Securing Android Inter-Process Communication (IPC) Using NGAC

Jason Simental, Elmaddin Azizli, Mahmoud Abdelgawad, and Indrakshi Ray
Department of Computer Science, Colorado State University
Fort Collins, CO, 80523, USA
{name.surname}@colostate.edu

***Abstract*—Android apps communicate with each other through a mechanism called Inter-Process Communication (IPC) that allows them to exchange messages known as intents. IPC uses Mandatory Access Control (MAC), referred to as an Intent Firewall, to protect and manage intents between apps. However, IPC poses a significant risk, as malicious apps can exploit IPC to attack other apps. This vulnerability arises from the security architecture and implementation inherent to the Android operating system. To mitigate this vulnerability, we have implemented NIST Next Generation Access Control (NGAC) on top of the Intent Firewall to strengthen the enforcement of IPC access control. The NGAC module enforces stricter IPC security policies by using app attributes, including the installation source, app signature, and app type. We tested the NGAC module on Android 13 across various apps, and we evaluated its performance to ensure that the time required to verify intents between apps remains efficient. The results show that the NGAC module is effective and efficient in securing Android IPC.**

***Index Terms*—Android Security, Inter-Process Communication (IPC), NIST Next-Generation Access Control (NGAC), Secure Apps Communication**

I. Introduction

Android users frequently use apps for communication, entertainment, financial transactions, navigation, and various other essential services in their daily lives. The seamless interaction between Android apps improves user experience and functionality. Android apps often require sharing data and resources through the Inter-Process Communication (IPC) mechanism [1].

IPC is a core component of the Android operating system. It allows apps to request services from one another through exported components, including activities, broadcast receivers, content providers, and services [2]. Android apps request services from each other through messages known as intents. IPC uses a Mandatory Access Control (MAC), referred to as an Intent Firewall [3], to regulate and secure intents between apps. However, IPC poses a significant security risk, as malicious apps can exploit it to attack other applications. This vulnerability stems from weaknesses in the Android security architecture and its IPC implementation, which make it susceptible to intent spoofing, hijacking, and privilege escalation attacks [1], [4], [5], [6], [7], [8].

To improve IPC security, we incorporate the NIST Next Generation Access Control (NGAC [9]) framework with the Intent Firewall. This incorporation strengthens fine-grained access control over IPC and handles Android MAC-based access control limitations. NGAC enforces stricter IPC security policies as it evaluates app attributes, including installation source, app signature, and app type. This evaluation determines whether an intent request should be granted or denied. The NGAC has been standardized by the National Institute of Standards and Technology in alignment with the American National Standards Institute/International Committee for Information Technology Standards (ANSI/INCITS); refer to the standard “INCITS 565-2020” [9]. NGAC is a generic architecture that defines access control as a reusable set of data abstractions and attribute-based functions. It is suitable for expressing access control policies for a wide range of applications, including those that span multiple distributed and interconnected processes.

We implemented the NGAC module on Android 13 and tested its effectiveness across various apps. We also evaluate its performance to ensure minimal verification overhead. The results demonstrate that the NGAC module effectively mitigates IPC vulnerabilities while maintaining efficiency, thereby providing a robust security solution for Android IPC.

The rest of the paper is organized as follows. Section II covers the details of the IPC mechanism and its security vulnerabilities. Section III explains how NGAC is implemented and how the NGAC package is integrated with the Intent Firewall. Section IV evaluates the effectiveness of the NGAC package along with its performance. Section V discusses the limitations of the NGAC package. Section VI explores related research on Android IPC security. Finally, Section VII summarizes our work and outlines the future plan.

II. Background

Android apps run in a sandbox environment, which ensures that each app operates within its own isolated space [10], [11]. This sandboxing mechanism is enforced through the Linux User ID (UID) model, where a unique UID is assigned to each installed app at the time of installation. If an app (caller app) needs to communicate with another app (target app) or with a component outside its sandbox, the target app must be listed as exported inside the target app’s manifest file. IPC utilizes the Intent Firewall, which uses these manifest files to control and secure intents between apps.

