An Empirical Study on Fingerprint API Misuse with Lifecycle Analysis in Real-world Android Apps

Xin Zhang*
Fudan University
zhangx22@m.fudan.edu.cn

Xiaohan Zhang*
Fudan University
xh_zhang@fudan.edu.cn

Zhichen Liu
Fudan University
liuzc22@m.fudan.edu.cn

Bo Zhao
Fudan University
bzhao23@m.fudan.edu.cn

Zhemin Yang
Fudan University
yangzhemin@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

Abstract—**Fingerprint-based authentication (FpAuth) is increasingly utilized by Android apps, particularly in highly sensitive scenarios such as account login and payment, as it can provide a convenient method for verifying user identity. However, the correct and secure use of Android fingerprint APIs (FpAPIs) in real-world mobile apps remains a challenge due to their complex and evolving nature.**

This paper presents the first systematic empirical analysis of FpAPI misuses in Android apps from the perspective of the FpAuth lifecycle. First, we develop specialized tools to identify and analyze apps employing FpAPIs, examining their characteristics. Then we define the threat models and categorize four prevalent types of FpAPI misuses through a detailed lifecycle analysis in practical settings. Finally, we develop tools to automatically detect these misuse types in 1,333 apps that use FpAuth and find alarming results: 97.15% of them are vulnerable to at least one type of misuse, with 18.83% susceptible to all identified misuse types. The consequences of such misuses are significant, including unauthorized data access, account compromise, and even financial loss, impacting a broad user base. We have responsibly reported these vulnerabilities, resulting in the issuance of 184 CVE IDs and 19 China National Vulnerability Database (CNVD) IDs, as well as acknowledgment from 15 vendors. We hope this work can raise awareness and emphasize the importance of proper usage of FpAPIs.

I. INTRODUCTION

Nowadays fingerprint-based authentication (FpAuth) has gained significant popularity in real-world mobile applications (apps), as it provides an efficient and convenient way for users to prove their identities. Specifically, FpAuth is widely used in various security-sensitive scenarios, such as login, payment, and authorizing access to sensitive resources. As a result, the security of FpAuth in real-world mobile apps becomes a major concern that demands more attention and protection.

In Android apps, developers are required to implement FpAuth using the provided Android fingerprint APIs (FpAPIs). However, these APIs come in different versions and are often integrated with cryptographic APIs to enhance security, which can be complex and may lead to inadvertent misuse by developers if not managed properly. Given the critical role of FpAuth in securing sensitive operations and the potential severe consequences of its compromise, such as unauthorized access and data breaches, there is a pressing need to investigate how these APIs are misused in the wild. This study, therefore, aims to conduct a thorough empirical investigation into the misuse of FpAPIs in real-world Android apps, providing insights into the vulnerabilities and enhancing the overall security of FpAuth[1].

Existing works on FpAuth security can be mainly classified into two categories: 1) *Spoofing Fingerprint Hardware/Algorithms*. These studies focus on spoofing the process of fingerprint matching using fake or wrong fingerprints [1–6]. For example, DeepMasterPrints [4] uses a neural network to create a synthetic fingerprint image that can fool a fingerprint matcher, while FakeGuard [2] improves the fingerprint forgery attack with more realistic and low-cost fake fingerprints. 2) *Spoofing Users*. The focus of these studies [7–9] is to trick users with hijacked and crafted UI, rather than locating vulnerabilities which can bypass the authentication process. However, the above works do not consider the implementation specifics of FpAuth, where improper FpAPI usage may also pose security threats.

The most relevant prior work to our research is BrokenFingers [10], which deeply analyzes the cryptographic check in FpAuth implementations during the verification stage. In contrast, our research goes beyond this scope by delving into the whole aspects of the FpAuth lifecycle from the user's perspective. Specifically, we uncover previously unexplored areas, including the proper deactivation of FpAuth functionality and the management of updates and changes to fingerprint templates on the device. By examining these overlooked aspects, our study provides a more comprehensive understanding of FpAPI misuses in real-world Android apps.

We begin our research by identifying 1,333 (2.05%) FpAuth apps from a dataset of 65,086 apps collected from various app markets. Most of the FpAuth apps are popular and provide

*Co-first authors.

Network and Distributed System Security (NDSS) Symposium 2025
24–28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230699
www.ndss-symposium.org

[1]For convenience, this paper abbreviates the Android fingerprint API as "*FpAPI*", fingerprint-based authentication using FpAPIs as "*FpAuth*", and Android apps utilizing FpAuth as "*FpAuth apps*".

sensitive services to a large user base. We then conduct a measurement study on these FpAuth apps to understand their characteristics. Based on our findings, we are able to delineate the threat models and summarize four common types of FpAPI misuses. These misuses include: ① using outdated, less secure versions of FpAPIs instead of the newer, more secure ones (*Obsolete API Usage*), ② failing to correctly bind cryptographic keys, which compromises the integrity of the fingerprint verification result (*Inadequate Cryptographic Validation*), ③ not requiring authorization during the deactivation of fingerprint protection (*Unauthorized Fingerprint Deactivation*), and ④ failing to revalidate existing sessions when fingerprints registered on the device are altered (*Mishandled Fingerprint Updates*).

We then develop a static analysis tool to automatically detect all four types of misuses, which first assesses the apps' intentions with FpAuth and identifies the specific stages of FpAuth lifecycles. The tool then proceeds to pinpoint the presence of FpAPI misuse patterns within FpAuth apps. In all 1,333 FpAuth apps, we find that 1,295 (97.15%) of them contain at least one type of misuses. More concerning is that 251 (18.83%) of them exhibit all four types, cumulatively accounting for over 109 billion downloads[2] and impacting a vast number of users globally.

Delving deeper into the details of each misuse, our measurement study reveals several instructive findings. Notably, despite the release of the newer, more secure version of FpAPIs over five years ago, nearly half (47.79%) of the apps assessed still rely solely on the older, more vulnerable version. Additionally, the official support library, designed to facilitate the secure and efficient use of these APIs, is underutilized, particularly in apps originating from China. Another significant observation is the tendency among some of the developers to prioritize functionality over security. This is evident in cases where developers have deliberately configured the specific FpAPIs to overlook the fingerprint updates on mobile devices. These findings highlight the pressing need for increased attention to the security of FpAPIs by both app developers and AOSP maintainers.

Our manual inspection of real-world apps has uncovered that the identified misuses of FpAPIs could lead to severe risks, such as unauthorized account takeovers, privacy violations, and financial losses. We have responsibly disclosed these vulnerabilities and received 184 CVE IDs and 19 CNVD IDs. For instance, consider *Binance*, a leading global cryptocurrency exchange and blockchain platform, which has been downloaded over 50 million times from Google Play. This app fails to properly manage fingerprint updates; consequently, when a new fingerprint is added to the device, Binance permits access without requiring re-authentication of the user. This oversight could potentially allow unauthorized individuals to execute transactions, leading to financial losses to legitimate users. We report this issue and it is assigned CVE-2024-31695.

[2]The downloads data is crawled from Google Play and the Huawei app market, which serve as the sources of our app dataset.

In summary, this paper makes the following contributions:

- We conduct the first comprehensive study on the entire lifecycle of FpAuth in real-world Android apps, categorizing the threat models of FpAuth and summarizing four common FpAPI misuse types.
- We design and implement tools based on static analysis to automatically identify FpAuth apps and detect FpAPI misuses in these apps.
- Our empirical measurement of real-world apps reveals that despite the critical security role FpAPIs serve, their deployment is often flawed, leading to significant security breaches and risks. The code and data used in this study are publicly available at https://github.com/FpAuth/FpAuthAnalysis.

