An adaptive and configurable protection framework against android privilege escalation threats

Yang Xu a,*, Guojun Wang b, Ju Ren a, Yaoxue Zhang a

a School of Information Science and Engineering, Central South University, China
b School of Computer Science and Technology, Guangzhou University, China

HIGHLIGHTS

- Adaptive Android security framework for defending against confused deputy attack.
- Policy and capability-based access control for inter-component communication.
- Runtime policy configuration with effective risk mitigation mechanism for policies.
- Proactive decision cache for ensuring efficient and dependable decision services.

ARTICLE INFO

Article history:
Received 31 March 2018
Received in revised form 26 August 2018
Accepted 17 September 2018
Available online 11 October 2018

Keywords:
Android
Privilege escalation threat
Dynamic permission mechanism
Capability-based access control
Configurability

ABSTRACT

Android is a successful mobile platform with a thriving application ecosystem. However, despite its security precautions like permission mechanism, it is still vulnerable to privilege escalation threats and particularly confused deputy attacks that exploit the permission leak vulnerabilities of Android applications. Worse, most existing detection and protection techniques have become costly and unresponsive in current Android dynamic permission environments. In this paper, we propose a configurable Android security framework to prevent the exploitation of permission leak vulnerabilities of third-party applications via confused deputy attacks. Our framework collects the runtime states of applications and enforces policy and capability-based access control to restrain risky inter-application communications, so as to provide more responsive, adaptive, and flexible application protection. Besides, our framework provides users with a flexible runtime policy configuration together with a complementary security mechanism to mitigate risks induced by inappropriate policies. Additionally, we present a sophisticated access decision cache system with a proactive maintenance method that ensures the efficiency and dependability of decision services. Theoretical analysis and experimental evaluation demonstrate that our approach provides configurable and effective protections for third-party applications against permission leak vulnerabilities at small performance and usability costs.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Android is a phenomenally successful operating system (OS) and has been deployed on billions of different devices, including smartphones, tablets, wearable devices and intelligent appliances, and will continue dominating the OS industry for the foreseeable future [1]. Android has a thriving application (app) ecosystem in which many apps can be conveniently delivered from developers to users through various booming app marketplaces and traditional methods like websites. However, such flexible supply of apps has backfired because vulnerable apps and malware can be easily obtained by the unsuspecting users without adequate security checks.

Although Android has adopted various security mechanisms such as sandbox and permissions to deal with related security threats, the results are far from satisfactory as malicious activities focusing on Android apps continue [2–6]. In this context, privilege escalation threats such as permission leak vulnerabilities are quite common in Android apps and are widely exploited in app-tier confused deputy attacks [7–12], in which benign but vulnerable apps are exploited by malicious apps, as confused deputies to execute privileged operations.

Android has enhanced its permission scheme with new dynamic features since the milestone version Android 6.0 Marshmallow (API level 23) [13], benefiting the majority of current...
Android users [14], but this approach has achieved limited success in protecting against these permission leak threats. Meanwhile, numerous relevant security hardening studies have focused on this arena [15,16] and fall into two broad categories: static analysis approaches like IccTA [17] and Covert [11,12], and dynamic solutions such as TaintDroid [18] and Xmandroid [19,20]. Unfortunately, static-analysis-based approaches are insufficient for handling all runtime problems due to the dynamic features provided by Android, like dynamic code loading techniques. Dynamic approaches are also inadequate, with most existing approaches burdened by substantial runtime overhead, especially in current dynamic scenarios [8,21]. Additionally, such methods are also unresponsive to dynamic features due to their inflexible and unadjustable security mechanisms that inevitably compromise app usability [19,20].

Motivated to address the problems described above, we propose an adaptive and configurable Android framework to precisely and responsively prevent the permission leak vulnerabilities of third-party apps from being exploited in the Inter-Component Communication (ICC)-based confused deputy attacks within dynamic permission scenarios. We propose a policy and capability-based access control mechanism, and develop a corresponding security framework with a risk retrainable runtime policy configuration system based on Android middleware. Our framework maintains runtime app states and mediates ICCs among third-party apps according to the policies and dynamic capabilities. Additionally, we propose a sophisticated access decision cache system with proactive and reliable maintenance mechanism, to ensure the efficiency and dependability of decision services. Finally, we analyze the security and complexity theoretically, and then test our approach comprehensively on smartphones. The results demonstrate that our approach provides configurable and effective protections for third-party apps against permission leak vulnerabilities with a small impact on performance and usability.

The contributions of our work are given below.

(1) We propose an Android security framework that provides appropriate and flexible protection to defend third-party apps against confused deputy attacks by monitoring corresponding dynamic states at runtime and enforcing policy and capability-based access control to prevent riskful ICC-based calls among third-party apps.

(2) We provide users with a multi-dimensional and flexible runtime policy configuration system together with a risk mitigation mechanism, which enables users to customize policies for macroscopic capabilities and specific ICCs, and supports the propagation of access control restrictions along with user-defined ICC policies, thereby meeting different security requirements and meanwhile reducing risks induced by inappropriate policies.

(3) We propose a sophisticated access decision cache system with eager maintenance mechanism for efficiency. The cached decisions are proactively updated with the changes of app states and security policies while adhering to the basic capability-based restrictions and risk mitigation mechanism, ensuring the efficiency and dependability of the decision services.

(4) We theoretically analyze the security and complexity of our framework, and implement an experimental prototype to properly evaluate effectiveness and performance.

In the remainder of this paper, we introduce the Android background and describe the app permission leak problem in Section 2. In Section 3, we present the adversary model with corresponding design goals. Section 4 describes the overall design and Section 5 details the implementation. In Section 6, we present theoretical analyses and the experimental evaluation. Section 7 discusses related work, and Section 8 concludes the paper and proposes possible future improvements.

2. Background and motivation

In this section, we provide contextual background on Android and describe how the app permission leaks are exploited in confused deputy attacks that motivate this work.

2.1. Background

Apps and Interactions. Android apps are composed of multiple components and there are four types of components: Activities (associated with corresponding user interfaces), Services (which perform background processing), Content Providers (which act as databases), and Broadcast Receivers (which act as mailboxes for notifications).

Apps are capable of interaction through several communication mechanisms, among which the Intent-based ICC mechanism is the most important one [17,22,23] for reusing and extending app functionalities. It is conducted by the Android middleware and acts as a message passing system, whose messages are specifically structured Intent objects that contain actions for the recipient to perform, e.g., starting app components. Although app interactions are also supported in other ways that circumvent the Android middleware, such as file sharing, in reality, these methods are rarely used in apps [20].

Permission Mechanism. To achieve better security, Android is designed as a privilege-separated system, which runs apps in limited-access, isolated sandboxes and guards sensitive external resources with permissions.

The Android permission mechanism is essentially a mandatory access control system based on permission labels, which checks whether an app has the specified permissions when attempting to access protected resources. Therefore, an app must explicitly declare the required permissions in its AndroidManifest.xml file and receives approval before it can use certain protected resources. Within an app, all components obtain identical permissions.

Android has around 140 standard permissions (labels) for protecting corresponding resources [25] and all the permissions are classified into several categories based on the sensitivities. The most notable categories available to third-party apps are the normal and dangerous permission sets. Dangerous permissions are further grouped according to the functional relationships, e.g., the "READ_SMS" and "RECEIVE_SMS" permissions comprise the "SMS" group. In addition, third-party apps can also be protected by the permission system if their developers apply standard or self-created permissions to the corresponding interfaces of their apps.