A. Android IPC Architecture

As shown in Figure 1, the caller app initiates an intent at the user level, which is then intercepted by the Intent Handler component at the kernel level, framework level (Step 1). The Intent Handler determines the appropriate component or target app based on the type of intent. However, before delivering the intent, it is sent to the Intent Firewall (Step 2). The Intent Firewall evaluates the intent using rules defined in XML files to decide whether to grant or deny its delivery (Step 3). In detail, when an intent is sent, the Intent Firewall's method, *checkIntent()*, handles the access control logic and returns a boolean value indicating whether the caller application is allowed to access the target app. If access is denied, the Intent Firewall notifies the Intent Handler on the caller app side with a rejection message. If access is granted, the Intent Firewall passes the intent to the Intent Handler on the target app side (Step 3) and also informs the Intent Handler on the caller app side. The Intent Handler on the target app side launches the target app (Step 4), and the target app responds to the caller app (Step 5).

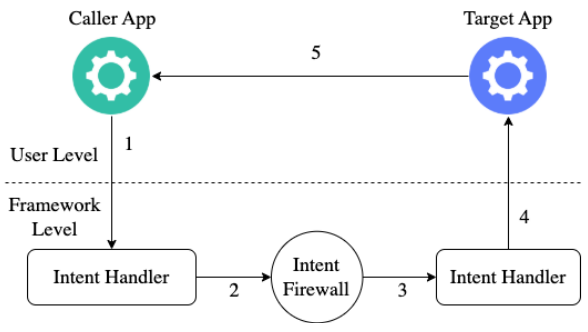

Fig. 1. Inter-Process Communication (IPC) Mechanism

B. IPC-based Attacks

Android IPC is susceptible to various attacks, where malicious apps intercept, modify and inject intents to gain unauthorized access or trigger unintended actions [4], [1], [12], [13], [14].

One common attack is *Activity Hijacking*, where a malicious app sets up an activity with intent filters similar to those of a legitimate app, misleading users into interacting with it under pretenses. Another attack is *Intent Hijacking*, where a malicious app intercepts intents intended for other apps, potentially accessing sensitive user data, and injecting malicious content into a target app process. *Service Hijacking* is an attack where an attacker exploits an exposed bound service without proper permission checks, allowing unauthorized access to sensitive functionality. Additionally, *Broadcast Hijacking* attack occurs when a malicious app sends forged broadcast messages, causing unintended behaviors such as modifying settings, logging out users, and crashing apps. *Pending Intent* exploitation occurs when an app improperly shares a pending intent and an attacker can reuse it with modified parameters to perform actions as the original app.

These attacks emphasize the importance of securing the IPC by implementing appropriate intent filtering and enforcing permissions. We chose to implement NIST NGAC as a fine-grained attribute-based access control mechanism integrated immediately after the Intent Firewall to mitigate unauthorized inter-app communication.

III. IPC NGAC Implementation

The IPC NGAC mitigates intent-based threats by enforcing strict validation of caller and target app attributes. It enforces policy-based access control on the IPC mechanism to prevent unauthorized intent redirection, unauthorized service binding, and malicious broadcast injection.

Policy ID	Policy Description
PC1	if caller app is from a trusted source and signed, it is granted permission to perform any operations on any signed or unsigned target app
PC2	if caller app is from a trusted source and unsigned then it can perform only *startActivity* operation on system resources
PC3	if caller app is sideload and unsigned, then deny all operations on any target app

TABLE I: IPC Enforcement Policies

As shown in Table I, we defined 3 security policies to restrict IPC between apps. PC1 allows all operations from trusted sources and signed apps to be performed on system resources and other target apps. PC2 permits only one operation, specifically (*startActivity*), to be executed on system resources if the caller app is from a trusted source, even if it is unsigned. PC3 prohibits all operations from sideload and unsigned apps from being performed on system resources and other target apps.

The NGAC package enforces these security policies on IPC intents exchanged between apps. NGAC package, as shown in Figure 2, consists of two main modules: an access control module and a database module. These modules store attributes of installed apps and then use these attributes to verify intents between apps against the security policies. The common app attributes are defined as:

- **A_sig** refers to the app signature, which is of the enumerated type `[Unsigned, Self-Signed, Google Signed, Vendor Signed]`.
- **A_is** refers to the app installation source, which is of the enumerated type `[Native, Play Store, Side Load]`.

