PendingMutent: An Authorization Framework for Preventing PendingIntent Attacks in Android-based Mobile Cyber-Physical Systems

Pradeep Kumar Duraisamy Soundrapandian
pradeepkumarst@gmail.com
Vellore Institute of Technology
Chennai, Tamilnadu, India

Carlos Rubio-Medrano
carlos.rubiomedrano@tamucc.edu
Texas A&M University - Corpus Christi
Corpus Christi, Texas, USA

Jaejong Baek
jbaek7@asu.edu
Arizona State University
Tempe, Arizona, USA

Geetha S
geetha.s@vit.ac.in
Vellore Institute of Technology
Chennai, Tamilnadu, India

Abstract

PendingIntent (PI) is a key Android feature that allows one app to delegate tasks to another while temporarily granting its permissions. However, this can create security risks if a malicious app gains control. Between 2020 and 2023, 106 CVEs highlighted such vulnerabilities. Our analysis of 180,606 apps found PI-related security flaws in 2.21% (3,993 apps), leading to risks like privilege escalation and unauthorized actions. Beyond mobile apps, PendingIntent (PI) plays a key role in Mobile Cyber-Physical Systems (MCPS) by enabling scheduled workflows and automating tasks. For instance, in IoT and Smart Home Systems, PI manages temperature adjustments based on preset schedules, while in Healthcare and Wearable Devices, it triggers workflows like medication reminders or orthopedic exercises for patients and the elderly. In industrial control systems, PI automates machinery operations. In all these scenarios, PI delegates control to a responsible component or app, which acts on behalf of the delegating app. However, security vulnerabilities could allow attackers to disrupt these operations.

To address this, we propose PendingMutent, a framework that combines dynamic authorization with binary analysis to secure PI. It verifies receiver permissions before execution, reducing attack risks while keeping flexibility. We tested it on 85 benchmark apps (including RAICC and StickyMutent) and five open-source applications. Results show PendingMutent effectively prevents PI-based attacks with minimal performance impact (0.0015%), achieving an F1-score of 88% for intra-app and 98% for inter-app analysis.

CCS Concepts

• **Security and privacy → Software and application security**; **Software security engineering**; **Authorization**; *Formal security models.*

SaT-CPS '25, June 4–6, 2025, PA, USA.

ACM ISBN 979-8-4007-1502-0/2025/6
https://doi.org/10.1145/3716816.3727972

Keywords

Android, PendingIntent, Permissions, Authorization, Privacy and Trust

1 Introduction

Android is a widely adopted operating system worldwide, with an increasing number of applications (apps) available for users. In Q2 2024, Android users could choose from 2.26 million apps, making Google Play the app store with the largest number of apps available [11].

Android-based Mobile Cyber-Physical Systems (MCPS) are equipped with advanced processors, diverse sensors and control APIs, are well-suited for applications in the realm of MCPS, where the physical and digital worlds are interconnected [51, 56]. In industrial settings, these devices often handle sensitive tasks involving critical system security data and proprietary enterprise information. This integration highlights the necessity for robust security measures to protect against potential threats [55]. Consumer protection systems are evolving, with mobile apps providing real-time information on product properties, health risks, manufacturers, and traceability. The IoT-CPS concept seamlessly integrates with Android apps' technological and computing capabilities [49].

This rapid expansion of MCPS system highlights the increasing integration of IoT technologies in industrial sectors, enhancing operational efficiency and connectivity. Despite efforts to maintain a secure app environment, malicious applications continue to emerge. For instance, in November 2024, cybersecurity firm McAfee identified 15 malicious loan apps, known as "SpyLoan" apps, which had been downloaded onto 8 million Android devices [61].

Android by default grants apps with minimal privileges, additional permissions like reading the GPS or making phone calls are granted upon explicit request through AndroidManifest.xml during installation. Malicious apps usually request minimal permissions and then try to collude with privileged apps and at runtime they acquire permissions via PendingIntent (PI). Detecting permission exploitation in colluding Android apps is challenging [65], as they dynamically alter behavior to evade detection. Static analysis often fails due to obfuscation and dynamic code loading [64]. These apps distribute malicious tasks across multiple applications, each requesting minimal permissions, making them appear benign individually.

PendingIntent (PI) is a key attack vector, enabling unauthorized access through inter-app communication. Effective mitigation requires dynamic monitoring of PI exchanges, runtime behaviors, and inter-app interactions to detect and prevent sophisticated threats.

In this paper, we examine the security challenges associated with Android PI misuse, which is essential for app collaborations such as triggering workflows in response to conditions, scheduled events, or external inputs. Components like AlarmManager and JobScheduler play a key role in designing mobile CPS systems. For instance, a health monitoring app may trigger alerts from abnormal sensor readings, while a smart home system automates device control in response to environmental changes.

Android PendingIntent (PI) enables secure task delegation by granting temporary permissions to a receiving app, but it also introduces security risks, including privilege escalation (PE) and unauthorized access [33, 40, 44]. An empty base Intent in a PI is particularly vulnerable, allowing attackers to modify it for external component invocation [33, 43]. Additionally, malicious actors can trigger a PI unexpectedly, disrupting the source app's workflow and leading to unauthorized PendingIntent invocation (UPiI) attacks [40]. StickyMutent [40] uses an ownership-based model [12] to encapsulate PendingIntent (PI) and prevent unauthorized access. However, unwrapping the PI grants the receiver the sender's execution rights, which can lead to privilege escalation and unauthorized actions, highlighting a lack of fine-grained access control.

To address this, we propose PendingMutent, a framework that secures PI interactions through dynamic capability-based authorization, binary analysis, and encapsulation within Ownership-Domain [54]. Ownership-Domain classifies PI operations as privileged or general, restricting sensitive actions (e.g., cancellation) while allowing general actions (e.g., invocation) by external apps. PendingMutent enforces capability-based access control by integrating Ownership-Domain with permissions, ensuring actions like phone calls require `PHONE_CALL`. While permissions support security, binary analysis is key to detecting vulnerabilities like Privilege Escalation in PI [9].

PendingMutent performs binary analysis within Ownership-Domain to mitigate PI vulnerabilities, preventing apps with conflicting permissions or exploit-prone behavior from invoking PI. This paper aims to address the following research questions:

- **RQ1:** How effective is PendingMutent in preventing various PI attacks?
- **RQ2**: How does the effectiveness of PendingMutent compare to other similar tools in the literature?
- **RQ3**: How easy is it for an app to transition from the Android's PI to PendingMutent?

To answer these research questions, we conducted an exploratory analysis on a dataset of 107,580 benign and 9,999 malware apps from a popular Android dataset [25]. Additionally, we crawled 38,246 Google Playstore apps and 22,732 Androzoo apps [5] to cross-validate vulnerabilities. We also examined 654 telecom apps from six major providers and 1,395 OEM apps from three top handset vendors to assess firmware vulnerabilities, sourcing these from APK Combo [13] and APK Mirror [59].

Our exploratory analysis revealed PI-based attacks, including unsafe PI creation and transfer leading to unauthorized PI invocation (UPiI) and Privilege Escalation (PE) attacks. Out of 180,606 apps, 3,993 expose unsafe PI linked to PE attacks, and 20.95% (37,837) have both implicit and broadcast methods to share PIs, making them vulnerable to UPiI attacks.

To evaluate our approach, we implemented PendingMutent as a pluggable Java library, offering developers a secure alternative to the native Android PI library.

In summary, PI-based attacks typically stem from insecure creation practices and the uncontrolled retrieval of PendingIntents by external system components, such as SliceProviders, Notifications, MediaBrowserServices, etc. Additionally, these attacks often exploit privileged methods on the retrieved PI. Replacing Android's native PI with PendingMutent prevents malicious receivers from accessing the encapsulated PI and performing privileged operations.

Testing with 85 benchmark applications showed 100% precision, 78. 3% recall, F1 scores of 0.88 for intra-app analysis, and 100% precision, 95. 7% recall, F1 scores of 0.98 for inter-app analysis, outperforming state-of-the-art methods such as RAICC [33] and StickyMutent [40]. Our contributions are summarized as follows:

(1) **Scenario Apps** - We offer nine real-time apps that demonstrate attack scenarios like Privilege Escalation and Unauthorized PI Invoke (§2.2),
(2) **A Formal Model** - We present a theoretical framework, termed as **PendingMutent**, developed upon the principles of Ownership-Domain, for facilitating secure PI-based communication (§4).
(3) **A Pluggable Library** - To evaluate PendingMutent, we developed a pluggable Java library (~1200 LoC) that can replace the Android PI library (§5).