The permission mechanism has undergone a significant innovation in Android 6.0 Marshmallow (API level 23). The widely criticized unmodifiable "all or nothing" install-time permission-granting scheme has been discarded and replaced by a partial dynamic runtime approach, providing more flexibility for users to control app functionalities. Under the updated system, the granting of permissions occurs at the first run of an app: required normal permissions are granted by the system automatically, while requests for the dangerous ones are explicitly submitted to the user to make decisions. Note that dangerous permissions are handled at the group level, i.e., the user is informed of the needed permission groups and the user’s decision for a group is applied to all the requested dangerous permissions belonging to it. Another considerable improvement was that a “Settings” interface was provided through which the user can dynamically grant and revoke dangerous permissions of apps at runtime.
2.2. Permission leak of apps

Despite such a sophisticated permission system, permission-protected resources are still vulnerable to privilege escalation attacks. As an example, permission leak vulnerabilities are quite common in Android apps due to Android’s developer-determined app access control mechanism, and are widely exploited in app-tier confused deputy attacks. Permission leaks occur when a privileged but vulnerable app is exploited as a confused deputy by a malicious app with fewer permissions, and executes privileged operations on behalf of the malicious app. Although permission leak vulnerabilities can occur in any app, third-party apps are most vulnerable [7,8,10,12], since they are often written by developers with insufficient security background who implement only casual security checks before making their apps available to users.

Fig. 1 illustrates an example. App $a_i$ is a malware without permissions, while app $a_j$ has a "CALL_PHONE" permission but an unprotected interface with two invokable functions: making phone calls and initiating ICCs to other apps. As a result, app $a_j$, which is accessible to any app (through ICCs), is capable of invoking corresponding resource and apps protected by the “CALL_PHONE” permission. In addition, another app $a_k$ is granted the “SEND_SMS” permission and has an interface which can be invoked to send short messages and is protected by the “CALL_PHONE” permission. In the example, although at first glance the malware $a_i$ can neither use the protected resources nor access app $a_k$, it can exploit the vulnerable app $a_j$ as a confused deputy through the ICC call (step 1), and make it dial a phone number to issue a subscription request for premium-rate services (step 2). Further, it also can leverage app $a_k$ indirectly (step 3) to reply the confirmation text message for the requested subscription (step 4).

Although Android has made significant improvements on its permission system in recent versions, new dynamic features have contributed little to solving this problem due to the single-app-oriented design perspective. Worse, the dynamic permission mechanism inadvertently invalidates many existing approaches since few can adapt to such dynamic, unpredictable, and complex circumstances. A more adaptive and effective solution is badly needed to defend against confused deputy attacks on apps in dynamic scenarios.

3. Adversary model and design goals

3.1. Adversary model

We assume that installed third-party apps are potential adversaries that can launch confused deputy attacks by leveraging privileged but vulnerable third-party apps to use permission-protected resources through Intent-based ICC calls. These adversaries can neither communicate with each other for collusion attacks, nor compromise Android system apps, the Android framework or the underlying kernel to carry out unauthorized privileged actions. Thus, security threats like malware based on root techniques are beyond the scope of this work.

3.2. Design goals

We aim to propose an effective remedy that provides better protection for the permissions of vulnerable third-party apps against malware exploitation. The major design objectives are presented from the perspectives of security, usability, performance, etc.

Security. A satisfactory security mechanism should be capable of defending the permissions of vulnerable apps against malicious exploitation based on ICC-based confused deputy attacks. Our focus is on securing permission-protected resources from being illegally used. Therefore, avoiding apps from improperly sharing legally acquired information [18] is orthogonal to this work.

Adaptability. Our security mechanism must be adaptive and responsive to dynamic environments, taking accurate protective actions rather than rigid underreactions or overreactions to suspected behaviors.

Usability. Although security improvements always come at an expense of usability, a practical security mechanism should be carefully designed to minimize adverse impacts on the normal app functionality.

Configurability. A desirable security approach should be configurable by users so as to meet different security requirements in dynamic scenarios and provide them with flexible control of app functionalities.

Low coupling. A reasonable security extension should be decoupled from existing Android modules to avoid interfering with critical routine services.

Performance. As Android is typically running on devices with limited resources, a practical security solution should be efficient with low overhead and not decrease the effectiveness of the user’s experience.

4. System design

We proposed a configurable and adaptive Android security framework to prevent permission leak vulnerabilities of third-party apps from being exploited by ICC-based confused deputy attacks in dynamic permission scenario. We presented a comprehensive access control mechanism based on capability and policy, and a corresponding protection framework based on the basic security architecture of Android middleware from our previous work [26] with enhanced functionalities such as flexible policy configuration, appropriate risk mitigation, and proactive and dependable
access decision maintenance. Our protection framework monitors runtime app states and mediates ICCs among third-party apps according to capability-based rules and user-defined policies for defending vulnerable apps. Further, it provides users with multidimensional runtime policy configuration for meeting variable security requirements and a corresponding risk mitigation mechanism for reducing risks incurred by user-defined policies. Additionally, for efficiency purpose, the existing access decisions are cached and proactively and reliably maintained with the changes of app states and security policies while adhering to the risk mitigation mechanism.

4.1. Policy and capability-based access control mechanism

To mitigate the app permission leak problem in an efficient and practical manner, we introduce the notion of capability to simplify permissions and propose a pragmatic access control mechanism for ICC requests based on capability and policy.

In general, the capability of an app is defined as its ability to execute privileged operations [27, 28]. Naturally, the permissions granted to an app indicate its capability in Android’s dynamic permission environment. With that in mind, we further make compromises to strike a balance between security and usability. We focus on a small portion of critical permissions available to third-party apps to construct the universal capability set. Besides, we evaluate each capability at the granularity of the permission group, in that, an app possesses an entire permission group even if it only requests and is granted one permission belong that group. This broad evaluation is not only reasonable but also helpful as it can decrease undesirable influences on usability without violating users' intentions for permission management (cf. Section 2.1).

Consequently, we choose all dangerous permission groups and a few frequently abused normal permissions [29–31] as the members of the universal set of capability, denoted by $\mathcal{C}_i$ (see Appendix). Hence, the capability of an app $a_i$ can be represented by $c(a_i) \subseteq \mathcal{C}_i$, which is dynamic at runtime.

On this basis, to provide users with a more general and flexible capability control that satisfies their security requirements in different scenarios, we further define the protected capability set $\mathcal{C}_p \subseteq \mathcal{C}_i$ to support a customizable global capability policy. We design three optional $\mathcal{C}_p$s with nested relationships and allow restricted customization on them (cf. Section 5). Derived from this, the protected capability of an app $a_i$ is represented by $pc(a_i) = c(a_i) \cap \mathcal{C}_p$.

According to the capability, we propose an access control mechanism for ICCs to protect the designated capabilities of all third-party apps from being exploited by malicious apps. Inspired by the classical access control model Biba [32], in our scheme, sender and recipient apps are treated as subjects and objects respectively. Likewise, the notion of protected capability is used in place of the original integrity concept in Biba. Accordingly, we enforce the following rule based on the capability coverage principle to prevent an app with less capability (low integrity) from exploiting the additional capability of another app (high integrity):

**General Rule.** For an ICC request from an app $a_i$ to another app $a_j$, i.e., $req(a_i \rightarrow a_j)$, if and only if the protected capability of the sender $a_i$ covers that of the recipient $a_j$, i.e., $pc(a_i) \subseteq pc(a_j)$, then that ICC request is permitted; otherwise, it is rejected.

Notice that the capability is actually represented by the UID of the app in our case, while the ICCs are considered to be unidirectional according to many typical solutions [8, 19, 20], to strike a balance between security and practical app usability. To provide users with flexible control of app functionalities, additional **Special Rule** is also imposed for supporting user-defined ICC policies.

**Special Rule.** Regardless of the **General Rule**, an ICC request $req(a_i \rightarrow a_j)$ should be permitted if there exists a special allowing policy for it. Contrarily, it should be rejected if there exists an additional denying policy for it.

Since the user-dependent **Special Rule** has a higher priority, its allowing policy that violates the **General Rule** may lead to new vulnerabilities. In particular, a malicious app that can reach the sender app belonging to such a policy is able to exploit the sender app as the deputy, and indirectly reaches the recipient app belonging to that policy with the aid of the **Special Rule**. Therefore, to avoid the proliferation of protected capability through the corresponding sender app, an additional **Supplemental Rule** must be set as a complementary control of the risks induced by the inappropriate configurations of users within the policy range.