These app attributes are stored in a small database that is installed on the device. The System Server manages this attribute database during the boot process. When a new app is installed, its attributes are added to the database. App attributes are accessed using the app's UID as a lookup key. A predefined system UID entry is manually inserted into the database to represent all native system apps. The access control module first checks whether the predefined system UID belongs to

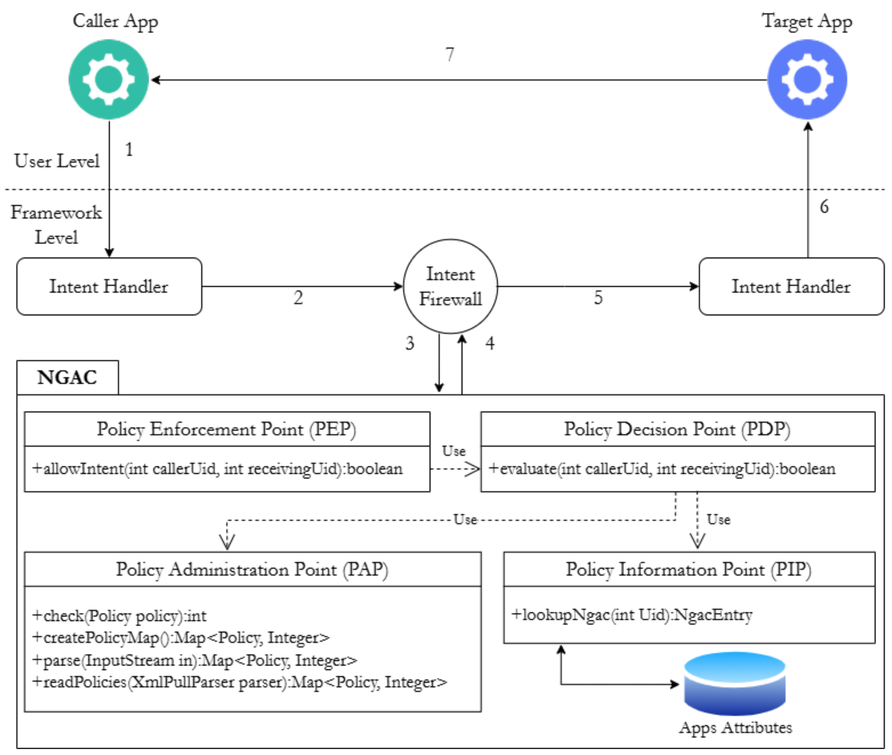
Caller App
Target App
7
User Level
1
Framework Level
6
Intent Handler
2
Intent Firewall
5
Intent Handler
3
4
NGAC
Policy Enforcement Point (PEP)
+allowIntent(int callerUid, int receivingUid):boolean
Use
Policy Decision Point (PDP)
+evaluate(int callerUid, int receivingUid):boolean
Use
Use
Policy Administration Point (PAP)
+check(Policy policy):int
+createPolicyMap():Map<Policy, Integer>
+parse(InputStream in):Map<Policy, Integer>
+readPolicies(XmlPullParser parser):Map<Policy, Integer>
Policy Information Point (PIP)
+lookupNgac(int Uid):NgacEntry
Apps Attributes

Fig. 2. IPC NGAC Enforcement Package

a system component before allowing access. If it does, the module queries the database using the digit "0" as the lookup key. Otherwise, non-system UIDs for non-native apps have unique entries in the database corresponding to their actual UID. This attribute database works in conjunction with the policies defined in the "Policies.xml" file, which is located in the privileged directory ("/data/system/ngac"). The NGAC package is integrated with the Intent Firewall, as we modified the *checkIntent()* function to trigger the NGAC functionality. The NGAC package is activated once the boot process is completed.