Organization. § II describes the workflow of FpAuth and the design details of FpAPIs. In § III, we introduce the method for identifying FpAuth apps and present their characteristics. § IV elaborates our principled analysis workflow, the FpAPI misuse details, and the consideration of app intentions, while § V shows the methodology for detecting four misuse types. Then § VI demonstrates and analyzes the detection results. § VII and § VIII make discussions and summarize related works, respectively. Finally, § IX concludes this work.

II. Background

A. Fingerprint Authentication (FpAuth)

Android utilizes dedicated fingerprint sensors and a Trusted Execution Environment (TEE) as the underlying foundation to provide robust fingerprint recognition capabilities. It offers high-level APIs for apps to perform user authentication based on fingerprints. Figure 1 illustrates a fingerprint-matching process and the entities involved in it.

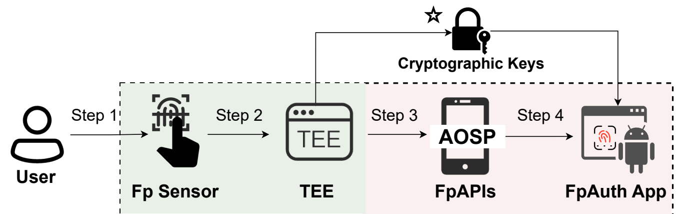


Fig. 1. The overall workflow of FpAuth in Android. The green area is resilient against OS-level attackers, while the pink area is not. Note that we abbreviate fingerprint to "Fp" for ease of representation.

In this workflow, users only interact with fingerprint sensors (Step 1), and the fingerprint data are only securely stored and matched in TEE (Step 2). That is, even if the Android OS is compromised, the fingerprint data will be secure. After matching the fingerprint within TEE, the result indicating the success or failure of fingerprint verification is passed through the Android framework (Step 3) to the apps through the FpAPIs (Step 4).

Table I
Two versions of Android fingerprint APIs.

Fingerprint API	Available API Level	Supported Features	Required Permission	Support Library
android.hardware.fingerprint.FingerprintManager	23 - 27	fingerprint	USE_BIOMETRIC or USE_FINGERPRINT	android.support.v4.hardware
android.hardware.biometrics.BiometricPrompt	28 - now	fingerprint, face, iris	USE_BIOMETRIC	androidx.biometric

While an OS-level attacker cannot disrupt the fingerprint matching process conducted within the TEE[3] (the green area in Figure 1), they can introduce tampering risks to the result processing procedure within the Android OS and FpAuth apps (the pink area in Figure 1). Consequently, to meet the requirements of apps in countering OS-level attackers, Android provides a mechanism [11] that allows FpAPIs to be used in conjunction with cryptography keys (☆ in Figure 1), which are also secured through TEE or StrongBox [12] and resilient to OS-level attackers. In such cases, parties with the cryptographic key privileges can verify the fingerprint-matching result without interference.

A typical example is in the design of FIDO (Fast IDentity Online) [13], where the remote server of an app possesses a public key, and the corresponding private key resides within the TEE of the user's mobile device. When fingerprint verification is completed within the TEE, the challenge from the server is signed using the private key and sent back to the remote server. This way, the app's remote server can ascertain the fingerprint verification result using the public key, immune to OS or network attackers.

However, the binding of FpAPIs and cryptographic APIs, while enhancing security, also introduces added complexity, making it more challenging for developers to use them correctly. Our discussion on developers' feedback in § VI-C shows that developers face difficulties in this issue according to their expressions in responses and their actions to deal with it.

B. Android Fingerprint APIs (FpAPIs)

FpAPI Versions. As a novel feature, Android FpAPIs have undergone a noticeable version change. As listed in Table I, there are two major versions of FpAPIs: 1) The first version, namely `FingerprintManager` APIs, was introduced in AOSP 6 (API level 23) and deprecated in AOSP 9 (API level 28); 2) The second and currently used version, namely `BiometricPrompt` APIs, was issued in AOSP 9 to replace the first version. AOSP also provides support libraries, such as `androidx.biometric` library, to ensure compatibility across different devices.

Compared to the first version, the newer version supports more biometric features such as face and iris. More importantly, in the first version, developers need to implement FpAuth UIs by themselves, which may be easily hijacked by local malicious apps [8, 9]. Thus in the second version, AOSP fixes this problem by providing developers with a unified and easy-to-use authentication user interface, effectively mitigating UI spoofing attacks [8, 9].

[3]In general, TEE is considered secure, and this paper does not consider the scenarios where TEE is compromised.

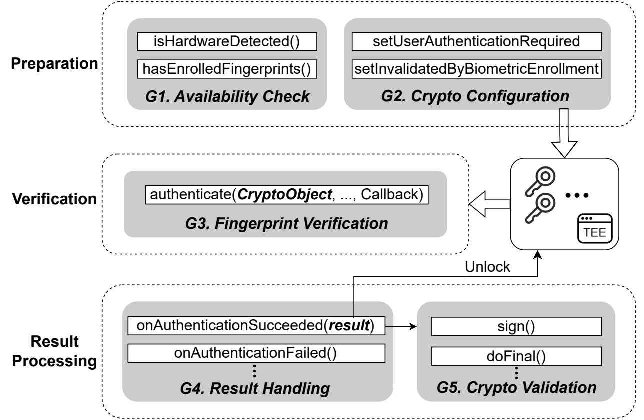


Fig. 2. Detailed FpAPIs in Android. Note that the functionality of these two API versions is roughly the same, and `FingerprintManager` APIs are used for illustration here.

FpAPI Details. Despite the above differences, the two versions of FpAPIs remain fundamentally consistent in design principles and usages. As shown in Figure 2, the process for apps conducting FpAuth can be divided into 3 main phases with 5 groups of FpAPIs: 1) During the *Preparation* phase, FpAPIs are called to check the availability of hardware and fingerprints, and to set up the cryptographic configurations. 2) In the *Verification* phase, authentication APIs are called to verify the provided fingerprint. 3) In the *Result Processing* phase, FpAPIs are called to handle the authentication result and validate it with cryptographic operations. We discuss the details of each group of FpAPIs below.

G1. Availability Check. Before initiating fingerprint verification, apps need to perform preliminary device checks, including checking the availability of the fingerprint scanner with `isHardwareDetected` and the existence of enrolled fingerprints with `hasEnrolledFingerprints` to ensure the feasibility of FpAuth.

G2. Cryptographic Configuration. In the preparation phase, apps can also prepare a configured cryptographic key bound with FpAuth to confirm the authenticity of the verification result, as illustrated in § II-A. Security-related key configuration here involves two APIs required to be set appropriately. `setUserAuthenticationRequired` binds the cryptographic key with fingerprint verification, and `setInvalidatedByBiometricEnrollment` configures that a newly enrolled fingerprint or deleting all existing fingerprints will in-

validate the key, allowing apps to identify fingerprint updates.

G3. Fingerprint Verification. Apps can start fingerprint verification by calling `authenticate` method, where the reference to the cryptographic key can be passed in via `CryptoObject` parameter. Then by comparing the provided fingerprint template with the ones enrolled on the device, the system will get a matching result and pass it back to the app.

G4. Result Handling. After verifying the fingerprint, the Android framework will inform the app of the result by invoking the given callback. For example, the invocation of `onAuthenticationSucceeded` means the user has successfully passed the fingerprint verification and apps can perform subsequent operations in the callback.