As part of our commitment to open and reproducible science, we made PendingMutent and our experimental results available [1].

2 Background

We start by providing an introduction to PI and its behavior (§2.1), followed by an illustrative example for the rest of the paper (§2.2), and the classification of PI vulnerabilities (§2.3).

2.1 PendingIntent

Android apps run in a secure sandbox, restricting access to sensitive resources and requiring permissions (e.g., INTERNET) for external access [62].PI acts as an authorization token, allowing a sender app (source) to delegate tasks and temporarily transfer its permissions to a receiver app (delegatee).

Android uses Intents for inter-component communication, allowing apps to request actions from other components. PI is a special type of Intent that grants temporary execution rights to another app while preserving the sender's permissions. In simple terms, while an Intent triggers actions immediately, a PendingIntent lets another app execute the action at a later time with the sender's permissions. The PI's base Intent can be empty, explicit, implicit, or broadcast, each with distinct security implications. Empty base Intents are highly vulnerable to Privilege Escalation, while implicit/broadcast Intents risk Unauthorized Intent Receiver attacks [44, 53]. Explicit Intents are safer. Despite Android's best practices, our analysis

[1] https://anonymous.4open.science/r/tools-1486/

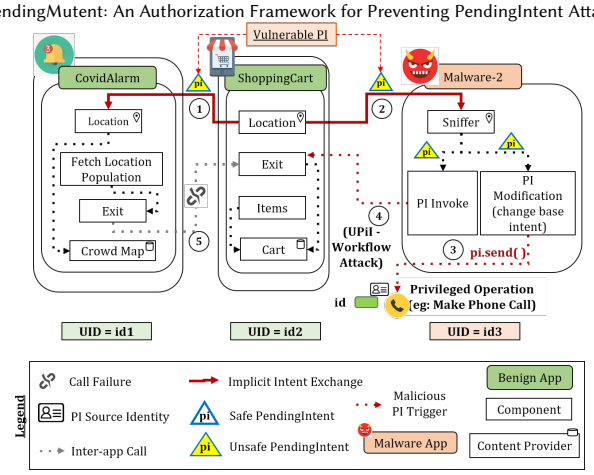


Figure 1: A Privilege Escalation Attack: the ShoppingCart app delegates an unsafe PI via broadcast ①, which is received by Malware-2 ② and modified to perform a phone call (Privilege Escalation) ③. ④ Workflow disruption by malware by invoking Exit activity before the required condition.⑤ The PI invoked by CovidAlarm is affected by Malware-2's Unauthorized PI Invoke (UPiI).

shows 1.7% of Google Play Store apps use vulnerable PI transfers, indicating poor coding practices in real-world apps.

PI includes seven flags that can be combined to define its operational characteristics. For example, `FLAG_ONE_SHOT` ensures the receiver can use the PI only once, reducing the risk of replay attacks [7]. Conversely, `FLAG_IMMUTABLE` prevents the receiver from modifying an existing PI, while `FLAG_MUTABLE` allows inline replies or bubbles, enabling user interaction with a PI via notifications.

Starting from build version `UPSIDE_DOWN_CAKE`, Android allows the use of `FLAG_ALLOW_UNSAFE_IMPLICIT_INTENT`, which bypasses checks for creating a PI with `FLAG_MUTABLE` and an implicit Intent [2]. However, misuse of this flag could lead to Unauthorized PI Invoke and Privilege Escalation attacks.

2.2 Illustrative Example

To illustrate vulnerabilities related to PI in a mobile-powered Cyber-Physical System (MCPS), consider a shopping cart app designed for a consumer-centric supply chain model that fulfills household orders [66]. This app enhances location-based safety by interacting with other apps, such as a COVID tracker, when the user enters a geofence area. The app can send an explicit PendingIntent to apps like Exposure Notifications [47] or use an implicit/broadcast PendingIntent for crowd-sourced apps like Outbreaks Near Me [60]. While explicit communication is secure, it limits app collaboration [29].

Both implicit and broadcast methods of sharing PendingIntents are vulnerable due to the lack of dynamic validation, which leaves them susceptible to unauthorized access and potential attacks like UPiI and PE. In the context of mobile-powered CPS, this implicit/broadcast communication, if not properly secured, could allow attackers to manipulate sensitive location data or trigger unauthorized actions, posing significant risks to both personal safety and CPS security.

The device contains two benign apps: ShoppingCart and CovidAlarm (as shown in Fig. 1), which are granted with permissions such as `ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`, `CALL_PHONE`, and `INTERNET`. The scenario unfolds as follows:

- When the user enters a geofence-tagged shopping mall that tracks COVID-19 population density, the ShoppingCart app allows them to scan item barcodes and add them to their cart.
- Upon entering the geofence, using implicit/broadcast method the ShoppingCart app delegates its geofence information to the CovidAlarm app by creating a PendingIntent, which maps to callback components pointing to the exit activity and includes the app's current geolocation.
- The CovidAlarm app tracks population density within specific geographic areas using third-party geofencing services.
- When the area becomes unsafe (e.g., crowd density exceeds the threshold), the CovidAlarm app triggers the ShoppingCart's PI, which invokes the app's "Exit Activity" to finalize the purchase and proceed to payment.

2.3 Known PI Vulnerabilities and PI Attacks

We categorize PI vulnerabilities into three types: (1) creation vulnerabilities, (2) the method of exchanging PI between components, and (3) behavioral logic between collaborating apps.

Vulnerabilities by Creation - When a PI is created with an empty base Intent (unsafe PI), it allows attackers to perform Privilege Escalation (PE) by modifying the base Intent [33, 44].

In Fig.1, the ShoppingCart app, granted with `CALL_PHONE` permission, creates a PI with an empty base Intent and delegates it via an implicit Intent, making it vulnerable to attacks. A malicious app (Malware-2) that lacks phone call privileges intercepts the PI and modifies the base Intent to initiate a call to its chosen number (Fig.1③). This kind of attack is an example of privilege escalation, and any app containing code that exploits PI can be classified as malware. Notable vulnerabilities include CVE-2023-20950 [21], which involved AlarmManagerActivity.java, and CVE-2023-20962 [22], where unsafe PI creation in MediaVolumePreferenceController.java led to a Privilege Escalation attack, such as the one reported in Samsung Reminder [19].

Vulnerabilities by Transfer - Signature permissions restrict access to a PI, allowing only apps signed with the same key as the sender to use it. However, if a benign app with the same key as the sender re-wraps the PI and broadcasts it, it can inadvertently delegate the protected PI with other apps, including malicious ones. This opens a potential attack vector, allowing unauthorized access to the PI [40]. Re-wrapping a base Intent into a new PI and redistributing it can lead to Privilege Escalation attacks, as seen in CVE-2023-35676 [23]. This vulnerability can result in Unauthorized PI Invokes and Privilege Escalation attacks.

Vulnerabilities by Design - This type of vulnerability arises when an app is poorly designed and fail to consider the side effects of a PI invocation by a receiver app, leading to an Unauthorized PI Invoke attack. In this study, we assume that a well-behaved app follows the correct PI workflow by avoiding base Intent modifications or redistributing the PI through implicit Intents. The PI allows for deferred activation of the base Intent following a specific workflow. Unauthorized triggers disrupt this workflow, as seen in the ShoppingCart example, where an unauthorized trigger (Fig.1 ④) that notifies of crowd bursts in a geofence prematurely interferes with the ShoppingCart app's intended call (Fig.1 ⑤), potentially harming sales by forcing the app to exit the geofence.