As an app may exploit additional capabilities via the allowing ICC policies, we transform the allowing ICC policies into a graph to discover all the capabilities available to an app through the paths on the graph. We regard these as the susceptible capability of the app which should be analyzed when establishing new ICC channels to it, lest such capabilities are exploited by other apps. Then, we formally define the susceptible capability set of an app as the combination of the capability set of all of its reachable apps in the graph, denoted by $sc(a_i)$. Here, $sc(a_i) = c(a_i) \cup \bigcup_{a_j \in \mathcal{R}(a_i)} c(a_j)$, where $\mathcal{R}(a_i)$ is the set of all of the reachable apps of $a_i$ through paths in the graph. Similarly, the susceptible protected capability of an app $a_i$ is denoted by $psc(a_i) = sc(a_i) \cap \mathcal{C}_p$. Finally, we present the **Supplemental Rule** as follows:

**Supplemental Rule.** If an app $a_i$ is the sender app of any allowing policy, for an ICC request $req(a_i \rightarrow a_j)$, if $psc(a_i) \subseteq pc(a_j)$, then that ICC request is permitted; otherwise, it is rejected.

Note that the **Supplemental Rule** is prior to the **General Rule** but subsequent to the **Special Rule**.

4.2. Architecture and workflows

The architecture of our framework is depicted in Fig. 2. We partly draw on the basic architecture of our previous Android security extension based on Generalized Framework for Access Control (GFAC) [26] and construct a new extended framework on Android middleware with new facilities including a multidimensional runtime-configurable policy module, a risk mitigation mechanism and a proactive and dependable decision cache system, so as to acquire the adaptive, customized, effective, and efficient protection in the dynamic environments.

In this framework, the original ActivityManager (AM) was slightly modified to invoke our Inspector to further inspect those granted ICC requests among third-party apps after routine checks. The PackageManagerService (PMS) was also altered to provide the StateMaintainer with latest states of third-party apps. The Inspector is the control center and contains a DecisionMaker for decision making and maintenance, and an AccessVectorCache (AVC) for caching previous ICC decisions of third-party apps. The AVC is proactively updated with the changes in app states and security policies while adhering to the risk mitigation mechanism, ensuring the efficiency and dependability of decision services. The StateMaintainer collects runtime app states and converts them into...
capability sequences based on its embedded hashmap CapabilityDictionary. It keeps the processed app states in its CapabilityState (CS) and maintains additional app states affected by current policies in the SusceptibleCapabilityState (SCS) for decision supports. The Configurator provides users with runtime policy configuration through a “Policy Setting” app and maintains security policies in its Policy database. It additionally maintains an AllowingPolicyGraph for risk analysis.

Fig. 2 illustrates the prime steps of three major workflows of our framework to show how it works.

**ICC Inspection Flow.** The AM is triggered by an ICC request between apps and performs original permission validation (step 1). Once the validation passes, the altered AM checks the information of the involved apps obtained from the PMS (step 2), and calls the Inspector for further analysis when the ICC occurs between third-party apps (step 3). The Inspector invokes its submodule DecisionMaker to search for existing result of a certain ICC in the AVC (step 4). If the cache is missed, the DecisionMaker queries the Configurator for the special ICC policy for the involved apps (step 5). If no ICC policy can be applied, the DecisionMaker continues to ask the StateMaintainer to provide the protected capability sequences of the involved apps (or the susceptible one for the recipient if existed), and then makes a new decision based on the embedded rules (step 6). The fresh decision is stored in the AVC by the DecisionMaker (step 7) and is returned to the AM by the Inspector (step 8). Finally, the AM permits or rejects that ICC request accordingly (step 9). Note that the Inspector will check state synchronization locks which can be set by other modules in its steps and will wait until locks release to get rid of state inconsistent risks.

**State-oriented Maintenance Flow.** The modified PMS monitors the state changes of third-party apps (i.e., permission changes and (un)installation), and informs the StateMaintainer of the latest information of the changed app (UID, permissions and packagelist) extracted from the mPackages. This allows the StateMaintainer to work out the new capability sequence of the app according to its embedded hashmap CapabilityDictionary and then updates the CapabilityState (CS) accordingly, with the previous information temporarily kept for future comparison (step a). Subsequently, the StateMaintainer informs the Configurator of the app information (step b). If the Configurator discovers such an app in its AllowingPolicyGraph, it returns the AllowingPolicyGraph to the StateMaintainer for reference. For a normal app uninstallation, the Configurator first removes the ICC policies related to the uninstalled app from its Policy database and then updates the AllowingPolicyGraph if needed. When the graph is updated, the Configurator returns the AllowingPolicyGraph to the StateMaintainer as well. Additionally, the capability policy will be returned to the StateMaintainer (step c). When receiving the AllowingPolicyGraph, the StateMaintainer works out the new susceptible capability sequences of apps existed in the graph according to the reachability analysis, and then updates them to its SusceptibleCapabilityState (SCS), with the previous information temporarily kept for future comparison. Subsequently, the StateMaintainer filters out the app(s) possessing new protected capability sequence(s) (or susceptible protected capability sequence(s)) by applying the current capability policy, and then informs the DecisionMaker of these apps (step d). By collecting the ICC policies from the Configurator (step e), the DecisionMaker finally updates the existing decisions related to the affected apps in the AVC according to the embedded rules (steps f).

---

4 In the case of sharedUID, (un)installation are equivalent to permission revoking/granting from the permission perspective (cf. Section 5).

5 There are two exceptions: one is the normal installation ending at step a. Another is the normal uninstallation in which the StateMaintainer should notify the
Policy-oriented Maintenance Flow. The Configurator receives the new configuration information from the user through the “Policy Setting” app. And then it updates its Policy database and AllowingPolicyGraph (if needed) accordingly (step i). Next, it provides the StateMaintainer with the new AllowingPolicyGraph so that the StateMaintainer can update its SCS accordingly based on the built-in reachability analysis algorithm as well as the information in the CS (step ii). Subsequently, both the Configurator and the StateMaintainer inform the DecisionMaker with the new configuration information in the Policy database and the apps with new susceptible protected capability sequences respectively (step iii–iv). Finally, the DecisionMaker updates the affected decision caches in the AVC according to the embedded rules (step v).\(^6\)

Note that in each flow, modules like the StateMaintainer and Configurator will check and use the possessed queued tokens and synchronization locks in their steps to work cooperatively.

5. Implementation

In this section, we expand the description of our implementation, which is based on Android 6.0 Marshmallow source code (branch 6.0.1_r62) from the Android Open-Source Project (AOSP).

Generally, we not only made modifications to existing Android middleware but also developed new modules to implement our prototype.

5.1. Modifications on existing modules

**ActivityManager.** The ActivityManager (AM) is a core Android module that plays several important roles, including the ICC interceptor and the permission checker. For runtime ICC requests, the AM analyzes whether the initiator has the permissions required by the recipient, and then enforces mandatory access control accordingly.

We have made slight changes to the AM so as to direct the original control flow for the ICCs among third-party apps to our Inspector module for further analysis. In detail, we have tweaked the codes of the checkComponentPermission() function, enhanced it with the ability to screen out ICC requests among third-party apps based on a custom function isThirdPartyAppUID(ApplicationInfo.FLAG_SYSTEM) and to call the inspector with the sender-recipient UID pair encapsulated in a JSON object for additional inspection when the original function returns “granted” for approving the certain ICC request.

**PackageManagerService.** Since the PackageManagerService (PMS) is a critical runtime service that handles app-related events (such as (un)installation and setting permissions) and maintains the runtime information of apps (UIDs, granted permissions, paths, etc.) in its mPackages object, we have made minor edits to it to collect the state changes of third-party apps for reporting to the StateMaintainer.

Practically, we selected two pairs of functions for observation. We modified the function pair that handle runtime permissions updating on the mPackages, i.e., grantRuntimePermission() and revokeRuntimePermission(), to detect permission-changing events of third-party apps. We also extended the installNewPackageLI() and removePackageLPW() functions, which update app information to the mPackages when (un)installing apps, to detect the occurrences of third-party app (un)installations.