G5. Cryptographic Validation. As previously stated, apps need to combine with a cryptographic key to be resilient against OS-level attackers. In this step, when previously `setUserAuthenticationRequired` is set to `True`, only a successful user fingerprint verification will unlock the use of the key. Then the app can use this key to do cryptographic operations, such as signing a message (using the API `sign`) and securely verify the result on its server.

Overall, the Android FpAPIs offer robust capabilities for FpAuth apps to defend against attacks. However, this requires the app to appropriately set several groups of FpAPIs, whose accompanying complexity may cause misuses of these APIs.

III. FpAPI Measurement Study

In this section, we conduct a measurement study on the popularity of FpAuth and how FpAPIs are used in real-world Android apps.

A. Identifying FpAuth Apps

First, we develop an automatic tool to identify real-world apps with fingerprint-based authentication. The overall idea is first to find all FpAPI invocations and then conduct a reachability analysis to find a path from the entry points to these invocations, as shown in Figure 3.

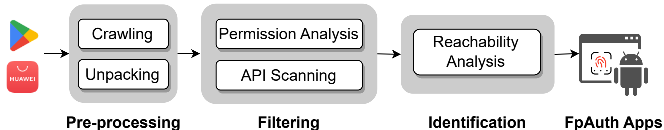


Fig. 3. Methodology of FpAuth app identification.

Pre-processing. We first use crawlers to collect popular apps from app markets. Upon collecting apps, we notice that some are protected by packing, making static analysis impractical. Therefore, we first make an effort to unpack these apps using frida-dexdump [14]. Any successfully unpacked apps are then included in our subsequent analysis.

Permission & API Filtering. we apply a preliminary filter based on the requested permissions and FpAPI invocations to narrow down the scope. Apps implementing FpAuth must request either `USE_BIOMETRIC` or `USE_FINGERPRINT` permission, as shown in Table I. Next, we use static analysis to search for the presence of FpAPIs in the app's code. Specifically, we create a list containing all versions of FpAPIs and scan the app's decompiled code with Apktool [15] to filter out the apps without invoking any FpAPIs.

Reachability Analysis. After API scanning, we obtain a candidate list of FpAuth apps. However, we observe that dead code may exist in these apps, e.g., integrating FpAuth-related third-party SDKs but not using FpAuth functionality. To get a more precise result, we further conduct a reachability analysis to find a path from the entry points of the app to the FpAPI invocations. In detail, we construct the call graphs for each candidate app based on state-of-the-art static analysis tools, including Soot [16] and FlowDroid [17]. Then we backtrack the call graphs, from the invocations of the FpAPIs, to check if they are initially called by the app's entry points, namely the exported components stated in its Manifest files. In this way, we can exclude apps that have FpAPI invocations but do not provide active FpAuth functionality.

Evaluating the Identification. We employ a best-effort approach to manually evaluate the identification result. First, for each reported FpAuth app by our static analysis, our team of security experts will execute the app, engage in the registration and sign-in process, test the app, and seek evidence of active FpAuth functionality. However, previous works [10, 18] have shown that several challenges exist for manual dynamic testing, such as the lack of certain prerequisites to register an account, e.g., a bank card for a financial app. Consequently, in cases where we encounter obstacles in creating accounts and conducting comprehensive testing, we resort to a best-effort manual examination of the app's decompiled code, app description in the market, privacy policies, and thorough online searches to check the presence of FpAuth functionalities. For example, the official website of the tested app, as well as user comments[4] may indicate that this app indeed supports FpAuth.

We randomly select 100 identified FpAuth apps (from 1,333 reported in Table II) and another 100 apps not recognized as FpAuth apps but containing FpAPIs (from 2,546 apps but not in 1,333 apps of Table II), to build the validation set. Two security experts apply the above methods to determine whether FpAuth exists in these 200 apps, which costs them about one week to get the final result.

Result. Out of the 100 identified FpAuth apps, all of them are confirmed to have FpAuth functionality, resulting in a 100% true positive rate. For the other 100 apps with FpAPIs but not recognized as FpAuth apps, we find that 17 of them do have FpAuth functionality. The reasons for these false negatives (FN) include Soot/Flowdroid error or timeout (6 apps), Flutter development (a framework for cross-platform app development, 4 apps), native calls (4 apps), and strong obfuscation (3 apps). These problems are mostly caused by complex static analysis challenges, as also met in previous studies [10, 19, 20].

[4]More specifically, we search the user comments on the app market, and any comment indicating the app contains FpAuth functionality will be strong evidence.

Because our primary aim is to investigate misuse issues within real-world FpAuth apps, we consider this proportion of false negatives to be acceptable. We acknowledge that more complex static analysis tools could enhance the identification rate, but we view this as orthogonal work to our current research focus.

B. Characteristics of FpAuth Apps

Prevalence. We collect all the top and free Android apps in 37 categories from Google Play Store [21] and 16 categories from Huawei market [22], including 29,848 and 35,238 apps respectively, as shown in Table II. Out of the total 65,086 apps, 5,545 apps request FpAuth-related permissions. Among them, 975 apps are equipped with packers that cannot be unpacked using existing tools [14]. Among the remaining 4,570 apps, our static analysis tool identifies that 2,546 of them (55.71%) contain FpAPIs. This observation highlights a substantial concern: a significant portion of apps (44.29%) request FpAuth-related permissions without utilizing any FpAPIs. This underscores the issue of excessive permission collection, consistent with prior research findings [10, 23]. Finally, our reachability analysis confirms that 1,333 of them are FpAuth apps.

Table II
FpAuth apps identification results.

Dataset	Total	Permission Analysis		API Scanning	Reachability Analysis
		unpacked	packed		
Google Play	29,848	2,043	272	1,597	786*
Huawei Market	35,238	1,352	1,878	949	547
Sum	65,086	3,395	2,150	2,546	1,333

* We have removed 14 redundant apps that also occurred in the Huawei Market according to the unique app package name.

Distribution & Significance. Figure 4 shows the distribution of FpAuth apps across different categories. It is evident that the majority of these apps belong to feature-rich or payment-involved categories such as business, lifestyle, socialization, and shopping. Additionally, health and sports apps often require fingerprint scanning in their functionalities. Consequently, considering the high sensitivity of these categories, the security of FpAuth implementations holds substantial significance for safeguarding critical aspects such as user privacy and financial security.

These apps collectively have over 218 billion installations, with an average of more than 164 million installations per app. Consequently, any instances of FpAPI misuses within these apps could potentially affect a large user base.

Origins of FpAuth Implementations. We also explore the origins of FpAuth implementations in these apps, i.e., where are the FpAuth code from. We first collect the package names of the code where the FpAPIs are invoked and cluster these package names based on their prefixes. These clustered groups, inspired by previous works [24], after filtering out support libraries and obfuscated ones, provide insights into the origins of the FpAuth code.

Table III demonstrates the top 10 origins employed by FpAuth apps in two markets. Through manual analysis of

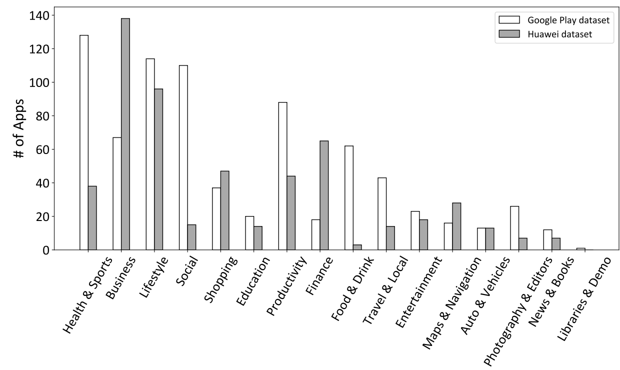


Fig. 4. FpAuth apps category distribution. Note that we merge similar categories from Google Play and Huawei Market into 16 categories.

