\_app}1 \rightarrow \mathit{User\_APP}1?\mathit{cmd}11.p1 \rightarrow\\
& \mathit{if}(\mathit{cmd}11 == \mathit{start})\{\\
& \quad \mathit{APP}1\_\mathit{Ac}1!\mathit{Msg}1.\mathit{intent\_from}\_1\_\mathit{to}\_2.p1 \rightarrow\\
& \quad \mathit{if}(\mathit{intent\_from}\_1\_\mathit{to}\_2.\mathit{getPermissionCategory}() == 1)\{\\
& \quad\; \mathit{APP}1\_\mathit{Ac}1?\mathit{data} \rightarrow\\
& \quad\; \{\mathit{hash\_value} = \mathit{intent\_from}\_1\_\mathit{to}\_2.\mathit{ComputeHash}(\mathit{data} + \mathit{token}_{a}\mathit{pp}1);\}\\
& \quad\; \rightarrow \mathit{User\_APP}1!\mathit{hash\_value} \rightarrow \mathit{APPs\_InterApp}\}\\
& \mathit{else}\{\mathit{APP}1\_\mathit{Ac}1?\mathit{data} \rightarrow \mathit{User\_APP}1!\mathit{data} \rightarrow \mathit{APPs\_InterApp}\}\ \ \}
\end{aligned}
$$

Here, we take *APPs_IntraApp* process as an example. It delineates the communication flow between two user applications (*User_APP*1 and *User_APP*2) and their respective internal activities (APP1 and APP2). Specifically, *User_APP*1 initiates a command *cmd*11 to APP1, generating an intent *intent_in*_11 if *cmd*11 is *start*, which is then relayed to *APP*1_*Ac*1. Then, the data from *APP*1_*Ac*1 is passed back to *User_APP*1, before the process loops back to *APPs_IntraApp*. Alternatively, if *cmd*11 is not *start*, a similar sequence of events follows for another command. Similarly, *User_APP*2 triggers a command *cmd*12 to APP2, creating an intent *intent_in*_21 if *cmd*12 is *start*, and transmitting it to *APP*2_*Ac*1. Then, the data received from *APP*2_*Ac*1 is conveyed back to *User_APP*2, before returning to *APPs_IntraApp*. Otherwise, if *cmd*12 is not *start*, a comparable sequence unfolds for another command. The descriptions of the processes *APPs_reg* and *APPs_InterApp* mirror the structure outlined in *APPs_IntraApp*. For brevity, we skip the detailed elaboration.

3.5 Activity modeling

Similar to the process APPs, the process Activities is composed of multiple activities running in parallel. For convenience, we omit some functions of the activity and simplify its life cycle, in which situation an activity has three states. When an activity is not started, it can receive a start intent from three kinds of sources. An activity can also be started by another activity in the same App. In the last case, the IAC mechanism in Android System helps an activity be started by activities in different Apps.

Any of the above three situations will bring the activity to the second state, in which it reacts depending on the command in the start intent. If the command is to start an activity in the same App, it sends a start intent to the target activity and waits for the returned data from it by the symmetric channel. Similarly, if the command is to start an activity in different Apps, it sends a start intent to the AS and wait for the returned data. In these situations, it goes to the third state, where it processes the returned data and sends it back to the source that started it. Another possible instruction in the intent is to process data.

Since there are three kinds of sources, we specifically define a process named Ret, in which it sends the message to the APPs if the source is the App. It sends an intent to the source activity when it was started by an activity in the same App. If the source is the AS, it sends an intent back to it. Then, the activity returns to the initial state after it sends the message back to the source.

The subprocesses *Activities_IntraApp* and *Activities_InterApp* are modeled as below.