The detailed information of the changing apps are collected from the mPackages and then delivered to the StateMaintainer as JSON objects through a BlockingQueue.

5.2. Developments of new modules

Since the existing system modules such as AM and PMS are extremely important and sophisticated as they are in charge of the burdensome and critical routine tasks, for decoupling reasons, we implemented the major functional components of our framework as sub-service modules of a new independent system process security_server (similar to the core system process system_server), to reduce the potential affects on existing critical system services. Besides, the modules of the security_server use possessive queued tokens and synchronization locks to work cooperatively.

**Inspector.** The Inspector is core security module that is mainly responsible for mediating ICC requests forwarded by the AM to defend third-party apps against malicious ICC-based invoking in confused deputy attacks. It consists of a core logic DecisionMaker and a decision cache AccessVectorCache (AVC).

- **DecisionMaker.** The DecisionMaker makes access control decisions for ICC requests and dynamically maintains the AVC based on the policy and capability-based restrictions and risk mitigation mechanism to ensure the efficiency and dependability of the decision services.

  In order to return the access decision for a certain ICC request, it first retrieves the previous decision result in the AVC by the shortened index (suid, ruid) obtained from the UID pair of the ICC. If no such a decision exists (i.e., it returns “unidentified”), it initiates alternative branch procedures. It refers to the Configurator for the special ICC policy for the current ICC request with the ordered UID pair as the hash index. If such a policy exists, our DecisionMaker writes the decision, i.e., “granted” or “denied”, to corresponding index in the AVC and leaves it to the Inspector to return. If no ICC policy can be applied, further analysis is needed to make the final decision. Specifically, it queries the StateMaintainer to acquire the protected capability sequences of the ordered UID pair (the susceptible protected capability sequence of the recipient will be returned if exists). According to the returned sequences, it makes the decision based on the General Rule: it performs a bitwise-OR comparison between the two sequences, to determine whether the sender’s cover the recipient’s. Then, it inserts the fresh decision in the AVC for future use and provides it to the Inspector which will eventually return the decision to the AM.

Apart from its primary inspection responsibility, the DecisionMaker is invoked by either the StateMaintainer or the Configurator when the app state and security policy changes, to keep the AVC compliant with the basic capability-based restrictions and risk mitigation mechanism, thereby ensuring the efficiency and dependability of the decision services. Upon receiving notifications, our DecisionMaker parses context such as the UID, ICC policies and the maintenance signal. If canceled ICC policies are contained in notifications, the DecisionMaker prioritizes them and resets the corresponding decisions to an initial value “unidentified”. If UIDs exist in the context, the DecisionMaker locates the decision caches that might have been affected in the row and column of a certain UID in the AVC according to the maintenance signal (see Algorithm 1), and then proactively updates their results by applying the embedded rules on the corresponding sequences of index pairs (or sets “unidentified” in the case of normal app uninstallation). For a certain UID, the StateMaintainer returns both the protected capability sequence and the susceptible one\(^7\) for reference. The valid ICC policies are applied after all UIDs being handled to avoid inconsistencies. For handling the modification of capability policies, the DecisionMaker resets the entire AVC to its initial state, i.e., all items are restored to “unidentified” as most cached decisions may not comply with the new policies.

---

\(^6\) The capability policy reconfiguration is an exception that involves no interactions with the StateMaintainer (since skip step ii–iii), and thus the Configurator simply asks the DecisionMaker to reset the AVC to its initial state, i.e., all the items are set to “unidentified” (step iv–v).

\(^7\) The susceptible protected capability sequence is used when an app with that UID acts as the recipient in a pair.
Note that the inspector checks synchronization locks which can be set by other modules in each atomic procedure and will wait until locks release for cooperative work, and the DecisionMaker handles updating events transaction-by-transaction with the support of synchronization technique, and performs the one-by-one updates for apps in the AVC, and blocks access to the AVC with an exclusive lock during updates, to get rid of state inconsistent risks.

- AVC. The AVC is an efficient access decision cache that is private to the inspector and is persistent across reboots based on serialization technique. Since ICCs frequently occur with relatively fixed relationships while user-determined settings are infrequent and unpredictable, the AVC is implemented as an "App UID × App UID" adjacency matrix (2-dimensional byte array implementation) to cache the decision for each possible ICC (presented by UID pair) because this implementation is fairly responsive and efficient for access, but less so for maintenance. In the AVC, the array subscript [su](ruid) is used as the decision index, where suid (or ruid) = sender UID (or recipient UID) — 10 000. The values of array elements are the decision results for ICCs, namely, the value "0" (denied), the value "1" (granted) and default value "2" (unidentified). Since the AVC only accommodates third-party apps installed on the devices\(^8\) with an average number of about 100 due to users’ actual requirements and limitations of devices according to a statistical survey [33], we set the practical array size to 400 (with a corresponding total size of 160 KB), which is large enough for extreme cases (this number can cover 99.9% of practical cases based on the five-sigma effect of normal distribution [34]) and efficient for maintenance with a low memory impact.

Algorithm 1: The Kernel of Proactive AVC Maintenance Algorithm

Input: the UIDs and corresponding maintenance signals sig's, valid and canceled ICC policies

Output: the updated AVC matrix

1. for all inputed u_i do
2. if (sig == cap escalation) then
3. for each decision D[u_i, u_j] == granted do
4. if (pc(u_i) ∪ pc(u_j)) ⊆ pc(u_i) then
5. D[u_i, u_j] = denied
6. end if
7. end for
8. for each decision D[u_i, u_j] == denied do
9. if (pc(u_i) ⊆ pc(u_j)) then
10. D[u_i, u_j] = granted
11. end if
12. end for
13. else if (sig == cap_reduction) then
14. for each decision D[u_i, u_j] == granted do
15. if (pc(u_i) ∖ pc(u_j)) then
16. D[u_i, u_j] = denied
17. end if
18. end for
19. for each decision D[u_i, u_j] == denied do
20. if ((pc(u_i) ∪ pc(u_j)) ⊆ pc(u_j)) then
21. D[u_i, u_j] = granted
22. end if
23. end for
24. end if
25. end for
26. for all valid ICC policies do
27. if (policy(u_i ⇒ u_j)) then
28. D[u_i, u_j] = granted
29. else if (policy(u_i ⇔ u_j)) then
30. D[u_i, u_j] = denied
31. end if
32. end for
33. for canceled ICC policies policy(u_i ⇒ u_j) do
34. D[u_i, u_j] = unidentified
35. end for
36. return updated AVC

For achieving better efficiency, the AVC maintenance algorithm performed by the DecisionMaker is further optimized for different modification scenarios (see Algorithm 1). As for changes that enlarge the protected capability set of the app a_i, typically, granting a permission group, we only need to recheck the "denied" caches for the ICCs initiated by a_i and the "granted" results for the ICCs ending at a_i, because more capability may make app a_i capable of accessing apps previously inaccessible, while may block ICCs directed to a_i from apps which were previously eligible to access it. In addition, the increase of the susceptible protected capability set can be addressed in a similar way, but since the susceptible protected capability set is not the actual capability that the app owns, only "granted" results for the ICCs ending at a_i are reviewed for security reasons. Similarly, for the capability-reducing cases, only "granted" decisions for the ICCs initiated by a_i and "denied" decisions for the ICCs ending at a_i are reviewed. For the susceptible protected capability set, only the "denied" results for the ICCs ending at a_i are reviewed. To avoid additional overhead, the following details were considered for optimization. Only existing decisions are proactively updated, because apps usually have stable communication relationships, i.e., the ones do not communicate with each other before may never do that in the future. Moreover, instead of checking ICC policies during the review of each UID pair, we handle the ICC policies as a batch at the end, which reduces the overhead while maintaining consistency.

StateMaintainer. The StateMaintainer acts as a manager and courier of the dynamic states of apps. It maintains the CS and SCS hashmaps to retain the most recent app state, and is responsible for instructing the DecisionMaker to update the AVC and informing the Configurator to update the Policy and AllowPolicyGraph when the state changes.