Dataset	#Apps	#C	#I	#PI	1s	Nc	Cc	Uc	Im	#PI(C)	#PI(T)	#a(PI(T))	%PI(T)	%Vulnerable Apps	Target SDK
AutoPsy(Benign)	107580	31866422	3736384	119626	3952	1659	12191	23769	393	20136	6325	3151	31.41	2.93	< = 27
AutoPsy(Mal)	9999	3884854	458406	14413	447	180	1337	2857	24	2465	798	68	32.37	0.68	< = 27
Androzoo	22732	49298913	995004	57829	5298	1968	6271	22697	199	7808	217	115	2.78	0.51	< = 32
Google Playstore	38246	80268387	2071370	108550	8108	2099	9822	35876	3282	9735	1249	652	12.83	1.70	< = 32
OEM	1395	3362400	100903	8874	531	247	966	4562	64	437	1	1	0.23	0.07	< = 31
Telecom	654	3319228	67289	6125	557	283	661	2574	30	473	13	6	2.75	0.92	< = 30
Total	180606	172000204	7429356	315417	18893	6436	31248	92335	3992	41054	8603	3993	20.95	2.21	-

NOTE:
$\#C$ - No.of.Classes $\#I$ - No.of.Intent $\#PI$ - No.of.PendingIntent $1s$ - FLAG_ONE_SHOT Nc - FLAG_NO_CREATE Uc - FLAG_UPDATE_CURRENT
Im - FLAG_IMMUTABLE $\#a(PI(T))$ - No.of.Apps with Unsafe PI Transfer $PI(C)$ - Unsafe PI Creation $PI(T)$ - Unsafe PI Transfer
$\%PI(T)$ - Percentage of Unsafe PI Transfer $\%PI(T) = (\#PI(T)/\#PI(C))^*100$ $\%VulnerableApps = 100 * \sum (\#a(PI(T)))/\#Apps$ Cc - FLAG_CANCEL_CURRENT

Table 1: Empirical Study with AndroPsy, Androzoo, OEM, Telecom, and Google Playstore Datasets.

The severity of the attack depends on the PI's behavior. For example, an attack on a PI with the FLAG_ONE_SHOT could prevent the actual receiver from using it again, causing non-deterministic behavior. While FLAG_IMMUTABLE can indicate that a PI should not be altered after creation, attackers can still trigger it to disrupt the sender app's predefined workflow.

Thus, unsafe use and implicit transfer of PI's can result in both Privilege Escalation and Unauthorized PI Invoke attacks. Below, we provide an exploratory study to identify such vulnerabilities in real-world applications.

3 An Exploratory Empirical Study

We conducted an empirical study on PI attacks, including Privilege Escalation and Unauthorized PI Invocation (UPiI), using two data sources: (1) reported Common Vulnerabilities and Exposures (CVEs) and (2) app datasets, including real-world apps from the Google Play Store (Table 1). The first PI attack in 2014 [15] exploited a Privilege Escalation flaw in Android versions before 5.0, granting SYSTEM UID access. By 2024, 107 PI-related attacks had been reported, comprising 19 UPiI and 88 Privilege Escalation cases.

Methodology. Following the approach used in StickyMutent [40], we collected real-world evidence of PI-based attacks from various Android app datasets. Our study involved the following datasets:

- Androzoo [5, 34]: A collection of 24 million APK files, from which we queried 22,732 APKs dated between January 2020 and October 2022.
- Andro-AutoPsy [25, 67]: This dataset consists of 107,580 benign and 9,999 malware apps.
- OEM and Telecom apps: A total of 2,049 preloaded device apps were collected from third-party app stores such as ApkMirror [59] and APK Combo [13].
- Google Play Store: We collected 38,246 random apps from categories such as Maps & Navigation, Strategy, Weather, and Productivity.

In total, our study analyzed 180,606 APKs. For this empirical study, we utilized the Murax APK binary analysis tool [40] to extract key information, such as: (1) Identification of implicit PI creation, (2) Flags used in PI creation, and (3) Transfer or leakage of vulnerable PI's through implicit Intents. Our system configuration included a Lenovo Windows x64 PC with an Intel i5-7200U CPU @2.50GHz, 2701 MHz, 2 Cores. Processing a 28 MB APK file with 25,570 classes took approximately 57 seconds to generate CSV output.

Overall Results. The results of our empirical study, summarized in Table 1, are as follows:

(1) Out of 315,417 PI's analyzed, 41,054 were created with an empty base Intent, demonstrating Vulnerability by Creation, which can lead to privilege escalation attacks.
(2) 8,603 PI's were transferred through implicit Intents, illustrating Vulnerability by Design, which may result in Unauthorized PI Invocation (UPiI) or PE attacks.
(3) 20.95% of PI's pose a potential risk for PE attacks.
(4) 1.7% of Google Play Store apps (652 apps) exhibited vulnerable PI transfers.
(5) Overall, 3,993 apps (2%) demonstrated Vulnerability by Transfer, leading to possible PE attacks.
(6) Out of 654 telecom apps, 6 exhibited unsafe PI transfers, while only 1 out of 1,395 OEM apps showed such vulnerabilities.

Our analysis confirms the empirical tool's robustness against Proguard/R8-obfuscated apps [63], a technique used by about 25% of apps on Google Play [45]. However, it has limitations in detecting dynamic code loading and analyzing web-based content [16, 24]. Our study underscores real-world PI vulnerabilities enabling attacks like PE and UPiI.

4 Our Approach: PendingMutent

We introduce PendingMutent, a dynamic capability-based authorization framework leveraging the Ownership-Domain [54]. It employs binary analysis to encapsulate PI objects, restricting direct receiver access. By enforcing high-level policies, PendingMutent regulates ICC object operations across security domains.

4.1 Theoretical Model

Permissions. The Android permissions set (Ω) categorizes app access levels, while Android services (Υ) define actions via PI, like calls or SMS [52]. Custom permissions [3] are excluded during evaluation, but the framework's set-theory model can accommodate them without modification, as they are a subset of standard permissions. PendingMutent can handle custom permissions without modifying its core mechanism. It uses a set-theory model to manage app permissions and their relationships. Since custom permissions are a subset of Android's standard permissions, PendingMutent can be easily expanded to support them. Custom permissions allow precise control over specific components within an app, such as granting App B access to App A's specific component A_x only if it

$$
\begin{aligned}
&\Omega = \{\ \texttt{normal, signature, dangerous, systems}\}\\
&\Upsilon = \{\ \texttt{Maps, Calls, SMS, ...}\}\\
&\Gamma(A) = \{\exists \text{p};\ \texttt{where}\ p \in \Omega\}\\
&\gamma(A) = \{\ \exists \text{g};\ \texttt{where g} \subseteq \Gamma(A)\ \}\\
&\gamma(c_i, A) = \gamma(A)\\
&\gamma(v \in \Upsilon) = \omega \subseteq \Omega\\
&\tau^A = \text{new Intent(X)}\\
&\text{X} = \varnothing \mid \text{S}\\
&P^A = \text{new PendingMutent(A, } r_c, \tau^A, F)\\
&\text{creator}(P^A) = \text{A}\\
&\pi^A = getPI(P^A)\\
&c_i = \{AC_i, BR_j, SR_j\}\ \texttt{where}\ i \in 1..n,\ \ j \in 0..n\\
&C^A = \{c_i^A; \texttt{where}\ \ i \in 0..n\}\\
&K^A = \{\tau_i^A \mid \pi^A; \texttt{where}\ \ i \in 0..n\}
\end{aligned}
\tag{1}
$$

Figure 2: Grammar for PendingMutent

has the required custom permission. PendingMutent extracts and validates all declared permissions, including custom ones, during its permission validation process.

When an app (e.g., A) is installed, it requests permissions $\Gamma(A)$ in Manifest.xml. The granted permissions $\gamma(A)$ form a subset of these, i.e. $\gamma(A) \subseteq \Gamma(A)$. The method $\gamma(c_i, A)$ retrieves permissions for a component, the method $\gamma(v \in \Upsilon)$ retrieves the permissions for a service request, where each permission ω belongs to the permissions set Ω.

Ownership-Domain. The Ownership-Domain is rooted in the World context and mapped to the device, where each app establishes its own domain consisting of a component domain (C^A) and a capability domain (K^A). The component domain C^A includes elements such as Activities (AC_i), BroadcastReceivers (BR_i), and Services (SR_i) as defined in the app's Manifest.xml. Using the android:exported attribute, an app can control which of its components are visible to other apps. The Capability Domain (K^A) in PendingMutent provides a structured access control mechanism for PendingIntent, ensuring that only authorized applications can perform specific actions. It categorizes operations based on sensitivity, allowing general actions (e.g., send(..)) while restricting privileged operations (e.g., modifying or canceling PendingIntent). This approach dynamically validates receiver permissions, preventing unauthorized access, privilege escalation, and security breaches. By enforcing execution constraints, the Capability Domain mitigates potential threats in Android inter-component communication, particularly in scenarios involving workflow delegation and event-driven triggers. The formal definitions of the utility functions are provided in Fig 2.