$$
\begin{aligned}
Activities\_InterApp = \ & APP1\_Ac1?Msg1.intent.permission \rightarrow \\
& if(intent.getCmd() == startAc)\{ \\
& \ if(intent.getTarget() == 22 \ || \ intent.getTarget() == 21)\{ \\
& \ \ \{call(process, intent.getData(), intent.getDes()); \\
& \ \ intent\_from\_1\_to\_2 = newIntent(intent.getSrc(), \\
& \ \ intent.getTarget(), processData, 0, intent.getData(), \\
& \ \ intent.getRequestLevel(), intent.getPermissionCategory());\} \rightarrow \\
& \ \ Ac11\_AS!intent\_from\_1\_to\_2.permission \rightarrow Ac11\_AS?msg \rightarrow \\
& \ \ APP1\_Ac1!msgTemp \rightarrow Activities\_InterApp\} \\
& \ else\{Activities\_IntraApp\} \ \} \\
& else \ if(intent.getCmd() == processData)\{ \\
& \ \{call(process, intent.getData(), intent.getDes());\} \rightarrow \\
& \ Ac11\_AS!intent\_from\_1\_to\_2.permission \rightarrow \\
& \ Ac11\_AS?msg \rightarrow APP1\_Ac1!msgTemp \rightarrow Activities\_InterApp\}
\end{aligned}
$$

The *Activities_InterApp* process governs the interactions between APP1's activity $Ac1$ and external processes based on received intents. Initially, it listens for $Msg1$ intents from $APP1\_Ac1$, and extracts permission information. If the intent's command is to start an activity *startAc* and its target matches specific criteria (22 or 21), it triggers a call to a designated process, followed by the creation of a new intent ($intent\_from\_1\_to\_2$). This new intent is then transmitted to $Ac11\_AS$, where it is processed and the resulting message *msg* is relayed back to $APP1\_Ac1$. Alternatively, if the command is to process data *processData*, it directly calls the designated process, sends the intent to $Ac11\_AS$ for processing, and proceeds with the loop. If the conditions aren't met, control is passed to *Activities_IntraApp.*

$Activities\_IntraApp = APP1\_Ac1?Msg1.intent \rightarrow$

$if(intent.getCmd() == startAc)\{if(intent.getTarget() == 0)\{$

$\{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());$

$intent.setCmd(retData);\} \rightarrow$

$APP1\_Ac1!msgTemp \rightarrow Activities\_IntraApp\}$

$else\{Activities\_InterApp\}\}$

$else\ if(intent.getCmd() == processData)\{\{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());$

$intent.setCmd(retData);\} \rightarrow$

$APP1\_Ac1!msgTemp \rightarrow Activities\_IntraApp\}$

$\square APP1\_Ac2?Msg1.intent \rightarrow if(intent.getCmd() == startAc)\{$

$if(intent.getTarget() == 0)\{\{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());$

$intent.setCmd(retData);\} \rightarrow$

$APP1\_Ac2!msgTemp \rightarrow Activities\_IntraApp\}$

$else\{Activities\_InterApp\}\ \}$

$else\ if(intent.getCmd() == processData)\{\{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());$

$intent.setCmd(retData);\} \rightarrow$

$APP1\_Ac1!msgTemp \rightarrow Activities\_IntraApp\}$

$\square APP2\_Ac1?Msg1.intent \rightarrow if(intent.getCmd() == startAc)\{$

$if(intent.getTarget() == 0)\{\{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());$

$intent.setCmd(retData);\} \rightarrow$

$APP2\_Ac1!msgTemp \rightarrow Activities\_IntraApp\}$

$else\{Activities\_InterApp\}\}$

$else\ if(intent.getCmd() == processData)\{\{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());$

$intent.setCmd(retData);\} \rightarrow$

$APP1\_Ac1!msgTemp \rightarrow Activities\_IntraApp\}$

$\square APP2\_Ac2?Msg1.intent \rightarrow if(intent.getCmd() == startAc)\{$

$if(intent.getTarget() == 0)\{\ \{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());$

$intent.setCmd(retData);\} \rightarrow$

$APP2\_Ac2!msgTemp \rightarrow Activities\_IntraApp\}$

$else\{Activities\_InterApp\}\ \}$

$else\ if(intent.getCmd() == processData)\{\{intent.setCmd(processData);$

$call(process, intent.getData(), intent.getDes());intent.setCmd(retData);\} \rightarrow\ APP1\_Ac1!msgTemp \rightarrow Activities\_IntraApp\}$

The *Activities_IntraApp* process oversees interactions within APP1‘s activity *Ac*1, triggered by received *Msg*1 intents. Upon receiving an intent, it checks if the command is to start an activity *startAc*. If so, and the target is 0, it modifies the intent’s command to *processData*, invokes a designated process, and then changes the command to *retData*. Subsequently, it forwards the modified message *msgTemp* back to *APP*1_*Ac*1. If the command is not to start an activity, control shifts to *Activities_InterApp*. Similarly, if the command is to process data *processData*, it follows a comparable sequence of modifying the intent, calling the designated process, changing the command to *retData*, and returning the message to *APP*1_*Ac*1, continuing the loop within *Activities_IntraApp*. Due to the uncertainty stemming from the four parts comprising the *Activities_IntraApp* process, we utilize the first part as an example for explanation, and omit the rest.