The CS and SCS are two private hashmaps that store the mappings between the UIDs of third-party apps to their actual corresponding capability sequences and susceptible ones, respectively. The UID is a distinct identifier of an app\(^9\) and the capability sequences (and the susceptible ones) are fixed-size strings. The value of each bit in the string indicates the enabling state of a certain capability, specifically, "1" denotes "granted" and "0" denotes "denied". The CS and SCS are initialized upon first boot by the StateMaintainer according to the information from PMS and Configurator, and are persistent across reboots through the use of serialization.

The StateMaintainer receives and analyzes the latest app state information (i.e., UID, permissions, packageList) from the PMS. If an app is uninstalled (i.e., empty packageList), the StateMaintainer removes the corresponding information of that UID from the CS and SCS, informs the Configurator of that UID to clean the related ICC policies and notifies the DecisionMaker to reset related decisions. Otherwise, the StateMaintainer works out a fresh capability sequence for that UID based on the granted permissions according to the CapabilityDictionary, which is a constant hashmap that maps a particular permission name to a preset bit position of the capability sequence.\(^10\) Accordingly, the StateMaintainer generates a new capability sequence with value "1" in bit positions to which the granted permissions are mapped and zeros for the rest. Subsequently, the StateMaintainer attempts to find the capability sequence of the UID in the CS for comparison. If no record exists, the StateMaintainer inserts a fresh hash pair (UID, capability sequence) into the CS for the new app and returns, because the new app is free of attachments.

The StateMaintainer does nothing special if the capability sequence remains constant. This case arises during the (un)installation of "sharedUID" instances when the capability of

---

8 Idealistically, Android is capable of accommodating at most 10000 third-party apps with constant app UIDs ∈ [10 000,19 999] during the apps’ lifetime, and UIDs used by uninstalled apps would be collected and reallocated efficiently.

9 One special case is "sharedUID", i.e., several apps with same "sharedUID" tags and same signatures run in the same sandbox. Indeed, they are treated as being the same app, with the same UID and merged permissions.

10 Permissions within a group are mapped to the same bit position.
an (un)installed app is covered by that of others under the same UID. However, if the capability sequence changes, after writing the new one to the CS, the StateMaintainer will also inform the Configurator of the UID to determine whether this change will affect the susceptible capability sequences of other apps. Once the AllowingPolicyGraph is returned by the Configurator, which means the UID is found in current AllowingPolicyGraph and thus other apps in the graph may be affected, the StateMaintainer must filter out the affected apps and updates their susceptible capability sequences in the SCS. The StateMaintainer implements a capability propagation algorithm for apps in the AllowingPolicyGraph to determine their susceptible capability sequences that are used in decision-making process to avoid potential risks. The algorithm traverses the graph, via a depth-first search, and merges the capability sequences of apps obtained from the CS with those of apps that are reachable along the reverse paths. In addition, the susceptible capability sequences of apps can be affected when the AllowingPolicyGraph changes due to user's configuration actions or app uninstallation as mentioned above. In this case, the StateMaintainer executes a similar procedure to determine the affected apps as well as their susceptible capability sequences.

Based on the changes of capability sequences, the StateMaintainer uses two inductive and unified maintenance signals covering multiple modification cases to instruct the DecisionMaker in the AVC update. Particularly, it selects an “cap escalation” signal when the new (susceptible) capability sequence covers the previous one and chooses “cap reduction” in the opposite situation. Eventually, the StateMaintainer informs the DecisionMaker to update the AVC through a JSON message that encapsulates the UIDs and corresponding maintenance signal.

Note that the StateMaintainer handles the modification events transaction by transaction, and locks the CS and SCS during the update.

Configurator. The Configurator is a runtime service that provides users with both the macroscopic and microscopic dimensions of security configuration through a user-interface implemented as an irrepressible “Policy Setting” system app with no public interface and sensitive permissions. The app is protected by system signature and any ICC call to it from other apps would be denied directly by the ActivityManager module.

The Configurator implements a configurable policy repository Policy that contains two kinds of policies: capability type and ICC type.

The configurable global capability policy is used to define the current protected capability set with specific members in the $C_A$, for supporting flexible capability protection from a general perspective. It is implemented as an exclusive fixed-size Boolean bitmask with the same size as the capability sequence. According to the risks of permissions if being maliciously leveraged and the permission protection levels specified by Google, the Configurator provides users with three increasing protection levels corresponding to three incremental capability policies. The primary protection level (level 1) contains capability members corresponding to some crucial “dangerous” permission groups that could lead to irreversible negative consequences such as financial loss, corruption of user's stored data, or critical private data leaks. The second protection level includes capability members corresponding to all the “dangerous” permission groups. Apart from the capability members in level 2, the highest protection level (level 3) further covers more capability members corresponding to several “normal” permissions which are frequently leveraged by malicious apps [30,31], i.e., level 3 covers all members in $C_A$ for fulfilling high security requirements. (cf. Appendix). In our implementation, the capability members in level 1 are mandatory. Users are allowed to add or remove some capability members belonging to the $C_A$ with whose minimum level higher than the current selected protection level for customized purposes. For example, under protection level 2, users can add or remove capability members with the minimum level tag 3, rather than level tag 1 or 2.

An ICC policy is set for a certain ICC between an app pair to provide better functional control of apps. Each ICC policy is implemented as a mapping from an ordered UID pair (senderUid, recipientUid) to a “Boolean” decision result, with the value “true’’ and “false’’ representing “granted’’ and “denied’’ respectively. Users can insert, delete, and modify ICC policies as needed. The Configurator also maintains an allowing ICC policy graph AllowingPolicyGraph complying with existing user-allowed ICC policies to help detect possible propagations of capabilities of the connected apps in the graph for risk mitigation. It is a directed graph implemented by an orthogonal list supported by the open-source JUNG graph library (version 2.0.1). Both the Policy and the AllowingPolicyGraph are stored in XML and persistent across reboots.

When the capability policy is modified by the user, the Configurator writes it to the Policy and notifies the DecisionMaker to restore the AVC with a JSON message. When the ICC policy is changed, the Configurator updates the Policy and the AllowingPolicyGraph (if needed), and then notifies the StateMaintainer with a JSON message encapsulated with a synchronous serial number (syn) and the AllowingPolicyGraph when the graph is changed, to assist the SCS update. Besides, the DecisionMaker will be informed of the changed policy, or the entire ICC policies together with the same syn for the StateMaintainer if the graph is changed. Additionally, the Configurator will also be triggered by the StateMaintainer as the latter needs the AllowingPolicyGraph when capability changes of a certain app are detected, to update the SCS. Besides, when an app is normally uninstalled, the StateMaintainer will inform the Configurator of the related UID, so that the Configurator removes all the ICC policies related to this app UID and updates the AllowingPolicyGraph accordingly. The same procedure is performed by the Configurator for handling ICC policy modification.

6. Analysis and evaluation

6.1. Theoretical analyses

In this section, we analyze the security of our framework under the assumption given in Section 3.1 and analyze the complexity of the major processes.

6.1.1. Security

We assume that a malicious third-party app $a_i$ running alone in a sandbox does not possess a dangerous permission group $p_e \in C_e$, namely, its protected capability set $pc(a_i)$ (as well as its capability set $c(a_i)$) does not contain the member $p_e (p_e \notin pc(a_i))$. As the Linux kernel and Android framework are trusted, the malicious app $a_i$ can neither tamper with its information (such as granted permissions) in mPackages, nor defeat the permission system to access resources protected by $p_e$ directly. Similarly, it can never compromise facilities of our framework to exploit vulnerable apps. We discuss security in the following situations.

**Situation 1**: No ICC policy exists in the system.

---

11 Only one ICC policy exists for a certain UID pair, i.e., “allowed” or “denied”, and previous settings will be overwritten by new ones so as to avoid policy configuration conflicts.