ICC Object Creation. Intent objects (τ^A) are created with the statement *new Intent*(X), where A is the source context and X can be an action string [52], or a target component class, or an empty value. PendingMutent (P^A) is created using *new PendingMutent*(A, r_c, τ^A, F), where A is the app context that creates it, r_c is the request code, τ^A is the base Intent object, and F represents the PI flags (`FLAG_ALLOW_UNSAFE_IMPLICIT_INTENT | FLAG_CANCEL_CURRENT | FLAG_IMMUTABLE | FLAG_MUTABLE | FLAG_NO_CREATE | FLAG_ONE_SHOT | FLAG_UPDATE_CURRENT`) (for details refer Section 2.1). Creating a PendingMutent (P^A) with an empty base Intent ($\tau^A = \varnothing$) results in an unsafe Pending Intent (ref. Section 2.3). The utility function $getPI(P^A)$ retrieves the encapsulated PI object (π^A), as shown in Fig 2. This function is intended for functional software testing [26, 32] and can only be invoked by the application that created P^A.

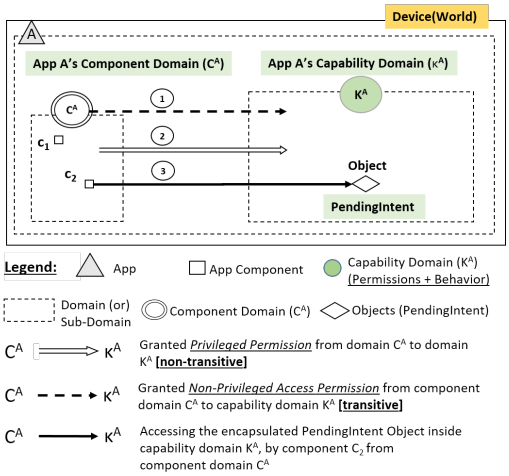


Figure 3: An illustration of capability-based Ownership-Domain with PI creation and access enforcement.

Thwarting PI Operations. An attacker can exploit an unsafe PI to perform a Privilege Escalation attack by altering the base Intent. To prevent this, we propose a subdomain relationship that restricts components in the component domain (C^A) from executing certain operations on encapsulated PI within the capability domain (K^A).

This subdomain relationship is classified into two types based on operational privileges: (1) the non-privileged access permission relationship (dotted arrow, Fig 3 (1)), and (2) the privileged creation permission relationship (double arrow, Fig 3 (2)).

A component domain with creation permission can create PI in the capability domain and update/mutate its encapsulated PI, represented as ($C^A \Rightarrow K^A$). The privileged methods (ρ) that allow CRUD operations (create, read, update, and delete) on the encapsulated Pending Intent are outlined below.

- M_c - Methods to create a new PI object - e.g. getActivity(..), getService(..), getBroadcast(..)
- M_d - Methods to cancel/delete a PI- e.g. cancel()
- M_m - Methods to modify the base Intent of a PI- e.g. send(..Intent intent_obj..), sendIntent(..Intent intent_obj..)

$$\rho = \{M_c, M_d, M_m\} \tag{2}$$

PendingMutent prevents accidental mutation of the encapsulated PI object located within the capability domain by confining the creation permission only to the component domain of the PI creator apps (a.k.a. the parent of PI). This permission is non-transferable between applications, thereby preventing unauthorized applications from performing privileged operations on the encapsulated PI object. (Note: Shared UID is not considered, as Google discourages its use and may remove it in future Android versions [4]).

Conversely, non-privileged access permission (dashed arrow, Fig 3 (1)) is granted to non-privileged deputies (η) that would not modify the internal state of the encapsulated PI object, represented as ($C^A \dashrightarrow K^A$). This non-privileged access permission is transitive and can be transferred between apps during ICC.

- M_s - send(..), which invokes the PI to perform the pre-defined task without modifying the PI's base Intent.

$$\eta = \{M_s\} \tag{3}$$

Therefore, when a component from the component domain attempts to access the encapsulated PI object (solid arrow, shown in Fig 3 (3)), the allowed operations depend on the permissions the component domain holds over the capability domain.

App Behavior. The function $creator(P^A)$ returns the app that creates the PendingMutent. The function $behavior(A)$ (Equation 4) evaluates app A for the existence of PI unsafe PI vulnerable codes.

$$behavior(A) = \begin{cases} MALWARE, & \text{if binary analysis of A has exploit,} \\ BENIGN, & \text{otherwise.} \end{cases} \tag{4}$$

The method **behavior(A)** does a binary analysis of application A for PI vulnerabilities (discussed in Section 2.3). If vulnerable code is detected, the app is classified as malware; otherwise, it is classified as benign.

PI Target Analysis. Android's PI provides lazy execution of the operations associated with the encapsulated base Intent. This lazy callback invokes the target components located within the same app or other apps or Android core services (e.g. Notification Manager Service (NMS), Location Manager Service (LMS), Telephony Manager Service, etc.).

The method piTarget(v^Y, π^A) (Equation 5) takes three parameters, (1) the invoking context (v^Y) - representing app that invokes the PI, (2) the PI (π^A) - specifying the target component invoked by the PI, (3) isMu(π^A) - a boolean specifying whether the PI (π^A) has been modified (T) or not (F).

$$piTarget(v^Y, \pi^A, isMu) = \begin{cases} c, & \text{where } c \in \{C^A \vee \Upsilon\} \\ & \wedge \text{ isMu}(\pi^A) == \text{F} \\ & \wedge \text{ behavior(Y)} == \text{(BENIGN | MALWARE)} \\ d, & \text{where } d \in \{C^A\}, \\ & \wedge \text{ isMu}(\pi^A) == \text{T} \\ & \wedge \text{ behavior(Y)} == \text{(BENIGN | MALWARE)} \\ & \wedge \gamma(d) \in \gamma(Y);\ \wedge \gamma(d) \in \gamma(A) \\ LEAK, & \text{otherwise.} \end{cases} \tag{5}$$

Table 2 specifies the classification of the invoking context into benign or malware based on the binary analysis of the invoking context for unsafe PI vulnerabilities. It considers whether the PI points to components inside or outside the ownership domain and if its base Intent is modified during transit, potentially breaking source application integrity.

The first column, "target," uses 1 to indicate that the invocation targets a component within the creator app's domain (C^A) and 0 for invocations of external Android services or components outside C^A. The second column shows the creator app's permissions for the action, while the third displays the receiver app's permissions (the invoking context). If binary analysis detects unsafe code in the invoking context, the app is classified as malware regardless of permissions. Otherwise, PendingMutent verifies that the invoking context has the required privileges (i.e., $\gamma(v, Y) \subseteq \gamma(A)$) to perform the requested action, assuming the source app holds those privileges.

Figure 4 illustrates three cases where the PI's base Intent is modified during the transit: a benign invocation within the creator's domain; a benign external service invocation when the receiver's permissions match the creator's; and a malware classification when the receiver's permissions are valid for an external service invocation but the creator lacks the corresponding permission.

Accessing Encapsulated PI. Access to the PI within the capability domain (K^A) is controlled by Equation 6. The method $\chi^{(v^Y)}$ invokes the PI (π^A) with the current context v^Y. If the invoking context risks a PI leak,

#	target (1 - Inside; 0 - Outside)	Can PI Source perform the PI Action? (Source Context)	Can PI Receiver perform the PI Action? (Invoking Context)	Is PI modified (by receiver (yes ‖ no) while delegating between apps)	Is Receiver (Benign, or Malware)
1	1	✓	✓	no	Benign
2	1	✓	✗	no	Benign
3	1	✗	✓	no	Benign
4	1	✗	✗	no	Benign
5	0	✓	✓	no	Benign
6	0	✓	✗	no	Benign
7	0	✗	✓	no	Malware
8	0	✗	✗	no	Malware
1	1	✓	✓	yes	Benign
2	1	✓	✗	yes	Benign
3	1	✗	✓	yes	Benign
4	1	✗	✗	yes	Benign
5	0	✓	✓	yes	Benign
6	0	✓	✗	yes	Malware
7	0	✗	✓	yes	Malware
8	0	✗	✗	yes	Malware

Table 2: Classifying the receiver by invocation target, sender, and action execution ability.