3.6 Android system modeling

Android System is the foundation and guarantee of the IAC mechanism. On one hand, it can process registration and deregistration requests to maintain the register table. On the other hand, it sends the intent to the right place by comparing the information in the intent with the register table.

We define two subprocesses to simulate the function of the AS, in other words, the AS is a process where Register, Unregister, and Matcher run in parallel. Register receives registration requests from Apps, if the information is correct and not repetitive, it records them in the register table and return a success message to the App, otherwise, it will be a fail message and the register table won’t update either.

Unregister works like Register, except it receives deregistration requests. Matcher receives intents from activities, and then sends them to the right place according to the register table. These processes share one register table, and the subprocesses *AS_reg* and *AS_InterApp* are defined as below.

$$
\begin{aligned}
AS\_reg = {} & APP1\_AS?msg21.p1 \rightarrow APP1\_AS!success \rightarrow AS\_reg \\
& \Box APP2\_AS?msg22.p2 \rightarrow APP2\_AS!success \rightarrow AS\_reg \\
AS\_InterApp = {} & Sec\_User\_AS?app\_111.token\_app111 \rightarrow \\
& \{intent\_from\_1\_to\_2.StoreToken(app\_111, token\_app111);\} \rightarrow \\
& Ac11\_AS?intent.permission \rightarrow \\
& (if(intent\_from\_1\_to\_2.getPermissionCategory() == 0)\{ \\
& \{call(process, intent.getData(), intent.getDes());\} \rightarrow \\
& Ac11\_AS!msgTemp \rightarrow AS\_InterApp\} \\
& else\ if(intent\_from\_1\_to\_2.getPermissionCategory() == 1)\{ \\
& \{b = intent.VerifyIdentity(intent.getData() + token\_app111, hash\_value)\} \rightarrow \\
& \{call(authentication, b);\} \rightarrow if(isValid == true)\{ \\
& \{call(process, intent.getData(), intent.getDes());\} \rightarrow \\
& Ac11\_AS!msgTemp \rightarrow AS\_InterApp\} \\
& else\{\{msgTemp = 0\} \rightarrow Ac11\_AS!msgTemp \rightarrow AS\_InterApp\} \\
& \}else\{if(intent.getRequestLevel() - 23 < 0)\{ \\
& if((intent.getRequestLevel() == 1 \&\& permission.getL1() == 1) \,\|\, \\
& (intent.getRequestLevel() == 4 \&\& permission.getL4() == 1) \,\|\, \\
& (intent.getRequestLevel() == 17 \&\& permission.getL17() == 1))\{ \\
& \{call(process, intent.getData(), intent.getDes());\} \rightarrow \\
& Ac11\_AS!msgTemp \rightarrow AS_InterApp \\
& \}else\{\{msgTemp = 0\} \rightarrow Ac11\_AS!msgTemp \rightarrow AS\_InterApp\} \\
& \}else\{if((intent.getRequestLevel() == 23 \&\& permission.getL23() == 1) \,\|\, \\
& (intent.getRequestLevel() == 27 \&\& permission.getL27() == 1) \,\|\, \\
& (intent.getRequestLevel() == 28 \&\& permission.getL28() == 1) \,\|\, \\
& (intent.getRequestLevel() == 29 \&\& permission.getL29() == 1))\{ \\
& \{call(process, intent.getData(), intent.getDes());\} \rightarrow \\
& Ac11\_AS!msgTemp \rightarrow AS\_InterApp \\
& \}else\{\{grant = user.getGrant();\} \rightarrow if(grant == true)\{ \\
& \{call(process, intent.getData(), intent.getDes());\} \rightarrow \\
& Ac11\_AS!msgTemp \rightarrow AS_InterApp \\
& \}else\{\{msgTemp = 0\} \rightarrow Ac11\_AS!msgTemp \rightarrow AS_InterApp\}\}\}\})
\end{aligned}
$$