Table III
Top 10 FpAuth implementation origins used in Google Play & Huawei Market apps. Note that some origins may have more than one prefix.

Origin in Google Play	# Apps	Rk.	Origin in Huawei Market	# Apps
org.telegram.messenger	24	1	com.baidu.sapi2/wallet	55
com.epic.patientengagement	21	2	com.tuya.security/smart	41
com.github.ajalt	21	3	com.wei.android	28
de.niklasmerz.cordova	12	4	com.alipay.security	25
com.americanwell.android	12	5	cn.org.bjca	23
io.flutter.plugins	10	6	com.tencent.soter	14
com.salesforce.androidsdk	9	7	us.zoom.androidlib	11
com.oath.mobile	8	8	ctrip.android.pay	11
com.visa.checkout	8	9	com.meituan.android	9
com.oblador.keychain	7	10	com.rnfingerprint.FingerprintHandle	7
Sum	132 (16.79%)	-	Sum	224 (40.95%)

these origins, we find that the FpAuth code may come from dedicated FpAuth SDKs, open-source repositories on Github, and apps developed by the same large companies, etc. Apps using these origins account for 16.79% in Google Play and 40.95% in Huawei FpAuth apps, implying that if there are misuse issues in them, a considerable number of apps may inherit such issues and associated risks, especially in the Huawei market.

Finding 1: FpAuth is widely used in popular and security-sensitive apps, making its security of great importance.

IV. The FpAPI Misuse Problem

In this section, we aim to figure out various types of FpAPI misuses present in real-world Android apps through a principled analysis approach.

A. Principled Analysis on FpAPI Usage

As shown in Figure 5, our principled analysis approach is launched with the exploration of three fundamental questions: *Q1: Lifecycle Stages*, what are the key stages in the FpAuth lifecycle in real-world apps? *Q2: Threat Model*, who are the potential attackers at these stages? *Q3: Possible Misuse*, what are the possible misuse scenarios for these stages within the context of the threat models?

Q1. What are the key stages in the FpAuth lifecycle in real-world apps? By studying the FpAPI documents and investigating real-world FpAuth apps, as well as our

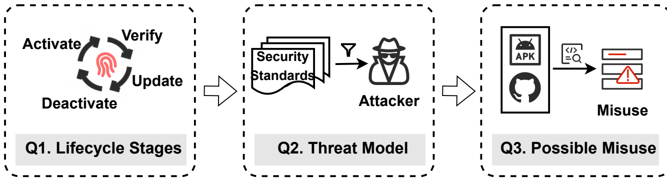


Fig. 5. Our principled analysis workflow.

experience of using smartphones with fingerprint sensors, we sum up the following key stages in the fingerprint usage lifecycle: 1) Activation, 2) Verification, 3) Updates, and 4) Deactivation.

Specifically, in the *Activation* stage, the app activates the FpAuth functionality by checking the environment and verifying the user's identity. In the *Verification* stage, the app uses FpAPIs with the help of correctly configured cryptographic APIs to finish the fingerprint verification, as demonstrated in Figure 1. After that, chances are that the enrolled fingerprint templates on the device may *Update*, i.e., adding new fingerprints or deleting fingerprints by the device users. Under such circumstances, the app should be responsible for effectively handling potential impacts to FpAuth caused by fingerprint updates. For example, if new fingerprints are added, potentially by someone other than the account owner, apps may want to re-verify the user's identity for account protection. Finally, in the *Deactivation* stage when users no longer wish to use FpAuth, the app should carefully verify whether such operations are initiated by legitimate users and deactivate the FpAuth functionality.

Through our investigation of this question, we have observed that previous research on fingerprint security has mostly focused on the *Verification* [8–10] stage while overlooking the significance of the other critical phases, such as *Updates* and *Deactivation*.

Q2. Who are the potential attackers at these stages?

By categorizing potential attackers and assessing their capabilities, we investigate the threat models that these key stages may encounter. We have identified four types of attackers that require attention, including 1) Local Software-level Attacker, 2) Local OS-level Attacker, 3) Physical Curious Attacker, and 4) Physical Intimate Attacker.

Specifically, a *Local Software-level Attacker* is able to install malicious apps on the victim's device (i.e., app-wide) or embed malicious code like SDKs in normal apps (i.e., in-app), thus they can induce the victim to perform fingerprint verification through UI attacks [9, 10]. A *Local OS-level Attacker*, also known as a root attacker, can compromise the Android system of the victim, allowing them to inject malicious code and hijack the execution flow of the FpAuth apps. We also consider physical attackers who can access the victim's mobile device, which is practical in daily social interactions. A *Physical Curious Attacker* may have the chance to access the victim's unlocked device. In some cases, they may have a one-time opportunity to access the target app, e.g., when the app is unlocked, but seek to gain permanent access to that app whenever the attacker can use the victim's unlocked device again.

On the other hand, we also consider a *Physical Intimate Attacker*, who not only has physical access to the victim's device but also knows the locking PIN, which may be obtained through shoulder-surfing attacks [25], guessing attacks [26], or any other ways leaked by the victim. In this scenario, the attacker aims to bypass the FpAuth protection, for example, to execute stealthy transactions.

Q3. What are the possible misuse scenarios for these stages within the context of the threat models? To gain a deeper understanding of fingerprint security within the context of the aforementioned threat models, we conduct a pilot study on real-world FpAuth apps.

Pilot Study Dataset. We collect a representative set of apps, including open-source and closed-source ones, to manually inspect the FpAuth implementations in them. First, we select 1 app from each of the previously identified 20 clustered origins (Table III), which can represent a substantial portion of FpAuth apps. Then we collect open-source FpAuth apps from a famous open-source Android app repository, F-Droid [27]. Specifically, we search FpAuth-related permissions and APIs and manually verify the existence of FpAuth in these apps, resulting in 14 apps. Finally, we randomly select 3 apps from each of the 2 app markets in our dataset. The above steps result in a total of 40 representative FpAuth apps, covering all app categories identified in our analysis. This dataset is presented in the Appendix (Table XIII).

Manual Inspection. We then conduct a manual inspection of each key stage within these apps, taking into account all potential attackers as mentioned earlier. In particular, we examine how these apps utilized FpAPIs, cross-referencing their usage with the official documentation to validate and identify any issues or discrepancies that may exist. As a result, we are able to summarize four common misuse types, as listed in Table IV.

Table IV
Overview of FpAPI misuses and corresponding threat models.

Misuse	Threat Model
M1. Obsolete API Usage	Local Software-level Attacker
M2. Inadequate Cryptographic Validation	Local OS-level Attacker
M3. Unauthorized Fingerprint Deactivation	Physical Curious Attacker
M4. Mishandled Fingerprint Updates	Physical Intimate Attacker

Among all the misuse types, *M1. Obsolete API* and *M2. Weak Crypto* have been partially touched in previous works [8, 10], while *M3. Unauthorized Deactivation* and *M4. Mishandled Updates* are first proposed by this work. We discuss the details of each misuse in the next subsection.

B. FpAPI Misuse Types

We discuss the details of the four misuse types in this subsection. Note that the misuses are determined considering the specific app intentions.

```
1 | // Pattern 1: Obsolete API Usage
  | fingerprintmanager.authenticate(...);   // Misuse: Only using the deprecated version
  | (FingerprintManager) of FpAPIs, instead of the new version (BiometricPrompt)
```

Fig. 6. Example of the misuse pattern of obsolete API usage (M1).