13 If an ICC policy is canceled, the corresponding decision cache in the AVC is reset to “unidentified”.
Consider a vulnerable third-party app $a_i$ with its protected capability set $pc(a_i)$ containing the permission group $p_k$ ($p_k \in pc(a_i)$). It has a public interface that can be invoked by ICC messages to perform privileged operations related to $p_k$ on behalf of others. When malware $a_i$ attempts to directly send a malicious ICC message to $a_i$, our framework is invoked to analyze whether this ICC request should be permitted. Since $p_k \in pc(a_i)$ and $p_k \notin pc(a_j)$, $pc(a_j) \subseteq pc(a_i)$. Thus, this ICC request is denied according to the General Rule. Furthermore, malware $a_i$ may attempt to invoke $a_j$ through some other third-party apps. For example, it tries to form an ICC call chain to $a_i$ like $(a_i \rightarrow a_{i-1} \rightarrow \ldots \rightarrow a_0 \rightarrow a_j)$. Based on our General Rule, for any two adjacent apps in the chain, we have $pc($successor$) \subseteq pc($precursor$)$. Then a conclusion can be drawn that if $pc(a_j) \supseteq pc(a_i) \supseteq \ldots \supseteq pc(a_{i-3}) \supseteq pc(a_k)$ since the subset relation is transitive, which is in contradiction to the fact that $pc(a_i) \not\subseteq pc(a_k)$ as $p_k \in pc(a_k)$ and $p_k \notin pc(a_i)$. Therefore, malware $a_i$ that lacks $p_k \in C_F$ can neither directly exploit app $a_j$ which possesses $p_k \in C_F$ nor do so through other apps in this situation.

When taking user-defined ICC policies into consideration, the additional denying ICC policies will not increase risk. Thus, only allowing ICC policies are given consideration. We further analyze security under two conditions:

**TheSupplementalRule.** An allowing ICC policy $pc(a_i) \rightarrow a_j$ exists that enables a third-party app $a_j$ with $pc(a_j)$ to access another third-party app $a_i$ with $pc(a_i)$ (if needed) requiring $pc(a_j) \subseteq pc(a_i)$, and the protected capability set of app $a_j$ is covered by that of the malicious app $a_i$, i.e., $pc(a_i) \subseteq pc(a_j)$.

According to the analysis above, malware $a_i$ is unable to exploit an app that possesses $p_k$. Although $pc(a_i) \subseteq pc(a_k)$ of the $a_k$, the susceptible protected capability of $a_k$ is $pc(a_k) = pc(a_i) \cup pc(a_k)$ and thus $p_k \in pc(a_k)$. Therefore, malware $a_i$ cannot access app $a_k$ to exploit $p_k$ indirectly as $pc(a_k) \not\subseteq pc(a_i)$. Similarly, malware $a_j$ cannot find a path consisting of third-party apps $a_s$ with $p_k \not\in pc(a_k)$ to access app $a_s$. Consequently, our approach avoids the proliferation of protected capability members through the corresponding sender app of an allowing ICC policy.

**Situation 3:** An allowing ICC policy $pc(a_i) \rightarrow a_j$ exists that enables malware $a_j$ to access another third-party app $a_i$ with $pc(a_i)$.

In this case, although malware $a_j$ can exploit protected capability members in $pc(a_i)$, or worse, the members in $pc(a_k)$ indirectly if there exist additional allowing ICC policies for app $a_k$ (as the sender), malware $a_j$ cannot gain additional privileges from apps not in its allowing policy paths based on the previous analyses. Therefore, our approach controls the risks induced by inappropriate configurations of users within the range of allowing ICC policies.

### 6.1.2. Complexity

We analyze complexity of our three major workflows to demonstrate the efficiency of our framework.

For ICC inspection, the function $IsThirdPartyApp()$ requires $O(1)$. Then determination of the decision index for an ICC request and the access of that decision in the matrix $AVC$ run in $O(1)$. If the cache is missed, attempting to acquire the corresponding ICC policy through the hashmap $Policy$ requires $O(1)$, and if such a policy does not exist, acquiring the corresponding capability sequences through hasmaps $CS$ and $SCS$ requires $O(1)$, while making a new decision by Boolean comparison and updating it to the matrix $AVC$ also run in $O(1)$. Thus, the ICC inspection process runs in $O(1)$.

For state changes, we assume that there is an $n \times n$ matrix $AVC$, m allowing ICC policies in the $Pokey$, and the modified app is granted with $k$ permissions. Then we require $O(k)$ to obtain a new capability sequence of the modified app, and updating it in hashmap $CS$ runs in $O(1)$. If the Configurator discovers such an app in $AllowingPolicyGraph$ or in the case of uninstallation, a further depth-first search in $AllowingPolicyGraph$ which consists of at most $m$ edges (since there exist $m$ policies) and no more than $2m$ nodes runs in $O(m + 2m)$, and will find $p$ (assumed) affected apps which need to be updated in the $AVC$, where $p < 2m$ and $p < n$. For the decision update in the $AVC$, determining the update mode runs in $O(1)$, and the for-each update loop executes $O(2n)$ times to traverse every item in the row and column corresponding to the app. The loop body requires $O(1)$ to check the value of an item to preselect suspect decisions, and further costs $O(1)$ for decision review and necessary replacement in the $AVC$. The whole update process of such apps in $AVC$ requires $O((p + 1)n)$. Hence the whole state-oriented maintenance process runs in $O(k + pn)$.

For policy-related maintenance flow, updating the Policy and $AllowingPolicyGraph$ (if needed) require $O(1)$. If the policy is a new denying ICC policy, no further operation is needed. Otherwise, the StateMaintainer requires $O(m + 2m)$, namely, $O(m)$, to traverse the $AllowingPolicyGraph$ consisting of at most $m$ edges and $2m$ nodes to determine all affected apps and obtain new capability sequences via a depth-first search. Assume that $p$ affected apps are found in the depth-first search for the $AllowingPolicyGraph$, the $SCS$ updating process runs in $O(p)$. In the $AVC$ updating process, the Decisionmaker also needs to update the decisions related to no more than $p$ apps. And for each app, $O(n)$ is required for updating corresponding rows and columns in the $AVC$. Therefore, in the usual case, the entire updating process will run in $O(mn)$. Therefore, the totally policy-related maintenance runs in $O(m + p + mn)$, i.e., $O(m + pn)$.

### 6.2. Experimental evaluation

In this section, we evaluate our solution in terms of functional effectiveness and performance. We selected the Android 6.0 Marshmallow source code (branch 6.0.1..r62) to build a custom Android OS and chose the Nexus 5X smartphones (Snapdragon™ 808 1.8 GHz processor, Adreno™ 418 GPU, 2 GB LPDDR3 RAM, 32 GB Internal Storage) as the experimental devices.

#### 6.2.1. Functional effectiveness

To verify the effectiveness of our prototype, we used 10 apps consisting of 5 customized malicious apps with no permissions and 5 vulnerable samples, 2 of which exist in the real world [12] and the others are specially developed. These samples have certain "dangerous" permissions which can map to certain members in the selected protected capability set to implement confused deputy attacks.

In experiments, each malicious app automatically starts up at system boot and intermittently attempts to ask a random vulnerable app to execute a privileged operation through a maliciously crafted ICC request with one of 4 Intent types. Note that these attacking experiments succeeded on a stock Android system. To more closely simulate the real environment, 50 additional benign apps (downloaded from official sources and detected by the McAfee Mobile Security v4.9) were also installed on our custom Android OS together with the test samples above.

We tested our prototype under the three protection levels. During experiments, we randomly opened and used apps, occasionally modified their permissions, performed (un)installations and set certain ICC policies. All ICC access control results were monitored and logged in files in time order.

According to the access control log files, our framework effectively thwarted every really harmful attack which aims at illegally leveraging certain permissions (capabilities) in the corresponding protection level and precisely avoided overreactions for the previously blocked ones when corresponding states became not vulnerable for keeping app functionality.