Here, the *AS_InterApp* process orchestrates interactions involving the exchange of tokens and intents between *Sec_User_AS* and *Ac*11*_AS*. Initially, it receives *app_*111*.token_app*111 from *Sec_User_AS*, which is then used to store a token *StoreToken* for *app_*111. Subsequently, it receives an intent's permission information from *Ac*11*_AS*. Depending on the permission category, different actions are taken: if it is category 0, a process is called to handle the intent's data, otherwise, when it is category 1, an identity verification is performed before processing the intent's data. In this case, if the request level of the intent is less than 23, additional checks are made based on different permission levels. For request level 23 or higher, further checks are conducted. If none of the conditions are satisfied, a grant is verified from the user, allowing the subsequent processing of the intent's data. If the grant is not provided, the process halts.

3.7 Intruder modeling

An intruder disguises itself as an ordinary App and interact with the AS to achieve its purpose.

Initially, it falsifies registration information and transmits it to the AS, whereupon the AS logs this fabricated data in the registration table. The intruders have no reason to uninstall themselves, therefore, there is no channel to send deregistration requests.

On one hand, when a normal App sends intent with data to another App, the intent is sent to the AS first, however, the fake information in the register table may lead it to the wrong destination through the channel *Fake_AI*. That's why intruders can get user's privacy information through the AS. On the other hand, it proactively sends intents to the AS whose destinations are activities in the user's Apps. In this situation, the user's App may be started by the AS incorrectly if the destination matches exactly, which means it is maliciously attacked.

The process Intruder is defined as follows, meanwhile, the AS is updated to *AS_InterApp_I*.

$Intruder = Fake\_AI?intent.permission \rightarrow$

$fakeI\{intentTemp = newIntent(11, 12, startAc, 22, 6, 4, 0)\} \rightarrow$

$Fake\_AI!intentTemp.permission \rightarrow Intruder$

$AS\_InterApp\_I = AS\_InterApp()$

$\square Sec\_User\_AS?app\_111.token\_app111 \rightarrow$

$\{intent\_from\_1\_to\_2.StoreToken(app\_111, token\_app111);\} \rightarrow$

$Ac11\_AS?intent.permission \rightarrow Fake\_AI!intent.permission \rightarrow$

$Fake\_AI?intent.permission \rightarrow$

$if(intent\_from\_1\_to\_2.getPermissionCategory() == 0)\{\{call(process, intent.getData(), intent.getDes());\} \rightarrow$

$Ac11\_AS!msgTemp \rightarrow AS\_InterApp\_I\}$

$elseif(intent\_from\_1\_to\_2.getPermissionCategory() == 1)$

$\{\{b = intent.VerifyIdentity(intent.getData() + token_app111, hash_value)\} \rightarrow$

$\{call(authentication, b);\} \rightarrow if(isValid == true)\{\{call(process, intent.getData(), intent.getDes());\} \rightarrow$

$Ac11\_AS!msgTemp \rightarrow AS\_InterApp\_I\}$

$else\{\{msgTemp = 0\} \rightarrow Ac11\_AS!msgTemp \rightarrow AS\_InterApp_I\}$

$\}else\{if(intent.getRequestLevel() - 23 < 0)\{if((intent.getRequestLevel() == 1\&\&permission.getL1() == 1) \|$

$(intent.getRequestLevel() == 4\&\&permission.getL4() == 1) \|$

$(intent.getRequestLevel() == 17\&\&permission.getL17() == 1))\{\{call(process, intent.getData(), intent.getDes());\} \rightarrow$

$Ac11\_AS!msgTemp \rightarrow AS_InterApp_I\}else\{\{msgTemp = 0\} \rightarrow Ac11\_AS!msgTempAS\_InterApp\_I\}$

$\}else\{if((intent.getRequestLevel() == 23\&\&permission.getL23() == 1) \|$

$(intent.getRequestLevel() == 27\&\&permission.getL27() == 1) \|$

$(intent.getRequestLevel() == 28\&\&permission.getL28() == 1) \|$

$(intent.getRequestLevel() == 29\&\&permission.getL29() == 1))\{\{call(process, intent.getData(), intent.getDes());\} \rightarrow$