A. Access Control Module

The Access Control Module consists of the following components:

- Policy Decision Point (PDP): The PDP is the main component responsible for making the final access decision. It evaluates an intent by comparing the attributes of the caller app and target app against policies and deciding whether to grant or deny the intent.
- Policy Enforcement Point (PEP): The PEP enforces decisions made by the PDP. It ensures that only granted intents can reach their destination.
- Policy Information Point (PIP): The PIP is responsible for querying the database to fetch attributes of the caller and target apps. The PIP provides this information to the PDP in order to make decisions about access intent.
- Policy Administration Point (PAP): The PAP manages access control policies. It allows administrators to add, remove, and modify policies that the PDP uses to evaluate intents.

As shown in Figure 2, the IPC architecture now includes additional steps described as follows.

1) The Caller app initiates the process by sending an intent.
2) One of the three handlers (*Activity, Broadcast, or Service*) intercepts the intent and processes it based on the type of component it intends to access. This handler subsequently sends an access request to the Intent Firewall.
3) Intent Firewall receives the intent and forwards control to the NGAC module by invoking the PEP *allowIntent()* function, including the caller and target UIDs. The PEP then calls the PDP using the PDP *evaluate()* function. The PDP calls the PAP *check()* function, which initializes a map variable named *policyMap* by invoking the *createPolicyMap* function. The *createPolicyMap()* first checks the directory */data/system/ngac* to parse the *Policies.xml* file through the *parse()* function. This function sets up a parser object and calls the *readPolicies()* function to parse and format each policy into a map object. The *readPolicies()* function then recursively returns the map object holding all policies as the *policyMap* variable. The PAP *check()* function uses the *policyMap* variable to find the relevant policy requested by the PDP. The PAP then invokes the PIP *lookupNgac()* function twice: once for the caller app UID and once for the target app UID. The *lookupNgac()* function utilizes the *NGAC Database Service Manager* to query the NGAC database using these UIDs.

4) Once the PDP obtains the attributes associated with both the caller app and the target app UIDs, along with the applicable policy, it makes an access decision. If there is no policy for given attributes, the default decision is to deny access; otherwise, access is granted. The PDP then communicates this decision back to the PEP, which sends it to the Intent Firewall.
5) The Intent Firewall returns the decision to the respective handler, and if the intent is valid, it is sent to the target app.
6) The handler subsequently allows the caller app to access the target app.
7) The target app exposes the requested component to the caller app.

B. Database Module

The attributes of both the caller and target apps are managed through the *NGAC Database Service Manager*. To save space, Figure 2 omits details about the *NGAC Database Service Manager*. However, this section thoroughly explains the *NGAC Database Service Manager* processes and its components.

The database is populated whenever a new app is installed. This process is managed by the NGAC Attribute Parser, which interacts with the native *"InstallPackageHelper"* class. Within the *"InstallPackageHelper"* class, there is a function called *"handlePackagePostInstall()"* that handles operations following the installation of an app [15]. This function takes two arguments: *PackageInstalledInfo* and *InstallArgs*. These arguments contain the essential information needed to parse the app attributes.

We retrieve the app signature and the installation source from the *InstallArgs*. However, the signature alone is not sufficient to identify the signer. To address this, we use a native class (*X509XCertificate*) to create an X509 certificate, which allows us to identify the common name and organization of the signer. This information is used to determine the type of signature attribute assigned to an app by comparing the organization and common name with those of known and trusted organizations. We finally parse the UID of the newly installed app from the *PackageInstalledInfo*, serves as an identifier for indexing into the database.

The NGAC database module provides these functionalities through several components:

- NGAC Database Service: It stores, updates, and deletes the attributes of the app on a device.
- NGAC Database Service Manager: It manages the interaction with the NGAC Database Service.
- NGAC Database Service Helper: It instantiates the NGAC Database Service.
- NGAC Attribute Parser: It parses attributes from a newly installed application and adds them to the database.
- Policy Class: It serves as a policy object that the PAP uses to create policies.
- NGAC Entry Class: It serves as a data structure used to store an app attributes.

IV. IPC NGAC Evaluation

We evaluated the effectiveness of the NGAC package through security testing, in which malicious apps were developed to exploit certain target apps. Furthermore, we evaluated the performance of the NGAC package by measuring the time it takes Android to verify intents with and without NGAC. We then assessed the differences in the time required by NGAC for intent verification.