M1. Obsolete API Usage. This misuse type refers to the app developers still using the deprecated, insecure version of FpAPIs, i.e., `FingerprintManager` (Figure 6). While it has been highlighted in previous work [8] that this version of the APIs is insecure, the proportion of real-world implementations that have migrated to the newer version and corresponding support libraries remains unclear.

M2. Inadequate Cryptographic Validation. This type of misuses refers to the inability to utilize cryptographic capabilities adequately to securely verify the authenticity of fingerprint verification results. As detailed in § II-B, the app receives fingerprint verification results from the Android framework. Even though the fingerprint matching process is secure within TEE, there remains a possibility of tampering during the transmission of verification results and processing in the apps, particularly by local OS-level attackers. Previous works [10] have discussed this problem, however, we observe that it is unnecessary to defend against OS-level attackers for some apps, and we will discuss this scenario in § IV-C.

```
   | // Pattern 2: Null Cryptographic Object
1  | fingerprintmanager.authenticate(null);  // Misuse: Setting parameter crypto to
   | null causes FpAuth is not protected by cryptographic key

   | // Pattern 3: Failed Key Binding
2  | keyGenerator.init(KEY_NAME)
3  |    ...
4  |    .setUserAuthenticationRequired(false);  // Misuse: Not explicitly setting this
   | API to true causes failure to bind FpAuth with cryptographic key
5  | fingerprintmanager.authenticate(CryptoObject(cipher.init(KEY_NAME)), ...);

   | // Pattern 4: Failed Result Validation
6  | keyGenerator.init(KEY_NAME)
7  |    ...
8  |    .setUserAuthenticationRequired(true);
9  | fingerprintmanager.authenticate(CryptoObject(cipher.init(KEY_NAME)), ...);

10 | public void onAuthenticationSucceeded(result) {
11 |    log.info("FpAuth succeeded!");  // Misuse: Not validating the result using
   | cryptographic key causes the result can be forged
12 | }
```

Fig. 7. Examples of three misuse patterns of inadequate cryptographic validation (M2).

To bind cryptographic keys with FpAuth, developers need to configure several FpAPIs correctly. During this process, three common code patterns may arise: *Null Cryptographic Object* (as seen on Line 2 in Figure 7), *Failed Key Binding* (Line 7), and *Failed Result Validation* (Line 21). On the other hand, the best practice is to utilize the appropriate cryptographic key, set the parameter of `setUserAuthenticationRequired` to `True`, and use the cryptographic key to sign the results and validate on the server side.

M3. Unauthorized Fingerprint Deactivation. This type of misuses refers to apps not verifying the operator's identity when FpAuth is deactivated, causing an unauthorized bypass of FpAuth. Initially, an app using FpAuth aims to protect itself from being used by unauthorized users, but if this protection method can be easily disabled, it becomes ineffective. In practice, a physical curious attacker may gain access to the victim's phone with the FpAuth app unlocked, e.g., when the victim shows a message/picture on the app to nearby users, so the attacker can deactivate the FpAuth protection. Subsequently, whenever the attacker obtains the victim's unlocked phone again, they can use the app freely. This poses the risk of potential information leakage and even financial loss, especially when considering a scenario where the victim's app is a payment app with FpAuth enabled to secure transactions. The code pattern for this type of misuses is shown in Figure 8.

```
   | // Pattern 5: Unauthorized Fingerprint Deactivation
1  | Switch fpSwitch = findViewById(R.id.fingerprintSwitch);
2  | listener = new OnCheckedChangeListener() {
3  |     public void onCheckedChanged(..., boolean isChecked) {
4  |         if (isChecked) {     // activating FpAuth
5  |             ...
6  |         } else {
7  |             ... // Misuse: Not authenticating when deactivating FpAuth
8  |         }
9  |     }
10 | fpSwitch.setOnCheckedChangeListener(listener);
```

Fig. 8. Example of the misuse pattern of unauthorized fingerprint deactivation (M3).

M4. Mishandled Fingerprint Updates. This type of misuses refers to a situation where the enrolled fingerprints on the device have changed, but the FpAuth apps fail to correctly respond to such change, leading to FpAuth being weakened or totally bypassed. In this case, we consider the presence of a physical intimate attacker, who manages to know the PIN code of the victim's device, using methods described in Q2 of § IV-A. Originally, the FpAuth is protected by fingerprint verification, so even if the attacker knows the PIN and can unlock the victim's device, they cannot directly use the functionality, e.g., payment, protected by FpAuth. However, if this misuse type exists, the attacker can take advantage of it by adding their own fingerprint (after first verifying the PIN), thereby gaining access to the FpAuth-protected functionality. The attacker may also choose to remove all enrolled fingerprints on the device. If the FpAuth app fails to detect these changes, there is a risk that the attacker may gain unauthorized access to the target app. Below we discuss the details of two kinds of fingerprint updates: addition and removal.

1) Addition. In Android, apps can perceive fingerprint additions on the device by using a cryptographic key bound to FpAuth (as discussed in *M2. Weak Crypto*). This perception requires accurate use and configuration of the relevant APIs. In this context, patterns 2 and 3 in Figure 7 also contribute to *M4. Mishandled Updates*, as they do not use a key or fail to bind the key to FpAuth. Also, another pattern contributing to M4 is shown in Figure 9, where despite correctly binding the key, mistakenly setting `setInvalidatedByBiomet-`

```
    // Pattern 2: Null Cryptographic Object
    // Pattern 3: Failed Key Binding

    // Pattern 6: Failed Enrollment Handling
1   keyGenerator.init(KEY_NAME)
2     ...
3     .setUserAuthenticationRequired(true)
4     .setInvalidatedByBiometricEnrollment(false);  // Misuse: Setting this API to
    false causes failure to handle the enrollment of new fingerprints
5   fingerprintmanager.authenticate(CryptoObject(cipher.init(KEY_NAME)), ...);
```

Fig. 9. Example of the misuse patterns of mishandled fingerprint updates (M4-Addition).

`ricEnrollment` API to false explicitly results in failing to handle the enrollment of new fingerprints.

```
    // Pattern 7: Failed Removal Handling
1   if (fingerprintmanager.hasEnrolledFingerprints()) {
2     ...
3   } else {
4     ... // Misuse: Not authenticating when all fingerprints are removed
5   }
```

Fig. 10. Example of the misuse pattern of mishandled fingerprint updates (M4-Removal).

2) Removal. A physical intimate attacker can also delete all enrolled fingerprints on the victim's device. If the FpAuth app fails to detect or mishandles this change, the attacker may gain access to the FpAuth app. A common pattern of this misuse is depicted in Figure 10, where the app checks for the existence of fingerprints on the device. If no fingerprints are found, it proceeds to unlock the protected functionality, such as displaying sensitive information. The best practice is to provide a fallback authentication method (such as setting a password), to ensure authentication protection in case no fingerprints for FpAuth.

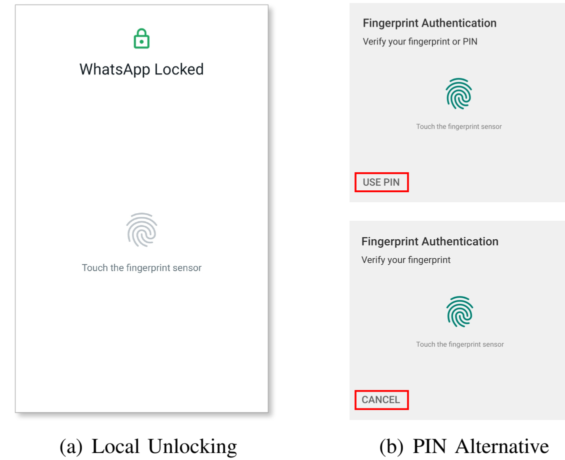
C. Consideration of App Intentions

In real-world deployments, apps may have their specific intentions. Our study categorizes app intentions concerning FpAuth from two different perspectives: usage scenarios and authentication choices, as follows.