$Ac11\_AS!msgTemp \rightarrow AS_InterApp_I$

$\}else\{\{grant = user.getGrant();\} \rightarrow if(grant == true)\{\{call(process, intent.getData(), intent.getDes());\} \rightarrow$

$Ac11\_AS!msgTemp \rightarrow AS\_InterApp\_I$

$\}else\{\{msgTemp = 0\} \rightarrow Ac11\_AS!msgTemp \rightarrow AS\_InterApp\_I\}\}\}\}$

Here, we explain the *AS_InterApp_I* process as an example. Initially, it inherits the functionalities of *AS_InterApp*, establishing token storage and permission handling between *Sec_User_AS* and $Ac11\_AS$. Additionally, it adopts the channel *Fake_AI* to forward received intents for further processing. Depending on the permission category, it either initiates data processing or conducts an identity verification process before proceeding. In the latter scenario, it includes conditional checks based on request levels and permission levels, ensuring appropriate handling of intent data. Ultimately, it maintains a continuous loop of intent processing and communication with $Ac11\_AS$, iterating through the outlined decision pathways.

4 Verification

In this section, we implement the model using the PAT model checker. Subsequently, we aim to verify several fundamental yet crucial properties of the model established in the last section.

4.1 Implementation

We need to do some preparatory work before establishing the model in PAT.

First of all, we define the number of Apps in the system and the activities in each App. We set the number of Apps to 2 (constant *N*), considering the possible state explosion. Similarly, the max number of activities in each App is 2 (constant *MM*), and each App has exactly $M[i]$ activities, both of them are 2. In PAT, they are presented as follows:

$$\begin{aligned}&\#define\ N\ 2;\\&\#define\ MM\ 2;\\&var\ M[N] = [2, 2];\end{aligned}$$

Then, we list all enumerations introduced in Section 3. For brevity, we give one of the statements as an example shown as below.

$$enum\{startAc, processData, retData\};$$

Next, some typical channels are defined as follows. Their buffer sizes are all set to zero, which indicates that they are all synchronous communication channels. To prevent process deadlock caused by resource competition on channels, we use arrays to denote multiple channels. For example, there are *MM* activities in each App, so we need $MM * MM$ $Ac_{ij}\_Ac_{ki}$ channels for communications in a single App, in other words, *N* Apps need $N * MM * MM$ channels.

$$\begin{aligned}&channel\ User\_App_i[N]\ 0;\\&channel\ APP_i\_AS[N * MM]\ 0;\\&channel\ Ac_{ij}\_Ac_{ki}[N * MM * MM]\ 0;\\&channel\ Fake\_AI\ 0;\end{aligned}$$

Finally, to make the program run normally and record some states in the program, we define the variables *userMsg*, *intruderMsg*, *msgTemp*, and *intentTemp*. The definition of *intentTemp* is as follows, where the Intent is a class defined by C#, whose parameters represent *src*, *des*, *cmd*, *target*, *data*, *requestlevel*, and *permissionCategory*, respectively.

$$var < Intent >\ \ intentTemp = new\ Intent(0, 0, startAc, 0, 5, 4, 0);$$

We introduce the C# extension in PAT to describe the intent-matching mechanism on a lower abstraction level, so as to improve the precision of our model. For instance, the function *GenerateRandomNumber*() is designed to produce the random number as the token, the function *ComputeHash*(*stringinput*) is adopted to compute the hash value by classic SHA algorithm, and the function *VerifyIdentity* is used to verify whether the original data is the same as the hash data computed by the function *ComputeHash*. Part of the code is demonstrated below.

```
public static int GenerateRandomNumber()
{
        Random random = new Random();
        return random.Next(1, 1000);
}
public static string ComputeHash(stringinput)
{
    using (SHA256 sha256 = SHA256.Create())
    {
        byte[] bytes = Encoding.UTF8.GetBytes(input);
        byte[] hashBytes = sha256.ComputeHash(bytes);
        return BitConverter.ToString(hashBytes).Replace(ε − ε, εε).ToLower();
    }
}
public static bool VerifyIdentity(string originalData, string hashedData)
{
    string computedHash = ComputeHash(originalData);
    return computedHash.Equals(hashedData, StringComparison.OrdinalIgnoreCase);
}
```