To perform these evaluation objectives, we set up a testing environment using an Ubuntu Virtual Machine (VM) with 27.9 GB of RAM, 8 processors, and 1 TB of disk space. Within this Ubuntu VM, we ran an Android emulator, which operates as a separate VM. This nested virtualization setup allowed us to simulate a real Android environment while maintaining control over system resources.

A. NGAC Security Testing

For security testing purposes, we created two custom apps: one functions as a caller app named "ngactest", and the other serves as a target app named "ngactesttargetapp". The caller app is designed to send three different types of intents, whereas the target app includes an exported service and a broadcast component. The caller app can send an implicit intent that opens a Web browser at *www.google.com*. It also can send two explicit intents targeting the target app broadcast and service components. We have observed that the Android x86 emulators do not come with the *Google Play Store* installed. As a result, we have explicitly marked certain apps to indicate that they were installed from the Google Play Store source. Lastly, access control decisions made by the Intent Firewall and our NGAC module are logged using the *ADB Logcat* command-line tool [16] for evaluation.

Policy Case	**Caller App Source**	**Caller App Signature**	**Target App Source**	**Target App Signature**	**Decision**
A	preinstall	vendor	preinstall	vendor	Allow
B	play store	google	play store	google	Allow
C	side load	self_signed	preinstall	vendor	Deny
D	side load	self_signed	play store	google	Deny
E	side load	self_signed	side load	google	Deny

TABLE II: IPC Enforcement Policies

As shown in Table II, preinstalled apps that are signed by the vendor can communicate with other preinstalled apps that are also signed by the vendor. Additionally, apps signed by Google on the Google Play Store are allowed to interact with other apps signed by Google from the Play Store, according to the policy file. However, IPC intents sent by side-loaded apps that use self-signed certificates are denied access to both preinstalled Play Store apps and other side-loaded apps signed by Google. These tests demonstrate that the NGAC module effectively blocks untrusted intents from accessing other apps based on app attributes.

B. NGAC Performance

During test execution, we measured the time it took for the Intent Firewall to make an access decision without using NGAC. In a separate round, we also recorded the time taken for an access decision made by the Intent Firewall with NGAC. Table 3 illustrates these two sets of time recordings. Overall, NGAC adds an average of 12.9623 milliseconds of processing overhead with a standard deviation of 2.1985 milliseconds. The difference and variation of execution times with and without NGAC can be explained by several factors. The main factor is that our implementation of NGAC reads and parses the "Policies.xml" file previously mentioned each time an intent is being processed. Other factors involve the current state of memory and the logical paths taken when evaluating various intents with different attributes.

Policy Case	Time With NGAC (Milliseconds)	Time Without NGAC (Milliseconds)
A	15.9686 ms	0.3899 ms
B	15.3521 ms	0.772 ms
C	11.0986 ms	1.5789 ms
D	12.588 ms	1.1481 ms
E	15.2061 ms	1.5129 ms

TABLE III: Performance Table

V. Discussion

Our work separates security decisions between developers and users to prevent vulnerabilities from exported components. In other words, users unknowingly risk exposure when they install insecure apps alongside malicious ones. Our approach eliminates such risk.

We tested the NGAC implementation on Android 13 to enforce IPC security. However, future Android versions may require adaptation due to evolving IPC mechanisms and security policies.

Moreover, NGAC efficiency optimization remains an area for improvement. For instance, the current implementation reads the policy file for each intent, whereas reading it only upon updates could significantly improve performance. The implementation of the NGAC package is hosted in our GitHub repository and is available upon request.

VI. Related Work

Several research efforts [6], [7], [17], [18], [19], [20] have been devoted to Android IPC security. These studies focus on identifying vulnerabilities and analyzing permission misuse. They propose techniques to mitigate unauthorized access and prevent malicious interactions between apps.

Kraunelis *et al.* [17] modified the Binder library to detect and counter deceptive user interface attacks such as malicious activity launched on Android devices. They achieved this by inspecting and analyzing IPC transactions in the operating system. The method involves recording app UIDs and timestamps whenever the *StartActivity* method is called. The recorded timestamp is then compared with the previously recorded timestamp. If the time difference is less than a specified threshold, and the UID of the previous caller is identified as the launcher, while the current caller's UID does not match the package name, the activity is determined to be malicious. In such cases, appropriate actions can be taken to mitigate the threat posed by malware.