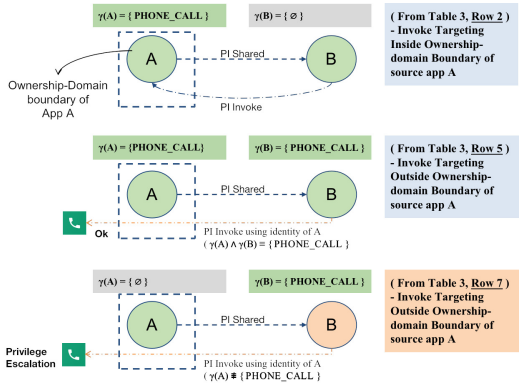


Figure 4: Illustrating the receiver classification, highlighted rows from Table 2.

access is denied; otherwise, it is granted. Equation 7 illustrates the capability domain of app A (K^A), which contains the PI object.

$$\chi^{(v^Y)}(\pi^A) = \begin{cases} DENY, & \text{if piTarget}(v^Y, \pi^A) == \text{LEAK} \\ GRANT, & \text{otherwise.} \end{cases} \tag{6}$$

$$K^A = \chi^A \llbracket\ \pi_i^A\ \rrbracket, \text{ where i} \in \text{1..n} \tag{7}$$

This section explains how PI invocation is restricted based on the behavior of the invoking context and the targeted actions of the PI. It also details the level of access (χ) granted to the invocation context, defining the types of operations permitted in PendingMutent.

The ψ function (Equation 8) validates whether the invoking context exhibits a subsumption relationship ($\preceq$) with the PI creator app. The subsumption relationship emphasizes the hierarchical relationship between applications and their upgrades installed on the device. If such a relationship exists, the invoking context is assumed to have the privileged permission ($\rightarrow$), or else the invoking context is assumed to have non-privileged access permission ($\dashrightarrow$).

$$\psi^{(v^Y)}(\pi^A) = \begin{cases} \Rightarrow, & \text{if } Y \preceq A \wedge \chi^{(v^Y)}(\pi^A) == \text{GRANT} \\ \dashrightarrow, & \text{if } Y \not\preceq A \wedge \chi^{(v^Y)}(\pi^A) == \text{GRANT} \\ \nrightarrow, & \text{otherwise} \end{cases} \tag{8}$$

PendingMutent manages access to the PI by evaluating the receiver's privileges and behavior, including susceptibility to exploit code (Equation 4) and potential leaks (Equation 5). Based on these evaluations, it determines the operations allowed on the encapsulated PI. The PendingMutent's subsumption property ($Y \preceq A$) ensures compatibility with future app updates.

Validating the Invoking Context. The Ξ function (Equation 9) checks whether the invoking context has a subsumption relationship with the PI creator. If so, it grants ρ access; otherwise, it grants η access to other applications or denies it entirely.

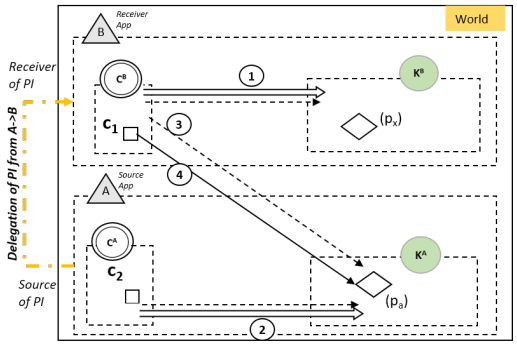


Figure 5: Illustrating the Ownership-Domain capability enforcement during ICC between benign apps, source app A and the receiver app B.

$$\Xi^{(\upsilon^Y)}(\pi^A) = \begin{cases} \rho, & \text{if } \psi^{(\upsilon^Y)}(\pi^A) == \Rightarrow \\ \eta, & \text{if } \psi^{(\upsilon^Y)}(\pi^A) == \dashrightarrow \\ DENY, & \text{otherwise.} \end{cases} \tag{9}$$

$$\Pi^{(\upsilon^Y)}(\pi^A, op) = \begin{cases} GRANT, & \text{if } op \in (\Xi^{(\upsilon^Y)}(\pi^A)) \\ DENY, & \text{otherwise.} \end{cases} \tag{10}$$

The Π function (Equation 10) executes the requested operation on the encapsulated PI if the necessary permissions are granted; otherwise, the operation is denied.

Inter-App Ownership-Domain Relationship. Android ICC allows components of an Android app to exchange information and coordinate functionalities. Proper implementation is crucial for building efficient apps. The Ownership-Domain encapsulates objects and operations, protecting them from malware attacks. By encapsulating PI and their operations from the components that access them, the Ownership-Domain shields the object from malware attacks. In this section, we provide a detailed explanation of the access based on the Ownership-Domain between applications.

Fig 5 illustrates two benign apps, A and B. By default, an app's component domain holds both creation and access permissions to the app's own respective capability domain (Fig 5 (1), (2)). When app A delegates its PI to app B, B gains access to A's capability domain (Fig 5 (3)). However, the component c_1 from C^B can only invoke non-privileged methods (M_s) on the delegated PI (p_a) (Fig 5 (4)). Despite delegating the PI (p_a) to app B, app A retains control over privileged methods. For example, C^A can cancel the active PI (p_a) by calling the privileged method M_d [10]. This demonstrates how PI delegation between apps can provide controlled access to certain actions while maintaining security over sensitive operations.

4.2 System Architecture

PendingMutent consists of five modules (as illustrated in Fig 6): (1) Decision Controller, (2) APK Query Module, (3) App Knowledge, (4) APK Introspect, and (5) PI Access Validator. The details of each module are explained in the following sections.

Decision Controller. The Decision Controller determines if the receiver can access the PI and what actions are allowed. It dynamically validates the receiver's context using Equation 6, while the 'PI Operations Filter' enforces restrictions and the 'Error Display Unit' handles issues. The 'Access Decision Controller' makes the final decision based on the PI creator. It extracts the receiver's context, operation, checksum, and package name (Fig.6 (3)). The APK Query module checks the App Knowledge DB for prior analysis (Fig.6 (4)). If found, the data is sent to the Decision Controller (Fig.6 (9)).

APK Query Module. This module verifies if an app's binary details, permissions, and other attributes exist in the local database using its checksum (Fig. 6 (5)). If app details exist in knowledge DB, they are forwarded to the introspect module for validation. If absent, it retrieves the APK via PackageManager and invokes the APK Introspect module's 'Dynamic Binary Code Inspector' for binary analysis and further validation (Fig. 6 (6)).

Dynamic Binary Code Inspector performs the binary analysis once per app and stored with its checksum in the App Knowledge DB, avoiding redundancy unless the app is updated. It occurs during the initial invocation of PendingMutent. Binary analysis of a 28 MB APK with 25,570 classes may take up to 57 seconds, introducing some overhead, which is being optimized.

App Knowledge. The internal database stores app details like checksum, package name, privileges, and PI exploit codes, maintaining records of app-specific communication.

APK Introspection Module. This module (Fig.6 (6)) retrieves the app's behavior and capabilities from its APK. The dynamic inspector conducts binary analysis to identify any PI exploit code, such as modifying the base Intent or exposing it via implicit broadcast. The findings are then updated in the database (Fig.6 (7)).

PI Access Validator Module. This module ensures that only valid contexts access the PI and permissible operations are granted. It uses Equation 9 to evaluate unsafe code based on the binary information received from Fig.6 (8), the PI invocation boundaries by Equation 5, and the state of the PI by Equation 7. Domain relationships and permissions are determined by Equation 8, and the access validation unit operates based on Equation 9 to decide the permitted operations. Finally, it decides whether requested operations can be executed based on Equation 10 (Fig.6 (9), (10)).

5 Evaluation

PendingMutent is a pluggable Java library for Android (v10) that replaces Android's PI library, comprising 1200 LOC. We tested it on a HONOR Pad X9 (Snapdragon 685, 7GB RAM, 128GB storage, Android 13) and an Android 14 emulator (Pixel 8, API level 34, arm64-v8a)[50]. Our evaluation focused on key research questions.

RQ1 How effective is PendingMutent in preventing various PI attacks?
RQ2 How does the effectiveness of PendingMutent compare to other similar tools in the literature?
RQ3 How easy is it for an app to transition from the Android's PI to PendingMutent?

5.1 Effectiveness of PendingMutent- RQ1

Datasets. Following **RQ1**, we evaluated the effectiveness of PendingMutent on a series of self-developed benign and malware Android apps acting in the *sender* and *receiver* roles of PI. We developed 6 benign apps (3 versions of ShoppingCart and 3 versions CovidAlarm apps), and three malware apps (explained in §2.2). These apps are customized to use PendingMutent instead of the native PI library. The ShoppingCart and CovidAlarm request *dangerous* permissions like `PHONE_CALL, LOCATION, and CAMERA`.