First, apps may employ FpAuth for two different usage scenarios:

- **Local Unlocking**, where the FpAuth is used for local data protection as shown in Figure 11(a). In this case, it's unnecessary to assume that the app should defend against OS-level attackers since if these attackers gain OS-level privileges, they can access all local data of the app without needing to specifically bypass FpAuth. This type of app should focus on the other three types: *M1. Obsolete API*, *M3. Unauthorized Deactivation*, and *M4. Mishandled Updates*.
- **Remote Authentication**, where the FpAuth is used for remote service protection such as payment and login. In this case, apps should consider all four misuse types.

Second, when implementing FpAuth, developers may have different authentication choices:



(a) Local Unlocking (b) PIN Alternative

Fig. 11. FpAuth examples with intentions of *Local Unlocking* and *PIN Alternative*. Note that (a) shows a fingerprint lock when entering the app.

- **PIN Alternative**, which configures the device PIN as an alternative to FpAuth, allowing users to choose to authenticate using FpAuth or PIN as shown in the upper panel of Figure 11(b). In this case, apps may not want to defend against physical intimate attackers, especially for those apps that do not contain highly sensitive operations. They may assume that the device PIN is the last level of security protection, and an FpAuth process does not provide additional protection over the device PIN. As a result, they can choose to explicitly set several relevant APIs listed in Table VI, so that the FpAuth is allowed for fallback to the lock screen PIN [28]. This type of app should focus on *M1. Obsolete API*, *M2. Weak Crypto*, and *M3. Unauthorized Deactivation*.
- **Fingerprint-only**, which disallows the device PIN to be an alternative to the FpAuth, requiring users to use only FpAuth for authentication as shown in the lower panel of Figure 11(b). In this case, apps should consider all four misuse types.

Table V
Summary of four types of misuses and their related app intentions, FpAuth lifecycle stage, and code patterns.

Misuse	App Intention	Stage	Code Pattern
M1. Obsolete API	-	Verification	P1
M2. Weak Crypto	NOT local unlocking	Verification	P2, P3, P4
M3. Unauthorized Deactivation	-	Deactivation	P5
M4. Mishandled Updates	NOT PIN alternative	Updates	P2, P3, P6, P7

Table V summarizes the mappings between each misuse type, its conditions in the context of app intentions and FpAuth lifecycle stages, and the corresponding code patterns. We will refer to this table to detect real-world FpAPI misuses.

V. FpAPI Misuse Detection

This section will describe how we automatically detect the FpAPI misuses in FpAuth apps based on static analysis. Then, we evaluate the effectiveness of the detection tool.

A. Detecting FpAPI Misuses

Although we have summarized the identified misuses into specific patterns, the actual implementations of these patterns in real-world apps vary significantly. Furthermore, some apps may employ obfuscation, encryption, or other techniques to obscure their implementation details, further complicating detection efforts. Thus, detecting these misuses accurately is challenging. Nonetheless, we strive to overcome these challenges to ensure the efficiency and accuracy of our misuse detection methods.

Approach Overview. We implement our misuse detection based on static analysis. Although dynamic analysis can yield more accurate results, it is limited by accounts login requirements [10, 18], such as the need for a bank card during registration. Besides, static analysis is more efficient for detecting misuses on such a large scale. As shown in Figure 12, the detection of the identified four types of misuses includes three major steps: 1) *Intention Analysis*, 2) *Stage Identification*, 3) *Pattern Matching*.

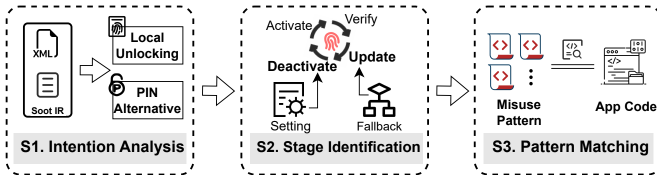


Fig. 12. Methodology of FpAPI misuse detection based on static analysis.

1) Intention Analysis: We begin by identifying the two specific types of app intentions outlined in Table V, namely *Local Unlocking* and *PIN Alternative*. The idea involves combining the semantics of both the UI and code to infer the intentions of FpAuth apps.

To identify FpAuth used for *Local Unlocking*, we first use static analysis to extract relevant FpAuth UI and code details. Inspired by prior work on recognizing login functions [19], we traverse through the FpAuth code call chains and UI elements texts to identify the semantics of "unlock". The specific process involves: a) Searching for terms like "fingerprint lock" and "unlock" to consider them as *Local Unlocking*. b) Excluding references to non-local types like "payment" and "login". c) In cases where the exact semantics are unclear, we categorize them as *Local Unlocking* conservatively.

Table VI
APIs related to *PIN Alternative* configurations.

API	PIN Alternative Conditions
setNegativeButton(...)[†]	Not invoking this API
setDeviceCredentialAllowed(boolean allowed)	allowed = `True`
setAllowedAuthenticators(int authenticators)	authenticators & DEVICE_CREDENTIAL != 0

[†] Corresponds to `setNegativeButtonText` API in `androidx.biometric` library.

To identify FpAuth used as *PIN Alternative*, we apply static data flow analysis to determine the exact parameter values for a couple of relevant APIs based on the conditions in Table VI. Note that the APIs offered by AOSP are a little intricate. For example, if developers configure the negative button in Figure 11(b) using `setNegativeButton` API, they will not be able to utilize the latter two APIs to set FpAuth as *PIN Alternative*. Otherwise, developers can utilize one of the latter two APIs in Table VI to designate FpAuth as *PIN Alternative*.

2) Stage Analysis: We also need to identify the deactivation and updates stages to detect *M3. Unauthorized Deactivation* and *M4. Mishandled Updates*. The core idea is to analyze the context of the related UI and code where FpAuth is conducted.

For the deactivation stage, we find that most apps provide the "Deactivate Fingerprint" functionality with a toggle switch in their "Setting" pages. Therefore, we use static analysis to firstly find such toggle switches and then locate their event handling functions (`onCheckedChanged`). We then employ backward analysis from `authenticate` API to the toggle switches, ending with the event callback `onCheckedChanged` of a switch UI element, and inspect whether FpAuth is performed when it is deactivated. Note that in cases where apps don't use `onCheckedChanged` method, we look for features related to the switch status like "checked", and utilize semantic features to identify custom switch component implementations.

For the updates stage, we aim to determine whether the removal of all fingerprints has triggered unauthorized access. Specifically, we associate the removal of all fingerprints with the invocation of `hasEnrolledFingerprints` or `canAuthenticate` API, which checks the presence of enrolled fingerprints. We then perform path-sensitive control flow analysis to trace the return values of these APIs and determine whether the apps have allowed access to sensitive resources or operations without any authentication protection.

3) Pattern Matching: We perform pattern matching after the app intention analysis and stage analysis. According to the 7 patterns summarized in § IV-B, we conduct static analysis to identify specific code patterns indicative of each misuse. This involves analyzing the app's bytecode to search for sequences of API calls and the value of API parameters or control flow structures that match the predefined patterns. For example, to detect *M2. Weak Crypto*, we analyze code segments where `authenticate` API is invoked without proper cryptographic key configuration or validation. Similarly, for *M3. Unauthorized Deactivation* we look for instances where the event handling function of the UI component (switch) in the specific context is previously invoked on the call chains of `authenticate` API. By systematically matching these patterns against the app's code, we can effectively identify instances of each misuse.