To represent the procedure of intent transmission and data processing more intuitively, we create a special rule to process the data in activities. First, we give each $Activity_{i,j}$ a unique ID, which is i multiplied by MM, and then add j. Additionally, the Intruder also gets an ID 4, since the max ID of normal activities is 3 in our model. Second, we store our temporary data in the intent, and when an activity is processing the intent, it adds its activity ID to the end of the data. For example, if an intent carrying the temporary data 5 is processed by $Activity_{1,1}$ whose ID is 3, the return intent carries the new data 53.

Here we present two typical situations, Intra-App Communication and IAC disturbed by the Intruder.

In Table 3, an intent carrying user's privacy data which indicates 5 is produced in state a. It is sent to $Ac_{0,0}$ and $Ac_{0,1}$ successively. In state c, the intent is processed by $Ac_{0,1}$ whose ID is 1, therefore, the new data in the intent becomes 51. When the intent is returned to $Ac_{0,0}$, the returned data is processed again and become 510 in state d since the activity's ID is 0. In the last state, the intent is brought back to the User, and the User finally gets *userMsg* which is 510.

In Table 4, the first two steps are similar to the Intra-App Communication situation. In state c, the intent is sent to the AS. After that, the intent is expected to be sent to $Ac_{1,1}$ and get new data 53 since the activity ID is 3. However, when the Intruder disturbs the communication, the intent is sent to the Intruder incorrectly. Therefore, in state d, the actual data is 54. Although the intent is sent back in state e to state g, the actual data is 540 at last rather than 530. This simulation shows how the Intruder obtains and tampers with our data.

Table 3 Simulation of Intra-App Communication

	User	APP_0		AS	APP_1		Intruder	User Msg	Intent Data
		$Ac_{0,0}$	$Ac_{0,1}$		$Ac_{1,0}$	$Ac_{1,1}$			
ID		0	1		2	3	4		
a	○							0	5
b		○						0	5
c			○					0	51
d		○						0	510
e	○							510	510

Privacy Data = 5 ○ : Intent Place

Table 4 Simulation of IAC (Disturbed by the Intruder)

	User	APP_0		AS	APP_1		Intruder	Except Data	Actual Data
		$Ac_{0,0}$	$Ac_{0,1}$		$Ac_{1,0}$	$Ac_{1,1}$			
ID		0	1		2	3	4		
a	X ○							5	5
b		X ○						5	5
c				X ○				5	5
d						X	○	53	54
e				X ○				53	54
f		X ○						530	540
g	X ○							530	540

Privacy Data = 5 X : Expected Place ○ : Actual Place
Expected UserMsg = 530 Actual UserMsg = 540

4.2 Properties verification

Here, we give the several different processes about *System* as below.

$$\begin{aligned}
System\_reg &= User\_reg \parallel APPs\_reg \parallel AS\_reg\\
System\_IntraApp &= User\_IntraApp \parallel APPs\_IntraApp \parallel Activities\_IntraApp\\
System\_InterApp &= User\_InterApp \parallel APPs\_InterApp\\
&\quad \parallel Activities\_InterApp \parallel AS\_InterApp\\
System\_total &= System\_reg \Box System\_InterApp \Box System\_IntraApp\\
System\_InterApp\_I &= User\_InterApp \parallel APPs\_InterApp \parallel\\
&\quad Activities\_InterApp \parallel AS\_InterApp\_I\\
System\_I &= System\_reg \Box System\_InterApp\_I \Box System\_IntraApp
\end{aligned}$$

4.2.1 Deadlock freedom

Initially, we assess the deadlock freedom of our model. Smooth communication between processes can only ensue when the system avoids entering a deadlock state, enabling subsequent property verifications. In PAT, this property can be easily confirmed with the following statement.

$$\#assert\ System\_total()\ deadlockfree;$$
$$\#assert\ System\_I()\ deadlockfree;$$

4.2.2 Divergence freedom

Subsequently, we examine the divergence freedom of our model. Divergence freedom signifies the flexibility and fault tolerance within a system, enabling individual components to evolve independently without compromising the overall functionality, stability, or security of the system. In PAT, this property can be readily confirmed with the following statement.

$$\#assert\ System\_total()\ divergencefree;$$
$$\#assert\ System\_I()\ divergencefree;$$

4.2.3 Data reachability

The ICC mechanism involves two types of communication, depending on whether the components are within the same app or not.