Methodology. The ShoppingCart app uses an *implicit* Intent to exchange PendingMutent in order to dynamically collaborate with third-party apps (say, CovidAlarm). For our tests, an experimental device was installed with the ShoppingCart (3 apps), CovidAlarm (3 apps), and the other 3 malware apps. The malware apps are designed to sniff the PendingMutent and exhibit some of the following malicious properties:

(A) Performing an unintended modification - i.e. the malware can mutate the received unsafe PendingMutent to perform certain malicious activities.
(B) Executing a privilege escalation attack where the receiver lacks the required privilege to invoke PendingMutent, such as making a phone call service without ACTION_CALL permission, following Equation 5.

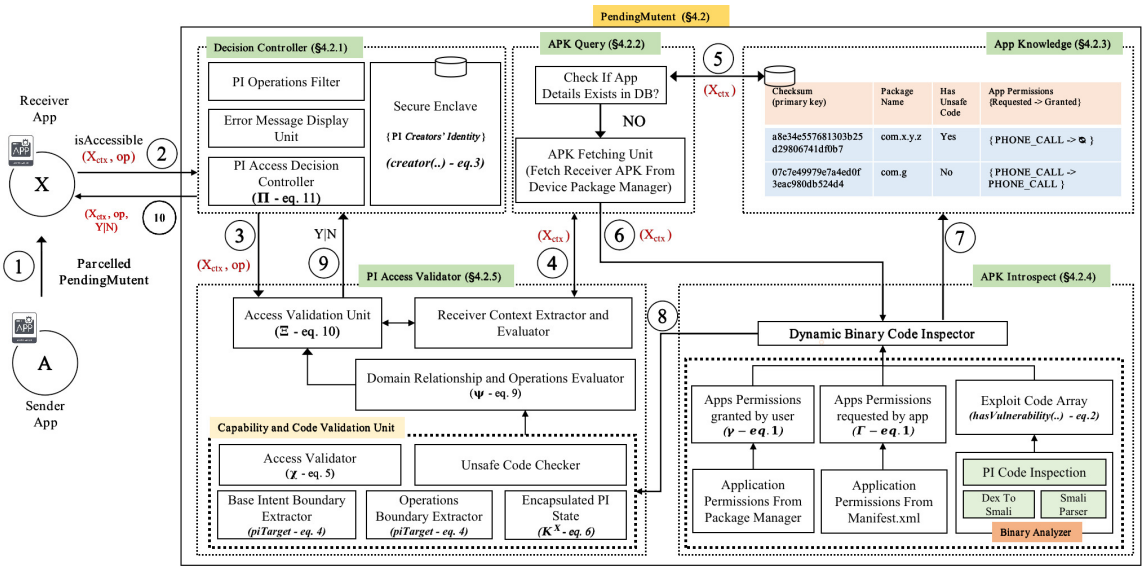


Figure 6: The Architecture of PendingMutent. The PendingMutent verifies if the current receiver context (X_{ctx}) has permission to access the encapsulated PI (②), and dynamically decides whether it can execute the specified operation *op* (⑩).

Ⓒ Performing a workflow attack - i.e. the receiver may invoke the PendingMutent disrupting the pre-defined workflow property (Section 2.3).

PendingMutent detects malware based on app behavior rather than acquired privileges. During testing, a simulated user entered a geo-fenced location, triggering ShoppingCart's Location component to delegate PendingMutent to CovidAlarm via an implicit/broadcast Intent. Malware on the device intercepted PendingMutent by registering on the same implicit/broadcast communication channel.

Results. However, the malware *receiver* could not access the PendingMutent, when it performs the malicious property Ⓐ and Ⓑ, however, PendingMutent cannot block the malicious property Ⓒ from happening. This approach safeguards PI against potential privileged escalation attacks. Moreover, through the utilization of Ownership-Domain, it effectively restricts the recipient's ability to invoke privileged methods on the PI.

5.2 RAICC(IccTA) vs StickyMutent vs Android PI vs PendingMutent- RQ2

Datasets. To address **RQ2**, we leveraged the StickyMutent dataset [40], a well-known collection of 22 apps displaying PI-based attacks, to assess the efficiency of PendingMutent as follows.

Methodology. The StickyMutent dataset consists of 22 intra-communicating apps with 51 components and 51 inter-communicating apps. In total, we have 73 applications all using PI's for ICC. In addition, we added 3 intra-communicating apps and 9 inter-communicating apps demonstrating the scenarios as mentioned in Ⓐ, Ⓑ, Ⓒ (described in Section 5.1).

Results. Table 3 compares PendingMutent with RAICC-instrumented IccTA and StickyMutent. PendingMutent mitigates 57 out of 60 inter-communication vulnerabilities but has limitations with workflow attacks, causing false negatives when a benign receiver inadvertently invokes the received PendingMutent (WorkflowAttack3). Despite this, PendingMutent outperforms StickyMutent and RAICC in defending against PI-based attacks. Evaluated with Precision, Recall, and F1 scores, PendingMutent achieved 100% precision and 78.3% recall in intra-app analysis, and 100% precision with 95.7% recall in inter-app analysis, with F1 scores of 0.88 and 0.98, respectively, supporting **RQ2**. Of 85 tested apps, 5 had false negatives in intra-app communication, and 3 in inter-app communication, where PendingMutent couldn't handle workflow attacks.

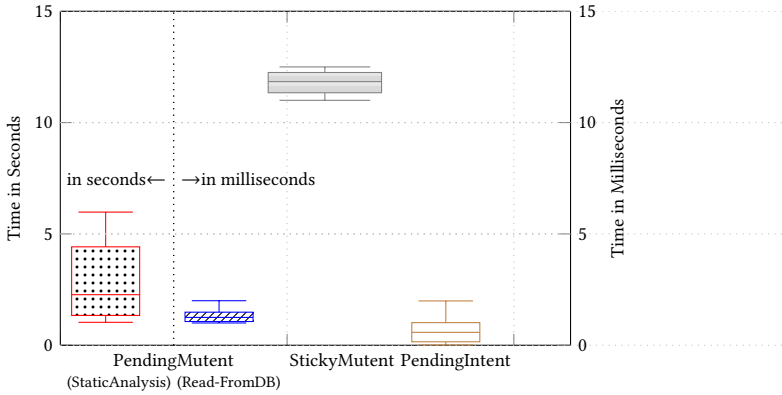


Figure 7: Creation Time from Benchmark Dataset: PendingMutent vs StickyMutent vs PI.

5.3 Efficiency in Adapting PendingMutent- RQ3

Datasets. To answer the **RQ3** question on the real-time adaptability of our approach, we chose five apps that are using PI: four from Github - PushNotification[6], SimpleNotification_1[14], SimpleNotification_2[48], SimpleNotification_3[57] and our self-developed Shopping Cart app (explained in §2.2).

Methodology. These five Android/Java projects were modified to use PendingMutent instead of Android's PI library. These apps were tested with Culebra [58], a playback tool that recorded simulated user interactions for 5 minutes, which were then replayed for 100 runs. The average time overhead to create an PendingMutent object, based on these tests, is shown in Fig.8

Results. The performance overhead of PendingMutent is minimal compared to PI creation, as shown in Fig.7. Users reported slight latency during the initial invocation due to binary analysis, but no noticeable delays afterward. On average, creating a PendingMutent object took 1–2 ms with database info and 1–6 s during static analysis, while a PI object took 0.01–1.99 ms

	RAICC(IccTA)				StickyMutent				**PendingMutent**			
	Intra		Inter		Intra		Inter		Intra		Inter	
	⊕	⊖ | ⊗	⊕	⊖ | ⊗	⊕	⊖ | ⊗	⊕	⊖ | ⊗	⊕	⊖ | ⊗	⊕	⊖ | ⊗
Total Vulnerabilities	16	7	13	10	18	5	20	3	18	5	22	1
Precision p = ⊕/(⊕+⊗)	88.9%		86.7%		100%		100%		**100%**		**100%**	
Recall r = ⊕/(⊕+⊖)	76.2%		61.9%		78.3%		87.0%		**78.3%**		**95.7%**	
F1-Score = $(2pr)/(p+r)$	0.82		0.72		0.88		0.93		**0.88**		**0.98**	
Accuracy = ((⊕+⊙)/(⊕+⊙+⊗+⊖))	70%		57%		78%		87%		**78%**		**96%**	

⊕ - True Positive (TP) ⊙ - True Negative (TN) ⊖ - False Negative (FN) ⊗ - False Positive (FP)

Table 3: RAICC (IccTA) vs StickyMutent vs PendingMutent.