4) Implementation Details: We implement the control flow and data flow analysis based on state-of-the-art tools, i.e., Soot [16] and FlowDroid [17]. We parse the APKs and construct the call graphs referring to these tools. In our misuse detection, we conduct data flow analysis to track the variable value based on `Shimple` intermediate representation (single static assignment form). Our static analysis approach considers various aspects critical to the accurate detection of FpAPI misuses. Specifically, we include the data flow and implicit

control flow, covering multi-threading constructs, some inter-component communications, lifecycles of activities, and common Android callbacks. During this process, we observed that the FpAuth functionalities are often implemented in Android fragments [29], a special type of component not modeled by these tools. Therefore, we extend the modeling of Android components to incorporate fragment lifecycles to build a more complete call graph for FpAPI misuse detection. Moreover, while code obfuscation is a prevalent issue in static analysis, our misuse detection targets FpAPIs, which are system-level APIs typically not subjected to obfuscation by apps. The code of this static analysis tool is open-sourced at https://github.com/FpAuth/FpAuthAnalysis to ease reproduction and foster subsequent studies.

B. Evaluating Misuse Detection

Evaluating the Detection. To validate the effectiveness of the proposed misuse detection method, we manually test all identified 1,333 FpAuth apps. As a result, we have successfully tested the FpAuth functionalities of 334 apps. The remaining apps could not be tested due to special restrictions including requiring a bank card, ID card, phone number of certain countries, area restrictions, etc., servers not responding, and legacy FpAuth implementations that cannot be dynamically triggered. We have two security experts to manually evaluate these apps, where all of these apps are cross-validated. They are asked to dynamically run each app, finish account login, and test all the FpAuth lifecycles for potential misuses, for example, verifying if user authentication is provided when deactivating FpAuth in the app settings.

To help our security expert better detect FpAPI misuses, we also develop scripts based on dynamic instrumentation using Frida [30]. Specifically, we hook the different versions of FpAPIs to record their invocation details and simulate root attackers to tamper with the fingerprint verification result to assess *M2. Weak Crypto*. These scripts can facilitate our manual analysis.

Result. We calculate the precision and recall of the proposed detection method, by comparing with the results from the manual evaluation on the testable 334 apps, as shown in Table VII. We can find that the proposed method can achieve relatively high precisions and recalls, considering that FpAuth is tightly coupled with the specific logic and various implementations in real-world mobile apps.

Table VII
Precision and recall of our misuse detection tool.

Misuse	TP	FP	TN	FN	Precision	Recall
M1. Obsolete API	192	0	142	0	100.00%	100.00%
M2. Weak Crypto	187	42	6	3	81.66%	98.42%
M3. Unauthorized Deactivation	206	34	18	76	85.83%	73.05%
M4. Mishandled Updates	199	29	33	47	87.28%	80.89%

We then manually examine the false positives and false negatives in our detection tool, and find that they are mainly attributed to the limitations of static analysis. Among the false positives, the most common reason is that there are multiple redundant FpAuth implementations in the app, and the one with misuse remains unexecuted. Although we have integrated a reachability analysis to exclude dead code, chances are that the static analysis tool fails to determine complex branching conditions.

On the other hand, the false negatives are primarily due to code obfuscation and invocation of native code, which result in an incomplete call graph. Additionally, some apps may use customized UI components, such as using `ImageView` as the switch for deactivating FpAuth (*M3. Unauthorized Deactivation*). These challenges are inherent to all static analysis techniques. Thus, we consider them orthogonal issues and recommend more advanced static analysis methods for future work.

VI. FpAPI Misuse Analysis

In this section, we present the overall results of FpAPI misuses and break down each type of misuses with detailed analysis and case studies.

A. Overall Result

Table VIII displays the overall results of four types of misuses within the 1,333 identified FpAuth apps. From the table, we can infer several insights regarding the prevalence and distribution of these misuse types:

- *High Incidence of Misuse*: Nearly all FpAuth apps (97.15%) are found to have at least one misuse, with each type of misuses present in nearly or more than half of the apps, indicating widespread security concerns across the real-world FpAuth implementations.
- *Most Common Misuse Type*: The most prevalent type of misuses is *M2. Weak Crypto*, affecting 999 apps (74.94%). This suggests that this specific misuse is more challenging for developers to avoid, making it a critical target for further security enhancements.
- *Use of Obsolete FpAPIs*: Despite the new version of FpAPIs has been introduced for over five years by AOSP, nearly half of the apps (47.79%) persist solely using the deprecated version. This suggests potential barriers to adoption, such as compatibility issues or a lack of awareness about the advantages of the new FpAPIs version.
- *Multiple Misuses within Single Apps*: A significant number of apps (251, representing 18.83%) exhibit all four types of misuses, vulnerable to various types of attackers, endangering the FpAuth security.

Table VIII
Four types of misuses in 1,333 FpAuth apps.

Misuse	# Apps (Proportion)	# Misuse	# Apps (Proportion)
M1. Obsolete API	637 (47.79%)	1	179 (13.43%)
M2. Weak Crypto	999 (74.94%)	2	377 (28.28%)
M3. Unauthorized Deactivation	871 (65.34%)	3	488 (36.61%)
M4. Mishandled Updates	894 (67.07%)	4	251 (18.83%)
# Apps without Misuse			38 (2.85%)

These findings reveal that misuses of FpAuth are common issues in real-world apps, affecting a vast user base, as

evidenced by the cumulative download count exceeding 109 billion for the 251 apps exhibiting all four misuses. The total downloads for all 1,295 apps with any misuse surpasses 217 billion. The observation underscores the challenges developers encounter when implementing FpAuth securely and the critical need for improved security practices.

Given these widespread issues, particularly prevalent in business apps (201 misused apps), which are often security-sensitive, there is a clear necessity to enhance developer awareness and comprehension of FpAPI misuses. In the next subsection, we will explore each misuse in detail to offer deeper insights, aiming to enhance developers' awareness of FpAuth security practices.

Finding 2: Misuse of FpAPIs is widespread in real-world apps, underscoring the urgent need to enhance developers' understanding and ability to correctly use FpAPIs.

B. Breakdown of Misuse

1) M1. Obsolete API Usage: One intriguing observation from the overall results is that despite 5 years passing since the release of the more secure and reliable new version of FpAPIs, a significant number of real-world apps continue to only use the old, insecure version. Thus we look deep into the details of various aspects including the distribution of utilized API versions, and the prevalence of apps using different versions of APIs.

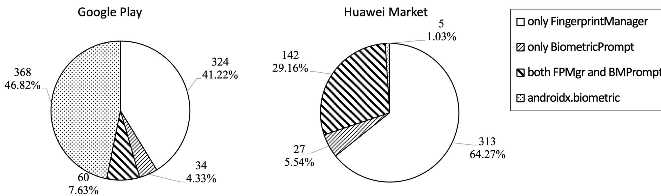


Fig. 13. Distribution of FpAPI version used by FpAuth apps.

- *Distribution of FpAPI Versions Used*: The distribution varies significantly between Google Play and the Huawei market as shown in Figure 13. Specifically, we find that *M1. Obsolete API* is more common in the Huawei market.
- *Usage of Recommended Library*: Google's recommended library `androidx.biometric`, designed to address both compatibility and security concerns, is underutilized by apps, especially in the Huawei market (only 1.03%) while Google Play exhibits a much higher usage (46.82%).
- *Prevalence of Apps Using Different FpAPI Versions*: Upon analyzing the popularity of these apps, we find that in Google Play, apps only using deprecated API (with *M1. Obsolete API*) average around 15 million downloads, lower than 41 million for those without M1. However, in the Huawei market, apps with M1 average 494 million downloads, significantly higher than the download count of apps without M1 (171 million). This shows a trend where popular Google Play apps favor the newer, more secure FpAPI version, while the Huawei market exhibits the opposite trend, potentially increasing security risks.