To simulate Intra-App Communication, we initiate $Activity_{1,0}$ in APP_1 and have it invoke $Activity_{1,1}$ with IDs 2 and 3, respectively. With our initial data defined as 5 and following the aforementioned data processing rule, the user precisely receives the message 53.

Consequently, we employ the following statements to validate reachability in the case of Intra-App Communication.

#define Data_Reachability_Intra_App

($retValue == 50 \,||\, retValue == 51 \,||\, retValue == 52 \,||\, retValue == 53$);

#assert System_total reaches Data_Reachability_Intra_App;

Similarly, to simulate Inter-App Communication, we initiate $Activity_{0,0}$ in APP_0 and have it commence $Activity_{1,1}$ with IDs 0 and 3, respectively. In this scenario, the user receives the message 53. The ensuing statements serve to confirm the reachability in the case of Inter-App Communication.

#define Data_Reachability_Inter_App

($retValue == 50 \,||\, retValue == 51 \,||\, retValue == 52 \,||\, retValue == 53$);

#assert System_total reaches Data_Reachability_Inter_App;

4.2.4 Data security

If an intruder is able to send an intent to our activity and obtain our activity ID, it signifies that the data in our normal Apps is not secure. In our model, the intruder sends an intent with the initial data set to 6, and its activity ID is 4. The intruder processes the data using the same rule we defined earlier, suggesting that the intruder may receive the message 63 in the event of data leakage. The following statements are employed to validate data security.

#define Data_Leakage

($intruderMsg == 60 \,||\, intruderMsg == 61 \,||\, intruderMsg == 62 \,||\, intruderMsg == 63$);

#assert System_I ⊧ []! *Data_Leakage*;

4.3 Verification results

In accordance with our model's implementation, the verification of these above properties is confirmed through Property Analysis Toolkit 3.5.1 (PAT), which is installed on a Windows x64 PC with an Intel i7 CPU and 16GB memory. The corresponding verification outcomes are presented in Table 5.

Table 5 Results of verification in PAT

	Deadlock Freedom	Divergence Freedom	Data Reachability		Data Leakage	
			Intra-App	Inter-App	Intra-App	Inter-App
Result	VALID	VALID	VALID	VALID	VALID	VALID
Time	0.0048485s	0.0022826s	0.0009878s	0.0016384s	0.0133691s	0.0018275s
Transitions	184	144	14	14	246	247

Fortunately, all four properties have been substantiated as valid, effectively mitigating concerns related to data leakage and security issues identified in our earlier research. The verification results not only attest to the robustness of our proposed permission control but also highlight the efficacy of integrating the sandbox approach with the original ICC mechanism.

Actually, to the best of our knowledge, in the current Android system, security measures such as application permission control and sandbox isolation are already in place. However, they often fall short of fully preventing attacks from malicious applications due to vulnerabilities such as over-permission granting or sandbox escapes in practice. The ICC mechanism integrated with sandbox approach proposed in this article primarily introduces an algebraic approach to enhance the overall design security, thereby theoretically ensuring data security.

5 Conclusion

In summary, this paper extended our previous research by specifically targeting data leakage and privilege escalation within the ICC mechanism. We primarily focused on theoretical analysis, introducing sandboxing and permission controls to effectively address these challenges. Verification results from CSP validate the efficacy of our approach.

In our future work, we will simulate and refine the extended ICC mechanism to enhance secure information sharing. Realistic use cases will showcase how our new sandbox mechanism effectively prevents or mitigates security threats.

Author contributions Jiaqi Yin and Sini Chen were responsible for experiment implementation and paper writing, Yixiao Lv primarily provided guidance for conducting experiments, and Huibiao Zhu was in charge of guiding and polishing papers.

Funding This work was partially supported by the National Natural Science Foundation of China (No. 62032024), the “Digital Silk Road” Shanghai International Joint Lab of Trustworthy Intelligent Software (No. 22510750100), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and China Basic Research Programs of Taicang (No. TC2022JC14).