Detailedly, we discuss several typical cases with the adoption of protection level 1 to demonstrate our functional effectiveness.\footnote{Explicit Intent, Implicit Intent, Pending Intent and Sticky Intent.}

\footnote{Similar phenomena were observed in other protection level tests.}
Fig. 3. The permission states of the sample apps prepared for testing.

Case 1. The app “test2” (UID: 10065) was designed as a benign but vulnerable sample with the “Phone” and “SMS” permission groups (see Fig. 3(a)), while malicious app “test1” (UID: 10052) that requests no permissions (like Fig. 3(b)) was designed to intermittently send “test1” malicious intents to make it dial a certain number. Note that both the “Phone” and “SMS” belong to the protected capability set in this case. According to our observations, no phone call was initiated by app “test2”, and a declined ICC request from “test1” to “test2” was found in the log (see Fig. 4(a)). Thus, our framework effectively protected the permissions of “test1” from being utilized by “test2”.

Case 2. After Case 1, we revoked all permission groups of app “test2” (see Fig. 3(c)). Then, the following ICC request from malicious app “test1” to benign app “test2” was permitted (see Fig. 4(b)) as our approach focuses only on the actual permissions of apps for the sake of maintaining app usability. Although the malicious “test1” succeeded in sending a malicious ICC message to the vulnerable app “test2”, it did not reach its attacking goal as app “test2” was ineligible to perform the corresponding privileged operation at that moment.

Case 3. The apps “instagram” (UID: 10051) and “facebook” (UID: 10068) were downloaded from Google Play and were granted with all their declared permissions. The “instagram” was used to establish an ICC channel to “facebook” successfully (see Fig. 5(a)). After inserting a denial ICC policy which blocked the ICC from “instagram” to “facebook” through the Setting UI (see Fig. 6), we discovered that such a policy was successfully applied, as depicted in Fig. 5(a).

According to testing results above, we found that our framework successfully blocked every really dangerous malicious ICC attempt and precisely approved some of the previously blocked ones when corresponding states (e.g., permissions, policies) changed and became not vulnerable to avoid affecting app functionality. The primary disadvantage of our approach was also revealed by the experiments. According to the analysis of the log files, a number of benign ICC requests among third-party apps were mistakenly declined, incurring false positive rates of 9.7% (128 ICs / 1,372 ICs), 13.5% (197 ICs / 1,473 ICs), and 20.3% (189 ICs / 931 ICs) when we adopted the protection level 1, level 2, and level 3 respectively. These results reflected an acceptable tradeoff between security and app usability. Especially, for meeting stronger security requirements, the highest protection level (level 3) test output a slightly high false positive rate as the framework provided additional protections for capability members corresponding to some permissions that are defined as low risk “normal” by Google but practically and frequently leveraged in malicious attacks for annoying or misleading users [30,31].

In conclusion, as a precise security framework, our framework successfully prevented the capabilities of vulnerable apps from being exploited by malicious apps via ICC-based confused deputy attacks. The approach is adaptive and responsive to runtime app changes and is able to take more reasonable and appropriate protective actions rather than underreactions or overreactions for handling suspect ICCs, maintaining app usability and not sacrificing security. Moreover, it provides users with flexible and effective app functionality management through a configurable policy system.

6.2.2. Performance

To measure the performance of our implementation, we performed not only a general assessment with benchmark tools but also specific evaluations for the time cost of ICC inspection and resource consumption. We deployed both the stock Android and the custom Android in Nexus 5X devices with the 60 apps mentioned above. For the specific evaluations, we performed the series of operations for two hours as had been done in previous experiments.

General Performance. We evaluated the overall performance of our prototype through two popular benchmark tools, i.e., AntuTu (v6.1.5 46579) and Geekbench (v3.3.2). The performance evaluation score given by the benchmark tool originally represents the relative performance of devices. Since this score is directly affected by the performance of operating system running on the testing device, it is also used to reflect the overall performance of the operating system in practice (cf. Reference [35]).

Table 1 compares the performance scores of our custom Android with the stock Android under these two benchmarks and demonstrates negligible overhead ratios of less than 1.5%.
Time Cost of ICC Inspection. Since ICC inspection was the most frequent and primary routine task of our prototype, we evaluated its time cost to measure performance. Additionally, we also implemented a prototype without the AVC for comparison, in order to verify its effect on performance.

According to Table 2, our framework equipped with the AVC incurred a time overhead of 1.51 ms on average, compared with the 0.16 ms for stock Android, and was far more efficient than the system without the AVC, indicating the effectiveness of the decision caching system as well as the proactive maintenance mechanism.

Runtime CPU & RAM Costs. As widely used performance indicators, the CPU and RAM costs of our solution were also evaluated. We connected a personal computer (PC) powered by Ubuntu 12.04 to the smartphones during experiments, issuing the Android Debug Bridge (ADB) commands to monitor the continuous outputs.
of corresponding information. According to the logs on the PC, we observed that our prototype consumes about 87.3M (3.7%) more RAM and raised the CPU utilization from 24.6% to 29.3% on average compared with the stock Android under otherwise identical conditions, figures that are readily acceptable to most current devices.

### 7. Related work

Since permission leak vulnerabilities exist in a large number of Android apps and have been widely exploited in app-tier confused deputy attacks, they have attracted wide attentions from the industrial and academic communities. So far, many solutions have been proposed and can be broadly classified into static and dynamic categories.

Initial studies and subsequent follow-ups have concentrated on leveraging static analysis techniques to detect the potential vulnerability and maliciousness of permission leaks through the analysis of app codes.

Typically, ComDroid [36] is an app vulnerability detection tool that decompiles apps and statically analyzes the Intent-related codes of app components, identifying the communication vulnerabilities of individual apps, including implicit Intents and exposed interfaces without adequate permission protections. Woodpecker [28] is another static analyzer that inspects the control flows of disassembled apps to discover the access to sensitive permissions from an unprotected exposed interface. CHEX [37] proposes a taint-based static analysis approach to vet permission leak vulnerabilities in apps. By tracking the taints in the data-dependency graph statically, it can discover reachable paths between exposed interfaces and sensitive sources of apps.

IccTA [17] is a sophisticated taint-based static analysis approach for detecting permission leak among apps. It enables data and control flow analyses for apps with the aid of existing tools including Epic [38], IC3 [39] and FlowDroid [40]. Further, it merges a bundle of apps so that tainted context can propagate among them, ensuring better precision and accuracy holistically. Similarly, COVERT [11,12] proposes a holistic and comprehensive inter-app vulnerability detecting scheme. It extracts analyzable formal security specifications from each disassembled app within a system, and then analyzes them in their entirety with the assistance of a satisfiability (SAT) solver to globally detect the permission leak vulnerabilities of interactions among apps in a certain system. MR-Droid [41] is a recent static analysis framework based on a MapReduce implementation for detecting inter-app communication threats. It first uses MapReduce to identify all ICC information such as nodes and edges, and then discovers dangerous ICCs according to the risk assessment.

However, since Android has provided several dynamic features such as dynamical-code-supported techniques (dynamical code loading [42] and reflection-based remote code loading [43]) and recent dynamic permission mechanism, it is nearly impossible for these static approaches to handle all the permission leak challenges gracefully in the runtime environment. As a result, additional researches have been conducted on runtime protection techniques with the concerns of permissions and inter-app communications.

QUIRE [9] is an enlightening dynamic solution for app permission leaks in the early days. Through the modification of IPC mechanism, it enables specifically revised apps to append IPC provenances in call-chain form to their IPCs. By analyzing the provenances carried by IPCs, revised recipient apps are able to decide whether to reduce permissions when acting as deputies. Scippa [35] demonstrates a similar idea with a more sophisticated implementation. It extends the Android IPC in both the framework and the kernel to establish provenance, and further provides an additional API to the apps to identify caller apps. However, these approaches are not practical since additional developments are required for millions of existing apps.