(Fig.7). Despite the static analysis overhead, PendingMutent 's performance impact is minimal (0.0015%).

5.4 Discussion

PendingMutent, a Java library, requires minimal code refactoring and integrates easily by replacing the PendingIntent API (Section 5.3). PI Fixer automates converting APKs to PendingMutent, processing a 46MB APK in 5.7 minutes (Section 5.4). It enhances Android's security by adding a dynamic validation step before a receiver accesses a PendingIntent, using binary analysis to detect vulnerabilities and assess capabilities, unlike Android's static permission checks. PendingMutent ensures only authorized actions are allowed. While the current Java implementation introduces latency, integrating it into AOSP or using cloud-based models like Dypoldroid [27] could reduce overhead. This study does not cover Java reflection or dynamic code loading, focusing on the receiver's actions on PendingIntent (Section 4.1, Eq. 2, 3).

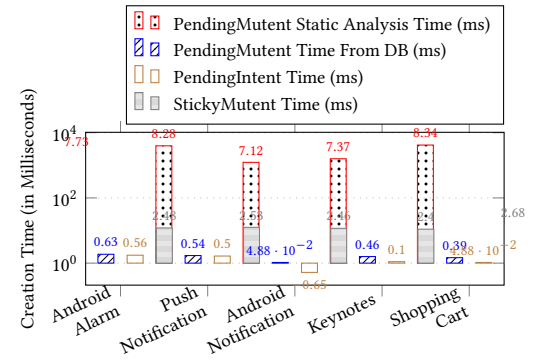
6 Related Work

We categorize the related literature into three areas: (1) PI security, (2) static dex analysis for filtering malicious components, and (3) incremental analysis using knowledge graphs and dynamic policies.

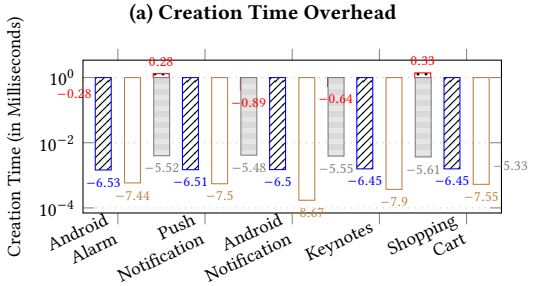
PI Security. Security vulnerabilities related to PI have been studied by RAICC [33] and PIAnalyzer [44]. PIAnalyzer uses static analysis to detect PI vulnerabilities. RAICC adds a method startActivity() to PI calls, converting them to standard ICC calls, improving the precision of tools like IccTA [36] and Amandroid [31]. PendingIntent-Tracker [28] reports vulnerabilities in OEM apps. Unlike these approaches, PendingMutent combines static and dynamic analysis to prevent malware from accessing PI. Vulnerabilities first reported in 2014 [15], with recent ones [17, 18, 20], enable attacks like Privilege Escalation (PE)[43] and Unauthorized Invoke (UI)[29], exploiting weak PI exchanges via implicit Intents [28, 33, 44].

ICC Static Analysis. Static analysis and dynamic monitoring effectively detect ICC attacks [30, 42, 46]. Barros et al.[39] track taint flow with security annotations. Permission re-delegation attacks bypass Android's permission system[9]. SALMA [38] uses a Multiple-Domain-Matrix graph to detect ICC issues, while Amandroid [31] tracks control and data flow. CHEX [37] treats component vulnerabilities as data-flow problems, and AnFlo [8] classifies apps as malware or benign. TERMINATOR [1] grants and revokes permissions based on system safety. Integrating tools like SALMA [38] and Amandroid [31] into PendingMutent 's introspection module (§4.2) improves app interaction predictions by creating event-flow graphs.

Policies & Intent-based Filters. Certifying software via source code inspection is an effective malware mitigation strategy. Aquifer [35] defines UI workflow policies for app interaction, while Mutent [41] protects ICC communication using policies and encryption. Maxoid [68] enables receivers to access sender data exchanged via Intents, while preventing leaks. In contrast, PI necessitates a dynamic authority framework that combines static analysis and dynamic decision-making, which PendingMutent incorporates into its design, to classify receivers before granting access.



(a) Creation Time Overhead



(b) Performance Overhead Percentage

Figure 8: Assessing the Time Overhead between PendingMutent, StickyMutent, and PI, Conducted through a Study on APKs Sized between 40-45 MB.

Methods	Benign Receiver with Capability Superset	Benign Receiver without Capability Superset	Malware Receiver with Capability Superset	Malware Receiver without Capability Superset
Android PI	TN	TN	FN	FN
StickyMutent	TN	FP	FN	TP
PendingMutent	TN	TN	TP	TP

TN - a benign receiver granted to access the encapsulated PI (All Good); ***TP*** - a malware receiver blocked from accessing the encapsulated PI (PI is protected); ***FN*** - a malware receiver granted to access the encapsulated PI (Lost data + identity); ***FP*** - a benign receiver blocked from accessing the encapsulated PI (Lost workflow triggers)

Table 4: Protecting PI from malware receiver: comparison of Android PI Library vs StickyMutent vs PendingMutent

7 Conclusion

This paper introduces PendingMutent, an authorization framework for securing PendingIntent in Android apps within mobile-powered Cyber-Physical Systems (CPS). By integrating capability-based authorization, binary analysis, and an ownership domain model, PendingMutent restricts privileged actions while enabling secure app interactions. Implemented as a pluggable Java library, it provides runtime protection against Privilege Escalation and Unauthorized PendingIntent Invocation attacks, ensuring

the integrity of mobile-driven CPS in domains like smart grids, autonomous vehicles, and industrial automation by enabling secure delegation and component interaction to prevent malicious disruptions.

Acknowledgments

This work was partially supported by the US National Science Foundation (NSF) under Grants No. 2131263 and No. 2232911, and by the US Department of Transportation (USDOT) Tier-1 University Transportation Center (UTC) Transportation Cybersecurity Center for Advanced Research and Education (CYBER-CARE). (Grant No. 69A3552348332).