In a recent work [7], Lyvas *et al.* explored encrypting transmitted intent data based on user-defined policies. Their approach generates a public/private key pair and a symmetric key via the Android Keystore for authentication and encryption. When an app starts an activity using an implicit intent, the system retrieves the source's keys, signs the intent with the private key, and encrypts it with the symmetric key. This allows users to control the IPC by verifying the signature and decrypting the intent data. However, this system introduces overhead, with the highest measured cost being 190ms, depending on the data size.

Another work by Lyvas *et al.* [20] introduce a tool referred to as *Anactijax*, which is able to identify target apps that may be vulnerable to activity hijacking attacks. After identifying vulnerable target apps, it produces malicious applications that launch activity-hijacking attacks. The authors also introduce a system-level service that authenticates apps that invoke each other called *taskAuth*. This service is an extension of the Activity Manager and, therefore, requires no user or developer intervention. The *taskAuth* tool performs authentication using built-in app signatures.

De Giorgi [18] developed a tool called *Ptracer* to monitor system calls and IPC in Android. *Ptracer* operates between the app and the kernel, intercepting and collecting data such as stack backtraces and system call parameters. It aims to detect debuggers, identify anomalies, and flag privacy concerns by analyzing how frequently an app requests sensitive data through kernel queries, focusing on dangerous behaviors related to privacy.

Khadiranaikar *et al.* [6] propose a hybrid analysis framework that combines static and dynamic analysis to detect IPC vulnerabilities in Android apps. Static analysis was conducted using tools (AndroBugs, QARK, and Jadx) to inspect manifest files and identify insecurely exported components, while dynamic analysis using Drozer confirmed the exploitability of these components at runtime. To mitigate IPC attacks, the authors introduce a local keystore-based intent verification mechanism. In this approach, the caller app generates a cryptographic key, attaches it to the intent, and stores a reference in its content provider. The target app then validates the intent by comparing the attached key with the stored reference to verify authenticity before executing the requested operation. This method strengthens trust between apps and prevents unauthorized intent by utilizing cryptography.

Meng *et al.* [19] propose an approach to improve Android IPC security by integrating static vulnerability detection tools directly into the development environment. Their work emphasizes developer-centered security enforcement. To address this, they introduce custom detectors into Android Studio using the Find Security Bugs (FSB) plugin to identify IPC-related vulnerabilities, such as unprotected broadcast receivers and insecure intent filters, particularly during the software development lifecycle. The approach allows developers to detect IPC flaws at design time rather than post-deployment.

The related works discussed above explore various approaches for detecting and preventing unauthorized IPC in Android apps. These approaches include system-level defenses and monitoring, cryptographic key methods, and the use of automated tools to identify vulnerable apps. In contrast, our approach focuses on providing a flexible and real-time policy enforcement mechanism using the NGAC framework. Inspired by the *IntentAuth* tool described and implemented by Lyvas *et al.* [7], [20], our solution also seeks to eliminate the need for user and developer interaction, which may introduce unnecessary security vulnerabilities. For example, in the *IntentAuth*, user-defined policies are used for secure IPC between apps. However, if a user is unaware of potential security issues, their policy-making decisions may not prioritize device security. In [19], user interaction is removed, and developers are provided with tools to improve IPC security. Nevertheless, a similar challenge occurs if a developer is not concerned about security; they may prioritize app capability over security. These two scenarios motivate us to focus on minimizing user and developer interaction. Although users can still change or add new policies in our implementation, default policies are designed to prioritize security for both users and developers.

VII. Conclusion & Future Work

This work presents the implementation of NGAC, a novel approach to secure Android IPC. We integrate NGAC with the Android Intent Firewall to manage access between apps based on their attributes, such as the installation source, app signature, and app type. The effectiveness and efficiency of the NGAC implementation were thoroughly evaluated. The results indicate that NGAC successfully prevents untrusted access redirection, malicious service binding, and broadcast injection, thus providing a more robust security framework for Android apps. The results also show that NGAC processes access requests between apps efficiently, with an average processing overhead of just 12 milliseconds per IPC request- a minimal amount of time. The NGAC package is hosted in our GitHub repository and is available on request.