Unlike the app-dependent QUlRE, IPC Inspection [8] is a system-focused approach for resisting IPC-based confused deputy attacks. For ICCs among apps, it creates a new instance of the recipient app with reduced sender permissions for the current interaction to avoid permission leaks. Similarly, another multi-instance-based security middleware [21] has been proposed to assist existing Android permission management tools in defending against app permission leaks via ICCs. It also checks ICCs at runtime, but creates permission-reduced app instances according to the blocked permission lists generated from the permission management setting. Obviously, these approaches, based on multiple instances, incur significant overheads that cannot be borne by hosting devices, and are unadaptable and inflexible in maintaining app usability. Given dynamic permission environments, the situation can becomes even worse as the cost grows linearly with the increase of app permission states. A little different from permission-reduced multi-instance approaches, RoppDroid [44] provides a resource virtualization framework to mitigate permission leak threats caused by ICCs without damaging the app usability. By maintaining and analyzing ICC provenance chains, it dynamically virtualizes related resources to ensure that malicious apps can access only virtualized ones to mitigate privilege escalation problems at runtime. Nevertheless, although this solution is aware of dynamic features, it provides no exceptional handling mechanism, leading to unnecessary efforts in some cases.

Except for the aforementioned approaches focusing on permission (resource) control, Xmandroid [19] and its extension [20] implement Android security frameworks via ICC control. These two approaches monitor app interactions during the runtime and maintain communication graphs consisting of existing approved interactions. They provide users with coarse-grained optional policies and ultimately enforce policy-based access control to manage apps’ interactions according to the reachability analysis of apps in the graph and the permissions of connectible apps. However, these approaches produce very high false positive rates because of their defective policies, and are inadequate in dynamic environments due to the inherent stabilization of the Chinese Wall access control model [45].

Taint-based analysis techniques have also been exploited in many dynamic approaches [18,46,47] to address permission leaks. Among these, TaintDroid [18] is the most representative one which analyzes and reports sensitive data leaks of third-party apps during the runtime. It taints the sensitive data generated by privileged operations, traces the corresponding data flow propagating in the system and warns the user if tainted data are detected crossing the sink. However, these taint-based approaches induce non-negligible performance costs that may not be acceptable to real-world devices and taint explosions that leads to many false positives.

In conclusion, most existing dynamic approaches suffer from substantial runtime overhead, especially in dynamic scenarios. They are also unresponsive to dynamic features due to their inflexible and unadjustable security mechanisms, which inevitably compromise their usability.
Table A.1
The capability map.

<table>
<thead>
<tr>
<th>Type</th>
<th>Permission name</th>
<th>Member in CP</th>
<th>Bit position</th>
<th>Protection level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dangerous</td>
<td>READ_PHONE_STATE</td>
<td>PHONE</td>
<td>0</td>
<td>1 &amp; 2 &amp; 3</td>
</tr>
<tr>
<td></td>
<td>CALL_PHONE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE_CALL_LOG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD_VOICEMAIL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>USE_SIP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PROCESS_OUTGOING_CALLS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dangerous</td>
<td>SEND_SMS</td>
<td>SMS</td>
<td>1</td>
<td>1 &amp; 2 &amp; 3</td>
</tr>
<tr>
<td></td>
<td>RECEIVE_SMS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ_SMS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RECEIVE_WAP_PUSH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RECEIVE_MMS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dangerous</td>
<td>READ_CONTACTS</td>
<td>CONTACTS</td>
<td>2</td>
<td>1 &amp; 2 &amp; 3</td>
</tr>
<tr>
<td></td>
<td>WRITE_CONTACTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GET_ACCOUNTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dangerous</td>
<td>READ_EXTERNAL_STORAGE</td>
<td>STORAGE</td>
<td>3</td>
<td>1 &amp; 2 &amp; 3</td>
</tr>
<tr>
<td></td>
<td>WRITE_EXTERNAL_STORAGE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dangerous</td>
<td>READ CALENDAR</td>
<td>CALENDAR</td>
<td>4</td>
<td>1 &amp; 2 &amp; 3</td>
</tr>
<tr>
<td></td>
<td>WRITE CALENDAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dangerous</td>
<td>ACCESS_FINE_LOCATION</td>
<td>LOCATION</td>
<td>5</td>
<td>2 &amp; 3</td>
</tr>
<tr>
<td></td>
<td>ACCESS_COARSE_LOCATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dangerous</td>
<td>CAMERA</td>
<td>CAMERA</td>
<td>6</td>
<td>2 &amp; 3</td>
</tr>
<tr>
<td>Dangerous</td>
<td>RECORD_AUDIO</td>
<td>MICROPHONE</td>
<td>8</td>
<td>2 &amp; 3</td>
</tr>
<tr>
<td>Dangerous</td>
<td>BODY_SENSORS</td>
<td>SENSORS</td>
<td>7</td>
<td>2 &amp; 3</td>
</tr>
<tr>
<td>Normal</td>
<td>INTERNET</td>
<td>INTERNET</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Normal</td>
<td>CHANGE_WIFI_STATE</td>
<td>WIFISTATE</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Normal</td>
<td>VIBRATE</td>
<td>VIBRATE</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Normal</td>
<td>WAKE_LOCK</td>
<td>LOCK</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Normal</td>
<td>DISABLE_KEYGUARD</td>
<td>KEYGUARD</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>Normal</td>
<td>SET_WALLPAPER</td>
<td>WALLPAPER</td>
<td>14</td>
<td>3</td>
</tr>
</tbody>
</table>

Note that we abbreviate permission names by omitting the prefix “android.permission.”

8. Conclusion

In this work, we designed and implemented a configurable and adaptive framework to protect against permission leaks in apps running in Android’s dynamic permission environment. Our framework maintains runtime app states and provides users with flexible runtime policy configuration. In complying with the risk mitigation mechanism, it mediates ICCs among third-party apps according to policies and dynamic capabilities, which provides appropriate and flexible protection for third-party apps, and mitigates the risk induced by user-defined policies. Additionally, we proposed a sophisticated decision cache system with a proactive update method that ensures the efficiency and dependability of decision services. A theoretical analysis and experimental evaluation show that our solution provides configurable and effective protections for third-party apps against permission leak vulnerabilities, with an extremely low impact on system performance and app usability.

Future improvements should defend against not only confused deputy attacks but also collusion attacks in dynamic scenarios. Besides, big data and artificial intelligence techniques (e.g., machine learning) may also be exploited to discover benign and malicious app interaction patterns by mining inter-app access behaviors and corresponding permissions states, helping to accurately identify threatening app interaction requests, and thereby achieving better access control precision. Moreover, except for Intent-based ICCs, more communication methods must be covered to provide comprehensive protection for vulnerable third-party apps.

Acknowledgments

This work was supported by the National Natural Science Foundation of China [grant numbers 61632009, 61702561, 61702562, 61472451]; the China Scholarship Council Foundation [grant number 201506370106]; the Hunan Provincial Innovation Foundation for Postgraduate, China [grant number CX2015B047]; the Guangdong Provincial Natural Science Foundation, China [grant number 2017A030308006]; and the Joint Research Project between Tencent and Central South University.

Appendix

See Table A.1.

References


**Ju Ren** received his B.Sc., M.Sc., Ph.D. degrees all in Computer Science, from Central South University, China, in 2009, 2012 and 2016, respectively. From 2013 to 2015, he was a visiting Ph.D. student in the Department of Electrical and Computer Engineering, University of Waterloo, Canada. Currently, he is a distinguished professor with the School of Information Science and Engineering, Central South University, China. His research interests include sensor networks & IOT, wireless communication & mobile computing, transparent computing and big data.

**Yaoxue Zhang** received the B.S. degree from Northwest Institute of Telecommunication Engineering, China, and the Ph.D. degree in computer networking from Tohoku University, Japan, respectively. He is a Professor with the Department of Computer Science, Central South University, China, and a Professor with the Department of Computer Science and Technology, Tsinghua University, China. He has authored over 200 technical papers in international journals and conferences, as well as 9 monographs and textbooks. His research interests include computer networking, operating systems, pervasive computing, transparent computing, and big data. He is a Fellow of the Chinese Academy of Engineering.