References

[1] Alireza Sadeghi et.al. 2018. A temporal permission analysis and enforcement framework for android. *In* 40th ICSE, (2018).
[2] Android 14 Beta. 2024. . https://developer.android.com/reference/android/app/PendingIntent#FLAG_ALLOW_UNSAFE_IMPLICIT_INTENT
[3] Android: Define a custom app permission. 2024. . https://developer.android.com/guide/topics/permissions/defining Accessed: 01-Nov-24.
[4] Android Developers Docs Guides, <manifest>. 2024. . https://developer.android.com/guide/topics/manifest/manifest-element#uid Accessed: 01-Nov-24.
[5] AndroZoo. 2024. . https://androzoo.uni.lu/ Accessed: 01-Nov-24.
[6] Lê Văn Anh. [n. d.]. *Alarm.* https://github.com/leanh153/Android-Alarm Accessed: 16-Sep-24.
[7] Replay attack. 2024. . https://en.wikipedia.org/wiki/Replay_attack Accessed: 01-Nov-24.
[8] Biniam Fisseha Demissie et.al. 2018. Anflo: detecting anomalous sensitive information flows in android apps. *In* MOBILESoft. (2018).
[9] Biniam Fisseha Demissie et.al. 2020. Security analysis of permission re-delegation vulnerabilities in android apps. *In* Empirical Software Engineering. (2020).
[10] PendingIntent Cancel. [n. d.]. . https://developer.android.com/reference/android/app/PendingIntent#cancel() Accessed: 25-Mar-24.
[11] Laura Ceci. 2024. *Number of available applications in the Google Play Store from March 2017 to June 2024.* https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ Accessed: 01-Nov-24.
[12] Clarke D.G et.al. 2001. Simple ownership types for object containment. *In* ECOOP'01. (2001).
[13] APK Combo. 2024. . https://apkcombo.com/,
[14] Android custom push notification layouts. 2024. . https://github.com/WebEngage/android-custom-push-layouts Accessed: 01-Nov-24.
[15] CVE-2014-8609. 2024. *Android Settings application privilege leakage.* https://nvd.nist.gov/vuln/detail/CVE-2014-8609/ Accessed: 01-Nov-24.
[16] CVE-2020-4100. 2024. *Android dynamic code loading.* https://nvd.nist.gov/vuln/detail/CVE-2020-4100/ Accessed: 01-Nov-24.
[17] CVE-2021-25352. [n. d.]. *Using PendingIntent with implicit intent.* https://nvd.nist.gov/vuln/detail/CVE-2021-25352 Accessed: 16-Sep-24.
[18] CVE-2021-25364. [n. d.]. *A pendingIntent hijacking vulnerability.* https://nvd.nist.gov/vuln/detail/CVE-2021-25364 Accessed: 16-Sep-24.
[19] CVE-2022-22285. [n. d.]. *Execute privileged action.* https://www.cvedetails.com/cve/CVE-2022-22285/ Accessed: 16-Sep-24.
[20] CVE-2022-22286. [n. d.]. *Execute privileged action.* https://nvd.nist.gov/vuln/detail/CVE-2022-22286 Accessed: 16-Sep-24.
[21] CVE-2023-20950. 2024. *Bypass background activity via a pendingIntent.* https://nvd.nist.gov/vuln/detail/CVE-2023-20950 Accessed: 01-Nov-24.
[22] CVE-2023-20962. 2024. *Start foreground activity via unsafe PendingIntent.* https://nvd.nist.gov/vuln/detail/CVE-2023-20962 Accessed: 01-Nov-24.
[23] CVE-2023-35676. 2024. *Trigger a background activity launch due to an unsafe PendingIntent.* https://nvd.nist.gov/vuln/detail/CVE-2023-35676
[24] CVE-2023-42471. 2024. *Remote attacker executing arbitrary JavaScript code via a crafted intent.* https://nvd.nist.gov/vuln/detail/CVE-2023-42471/ Accessed: 01-Nov-24.
[25] Andro-AutoPsy Dataset. 2024. *Andro-AutoPsy Dataset.* https://ocslab.hksecurity.net/andro-autopsy Accessed: 01-Nov-24.
[26] Ariel Rosenfeld et.al. 2018. Automation of Android applications functional testing using machine learning activities classification. *In MOBILESoft'18.* (2018).
[27] Carlos E. Rubio-Medrano et.al. 2023. DyPolDroid: Protecting Against Permission-Abuse Attacks in Android. *Information Systems Frontiers,* (2023).
[28] Chennan Zhang et.al. 2022. PITracker: Detecting Android PendingIntent Vulnerabilities through Intent Flow Analysis. *In Proc. of WiSec'22.* (2022).
[29] Erika Chin et.al. 2011. Analyzing inter-application communication in android. *In* 9th MobiSys. (2011).
[30] Felt A P et.al. 2011. Permission re-delegation: Attacks and defenses. *In* USENIX Symposium (2011).
[31] Fengguo Wei et.al. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *In* ACM CCS'14. (2014).
[32] Glenford J.Myers et.al. 2015. The Art of Software Testing, 3rd Edition. *Wiley Publishing. ISBN: 978-1-119-20248-6.* (2015).
[33] Jordan Samhi et.al. 2021. Raicc: Revealing atypical inter-component communication in android apps. *In* 43rd ICSE, IEEE/ACM, (2021).
[34] Kevin Allix et.al. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. *In ACM 13th MSR'16.* (2016).
[35] Limin Jia et.al. 2013. Run-time enforcement of information-flow properties on android. *In* ESORICS. (2013).
[36] Li Li et.al. 2015. Iccta: Detecting inter-component privacy leaks in android apps. *In* ICSE'2015, *volume 1.* (2015).
[37] Long Lu et.al. 2018. Chex: statically vetting android apps for component hijacking vulnerabilities. *In* ACM CCS. (2018).
[38] M Hammad et.al. 2018. Self-protection of android systems from inter-component communication attacks. *In* ACM/IEEE ASE'18. (2018).
[39] Paulo Barros et.al. 2015. Static analysis of implicit control flow: Resolving java reflection and android intents. *In* 30th ASE'15. (2015).
[40] Pradeepkumar D.S et.al. 2022. On Shielding Android's Pending Intent from Malware Apps Using a Novel Ownership-Based Authentication. *In JCSC* (2022).
[41] Pradeepkumar D S et.al. 2021. Mutent: Dynamic android intent protection with ownership-based key distribution and security contracts. *In* l HICSS'54. (2021).
[42] Steven Arzt et.al. 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *In* PLDI'14, (2014).
[43] Sven Bugiel et.al. 2012. Towards Taming Privilege-Escalation Attacks on Android. *In* NDSS'12. (2012).
[44] Sascha Groß et.al. 2018. Pianalyzer: A precise approach for pendingintent vulnerability analysis. *In* In ESORICS.. (2018).
[45] Xiaolu Zhang et.al. 2021. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation* (2021).
[46] Youn Kyu Lee et.al. 2017. A sealant for inter-app security holes in android. *In* 39th ICSE, (2017).
[47] Exposure Notifications. 2024. . https://www.google.com/intl/en_us/covid19/exposurenotifications/ Accessed: 01-Nov-24.
[48] Jaison Fernando. [n. d.]. *Notification.* https://github.com/jaisonfdo/NotificationExample Accessed: 02-Nov-24.
[49] Genādijs Moskvins. 2022. On Intelligent Sensors and Internet of Things Based Cyber-Physical System for Consumer Protection. *In International Journal of Agricultural Science* (2022).
[50] Google Play ARM 64 v8a System Image. 2024. . https://developer.android.com/about/versions/14/get Accessed: 24-Mar-24.
[51] et.al. Guo, Yanxiang. 2018. Mobile Cyber Physical Systems: Current Challenges and Future Networking Applications. *IEEE Access* (2018).
[52] Android Common intents. 2024. . https://developer.android.com/guide/components/intents-common/ Accessed: 01-Nov-24.
[53] Hongqi Wu. Jice Wang. 2018. Android Inter-App Communication Threats, Solutions, and Challenges. (2018). https://arxiv.org/abs/1803.05039
[54] Neel Krishnaswami and Jonathan Aldrich. 2005. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. *In* PLDI'05. (2005).
[55] Lingguang et.al Lei. 2013. A Threat to Mobile Cyber-Physical Systems: Sensor-Based Privacy Theft Attacks on Android Smartphones. In *IEEE TrustCom'13.*
[56] Haoyu et.al Ma. 2021. Deep-Learning–Based App Sensitive Behavior Surveillance for Android Powered Cyber–Physical Systems. *IEEE Transactions on Industrial Informatics* (2021).
[57] Akash Manna. 2024. *Keynotes.* https://github.com/akash2099/KeepNotes-AndroidApp Accessed: 01-Nov-24.
[58] Diego Torres Milano. 2024. *Culebra: Ready-to-execute scripts for black box testing.* https://github.com/dtmilano/AndroidViewClient/wiki/culebra
[59] APK Mirror. 2024. (2024). https://www.apkmirror.com/
[60] Outbreaks Near Me. 2024. . https://outbreaksnearme.org/us/en-US Accessed: 01-Nov-24.
[61] Fernando Ruiz. [n. d.]. *SpyLoan: A Global Threat Exploiting Social Engineering.* https://tinyurl.com/5n95un2f Accessed: 20-Feb-25.
[62] Application Sandbox. 2024. . https://source.android.com/docs/security/app-sandbox Accessed: 01-Nov-24.
[63] Shrink, obfuscate, and optimize your app. 2024. . https://developer.android.com/build/shrink-code Accessed: 01-Nov-24.
[64] Latika Singh and Markus Hofmann. 2017. Dynamic behavior analysis of android applications for malware detection. In *2017 ICCT.*
[65] Maddie Stone. 2019. *Securing the system: A deep dive into reversing android pre-installed apps.*
[66] Deloitte Survey. 2024. . https://www2.deloitte.com/us/en/pages/consumer-business/articles/retail-recession.html Accessed: 01-Nov-24.
[67] Jae wook Jang et.al. 2015. Andro-AutoPsy: Anti-malware system based on similarity matching of malware and malware creator-centric information,. *In Journal of Digital Investigation.* (2015).
[68] Yuanzhong Xu and Emmett Witchel. 2015. Maxoid: Transparently confining mobile applications with custom views of state. *In* EuroSys'15, (2015).