Future work will investigate the compatibility of NGAC with future Android versions. It will also investigate broader policy configurations to allow developers and system administrators to define fine-grained access control tailored to specific security needs.

Acknowledgement

This work was supported in part by funding from NIST under Grant Number 60NANB23D152 NSF under Award Numbers DMS 2123761, CNS 2335687, CNS 1822118, NIST, ARL, Statnett, AMI, NewPush, and Cyber Risk Research, and State of Colorado SB 18-086 funding.

References

[1] M. Backes, S. Bugiel, and S. Gerling, "Scippa: System-centric IPC provenance on android," in *Proceedings of the 30th Annual Computer Security Applications Conference(ACSAC)*, 2014, pp. 36–45.

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

[3] S. Mutti, E. Bacis, and S. Paraboschi, "An selinux-based intent manager for android," in *Proceedings of the 2015 IEEE conference on communications and network security (CNS)*. IEEE, 2015, pp. 747–748.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, vol. 17. The Internet Society, 2012, p. 19.

[5] H. Feng and K. G. Shin, "Understanding and defending the binder attack surface in android," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. ACM, 2016, pp. 398–409.

[6] B. Khadiranaikar, P. Zavarsky, and Y. Malik, "Improving android application security for intent based attacks," in *Proceedings of the 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 2017, pp. 62–67.

[7] C. Lyvas, C. Lambrinoudakis, and D. Geneiatakis, "Intentauth: Securing android's intent-based inter-process communication," *International Journal of Information Security*, vol. 21, no. 5, pp. 973–982, 2022.

[8] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein, "Dynamic security analysis on Android: A systematic literature review," *IEEE Access*, 2024.

[9] American National Standards Institute, "Information technology - Next Generation Access Control (NGAC)," ANSI, New York, NY, Tech. Rep., 2020, accessed: 2023.

[10] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into Android applications," in *Proceedings of the 28th annual ACM symposium on applied computing (SAC)*. ACM, 2013, pp. 1808–1815.

[11] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock Android," in *Proceedings of the 24th Security Symposium (USENIX)*, J. Jung and T. Holz, Eds. (USENIX) Association, 2015, pp. 691–706.

[12] X. Zhang, Z. Yu, X. Li, C. Zhang, C. Sun, N. Zhang, and R. H. Deng, "Understanding the bad development practices of Android custom permissions in the wild," *IEEE Transactions on Dependable and Secure Computing*, 2025.

[13] A. Nirumand, B. Zamani, and B. Ladani, "A comprehensive framework for inter-app ICC security analysis of Android apps," *Automated Software Engineering*, vol. 31, no. 2, p. 45, 2024.

[14] S. Akhtar, "A case study: An android security," *Journal of Artificial Intelligence and Computational Technology*, vol. 1, no. 1, 2024.

[15] Android Source Code, "Interact with other apps," march 26, 2025. [Online]. Available: https://cs.android.com/android/platform/superproject/+/android14-qpr3-release:frameworks/base/services/core/java/com/android/server/firewall/IntentFirewall.java;l=58?q=intentFire&ss=android%2Fplatform%2Fsuperproject

[16] Android Developers, "Logcat command-line tool," https://developer.android.com/studio/command-line/logcat, 2024, accessed: 2025-03-21.

[17] J. Kraunelis, X. Fu, W. Yu, and W. Zhao, "A framework for detecting and countering Android UI attacks via inspection of IPC traffic," in *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[18] M. De Giorgi, "System calls monitoring in Android: An approach to detect debuggers, anomalies and privacy issues," *Università Ca'Foscari Venezia*, 2023.

[19] X. Meng, K. Qian, D. Lo, H. Shahriar, M. Talukder, and P. Bhattacharya, "Secure Mobile IPC Software Development with Vulnerability Detectors in Android Studio," in *Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, 2018, pp. 829–830.

[20] C. Lyvas, C. Lambrinoudakis, and D. Geneiatakis, "On Android's activity hijacking prevention," *Computers & Security*, vol. 111, p. 102468, 09 2021.