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interest The authors declare no competing interest.

References

Almomani, I. M., & Al Khayer, A. (2020). A comprehensive analysis of the android permissions system. *IEEE access, 8*, 216671–216688.

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012). Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 217–228).

Bhandari, S., Jaballah, W. B., Jain, V., Laxmi, V., Zemmari, A., Gaur, M. S., Mosbah, M., & Conti, M. (2017). Android inter-app communication threats and detection techniques. *Computers & Security, 70*, 392–421.

Biswas, S., Sohel, M., Sajal, M. M., Afrin, T., Bhuiyan, T., & Hassan, M. M. (2018). A study on remote code execution vulnerability in web applications. In *International Conference on Cyber Security and Computer Science (ICONCS 2018)* (pp. 50–57).

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A. R., & Shastry, B. (2012). Towards taming privilege-escalation attacks on android. *In NDSS, 17*, 19.

Chin, E., Felt, A. P., Greenwood, K., Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (pp. 239–252).

da Costa, F. H., Medeiros, I., Menezes, T., da Silva, J. V., da Silva, I. L., Bonifácio, R., Narasimhan, K., & Ribeiro, M. (2022). Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification. *Journal of Systems and Software, 183*, 111092.

Developers A. Developer guides: Intents and intent filters. https://developer.android.com/guide/components/intents-filters.html. Accessed in 2023.

DiMarzio, J. F. (2008). Android™ A Programmer's Guide.

Fang, Z., Han, W., & Li, Y. (2014). Permission based android security: Issues and countermeasures. *Computers & Security, 43*, 205–218.

Gadient, P., Ghafari, M., Frischknecht, P., & Nierstrasz, O. (2019). Security code smells in android icc. *Empirical Software Engineering, 24*(5), 3046–3076.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Lowe, G., & Roscoe, B. (1997). Using csp to detect errors in the tmn protocol. *IEEE Transactions on Software Engineering, 23*(10), 659–669. https://doi.org/10.1109/32.637148

Lv, Y., Yin, J., Chen, S., & Zhu, H. (2023). Formalization and verification of the icc mechanism in android system using csp. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 89–95). IEEE.

Neuner, S., Vander Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., & Weippl, E. (2014). Enter sandbox: Android sandbox comparison. Preprint retrieved from http://arxiv.org/abs/1410.7749

Samhi, J., Bartel, A., Bissyandé, T. F., & Klein, J. (2021). Raicc: Revealing atypical inter-component communication in android apps. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 1398–1409). https://doi.org/10.1109/ICSE43902.2021.00126

Sammler, M., Garg, D., Dreyer, D., & Litak, T. (2019). The high-level benefits of low-level sandboxing. *Proceedings of the ACM on Programming Languages, 4*(POPL), 1–32.

Vasilescu, M., Gheorghe, L., & Tapus, N. (2014). Practical malware analysis based on sandboxing. In *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference* (pp. 1–6). IEEE.

Xu, J., Yin, J., Zhu, H., & Xiao, L. (2021). Modeling and verifying producer-consumer communication in kafka using CSP. *7th Conference on the Engineering of Computer Based Systems*.

Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., Gunter, C.A., & Nahrstedt, K. (2013) Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (pp. 1017–1028)

Jiaqi Yin is currently an assistant professor in School of Software, Northwestern Polytechnical University. He received his Ph. D degree in software engineering in East China Normal University in 2022. His research interests include process algebra, program verification, edge computng and trustworthy AI.

Sini Chen is currently a Ph.D student in Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai. She received her B.S. degree in software engineering in East China Normal University in 2021. Her research interests include process algebra, program semantics, object-oriented systems and type systems.

Yixio Lv is currently a software engineer for a semiconductor company. He received his B.S. degree in software engineering in East China Normal University in 2024. His research interests include process algebra and Android systems.

Huibiao Zhu is currently a professor in East China Normal University, Shanghai. He earned his Ph.D. degree in formal methods from London South Bank University, London, in 2005.During these years, he has studied various semantics and their linking theories for Verilog, SystemC, web services and probability system. He was the Chinese PI of the Sino-Danish Basic Research Center IDEA4CPS.