Case 1: Meituan (com.sankuai.meituan), which is a comprehensive lifestyle platform app in China offering diverse services like food delivery, and hotel reservations, with over 17 billion downloads[5] in the Huawei market, It offers FpAuth as one of the means of authorizing payments. Our analysis identifies *M1. Obsolete API* within its FpAuth implementation, specifically only using the insecure and deprecated `FingerprintManager` API. As elaborated § IV-B, apps utilizing this API are susceptible to the fingerprint-jacking attack where a malicious app can deceive the user into authorizing fingerprint verification for fraudulent payments, potentially resulting in financial losses for the user.

Finding 3: The adoption of the newer, more secure version of FpAPIs should receive increased attention from both the developers and AOSP.

2) M2. Inadequate Cryptographic Validation: Among all four types of misuses identified, this particular misuse is the most prevalent in apps. Table IX provides a detailed breakdown of the various code patterns that lead to this misuse. We discover that the most common code pattern, P2, accounts for 73.37% of cases, where developers directly omit the cryptographic parameters. This may imply that the majority of developers implementing FpAuth are not conscious of the need to combine keys and server-side verification to ensure the security of FpAuth.

Table IX
Distribution of the three patterns leading to inadequate cryptographic validation misuse (M2). Note that we exclude apps with the intention of *Local Unlocking*.

Misuse Pattern	# Apps
P2. Null Cryptographic Object	733 (73.37%)
P3. Failed Key Binding	67 (6.71%)
P4. Failed Result Validation	199 (19.92%)
Sum (with *M2. Weak Crypto*)	999

Case 2: 58 Tongcheng (com.wuba), which is an online classifieds marketplace app in China providing diverse services like job postings, and real estate, with over 10 billion downloads[6] in the Huawei market. The app uses FpAuth in a remote authentication scenario, specifically account login. However, 58 Tongcheng's FpAuth implementation exhibits misuse pattern 2 (Null Cryptographic Object). Through tracking the crypto parameter set in the `authenticate` API, it is eventually found that the value of this parameter is `Null`.

The existence of *M2. Weak Crypto* poses potential risks of taking over the user's account, compromising user data and privacy such as work and rental details, and even leading to financial loss. After reporting this issue, the developer

[5]In October 2022, the download data was obtained from https://appgallery.huawei.com/app/C5206.

[6]In October 2022, the download data was obtained from https://appgallery.huawei.com/app/C10026044.

confirmed and acknowledged our concerns, then promptly addressed it by removing FpAuth functionality on rooted devices to defend against OS-level attacks in the new app version.

Finding 4: The majority of FpAuth apps fail to use and configure cryptographic keys to defend against OS-level attacks.

3) M3. Unauthorized Fingerprint Deactivation: We identified that 462 apps perform authentication when deactivating FpAuth, while the remainder fail to properly authorize the deactivation operation. Interestingly, we discovered 120 apps that fail to perform FpAuth when deactivation but they do so when activation, as shown in Table X. This implies that developers grasp basic security concepts but struggle with their effective implementation. Conducting FpAuth solely during activation does not guarantee the integrity of FpAuth protection. It merely checks the availability of FpAuth, which can be easily achieved using relevant FpAPIs.

Table X
Verification practices when activating/deactivating FpAuth.

	Verified Deactivation	**Verified Activation Only**	**Verified None**
# Apps	462 (34.66%)	120 (9.00%)	751 (56.34%)

Case 3: WhatsApp (com.whatsapp). WhatsApp, a popular messaging app with 5 billion Google Play downloads, offers secure communication through end-to-end encryption. It provides high-strength protection through its fingerprint lock (requiring FpAuth to access the app). Specifically, users are required to authenticate via FpAuth every time they switch back to WhatsApp, even if it's running in the background. However, its FpAuth implementation exists *M3. Unauthorized Deactivation*, lacking authentication when deactivating the fingerprint lock. This misuse could enable attackers with physical access to the device to gain persistent access to sensitive user data like contacts and chat history.

Finding 5: More than half of FpAuth apps fail to verify the user's identity when deactivating FpAuth, leading to unauthorized access once FpAuth is illegitimately deactivated.

4) M4. Mishandled Fingerprint Updates: We list the examples of code patterns leading to this type of misuses in Table XI. Note that we consider an FpAuth app to exhibit M4 if it fails to correctly respond to fingerprint addition or removal. Overall, we find that the most common misuse pattern causing M4 is the failure to combine the cryptographic key with FpAuth (P2).

Furthermore, by delving into the details of this type of misuses, we can uncover some interesting observations:

- *Prioritization of Functionality over Security*: In real-world scenarios, developers may prioritize functionality over security, sometimes neglecting security altogether. For instance, we found that 17 apps, which disallow *PIN Alternative*, correctly configure `setUserAuthenticationRequired` API to `True`, but intentionally set `setInvalidatedByBiometricEnrollment` to `False` to ignore the fingerprint updates, leading to an unauthorized person who obtains the lock screen PIN can compromise the protection of FpAuth. Consequently, this decision, while focusing on basic functionality and user experience, results in mishandled fingerprint updates, potentially compromising security by allowing unauthorized access.
- *Insufficient Security Knowledge*: Some developers strive to uphold security measures but lack comprehensive security understanding and knowledge, resulting in *M4. Mishandled Updates*. We found that 5 apps intend to invalidate keys when a new fingerprint is added by setting `setInvalidatedByBiometricEnrollment` to `True`. However, they inadvertently nullify their security intentions by also setting `setUserAuthenticationRequired` to `False`, revealing a gap in their security knowledge.
- *Ambiguous Official Guides*: We found that samples in the Android developer guides [11] may contain ambiguities, potentially misleading developers. Specifically, the comments in the samples state that `invalidatedByBiometricEnrollment` is `True` by default. However, this default behavior only applies after explicitly setting `setUserAuthenticationRequired` to `True`, which could confuse developers and contribute to *M4. Mishandled Updates*.

Table XI
Distribution of the four patterns leading to mishandled fingerprint updates misuse (M4). Note that we exclude apps with the intention of *PIN Alternative*.

Misuse Pattern	**# Apps**
P2. Null Cryptographic Object	760 (85.01%)
P3. Failed Key Binding	90 (10.07%)
P6. Failed Enrollment Handling	17 (1.90%)
P7. Failed Removal Handling (Removal)	85 (9.51%)
Sum (with *M4. Mishandled Updates*)	894

* Note that the percentages add up to more than 100% because failed handling of either fingerprint addition (P2, P3, P6) or removal (P7) will result in *M4. Mishandled Updates*, and some apps fail in both cases at the same time.

Case 4: Binance (com.binance.dev). Binance, a global cryptocurrency trading app with 50 million downloads in Google Play, offers fingerprint lock protection when entering the app. Specifically, access to the app needs first unlock the device using the lock screen PIN and then unlock the app using a fingerprint. However, *M4. Mishandled Updates* (addition) in its FpAuth leads to degrading the dual protection to single, where physical intimate attackers can use the PIN to add their fingerprints on the device and bypass the FpAuth. Consequently, the intended dual protection is invalidated and unauthorized access to the victim's app account will lead to sensitive financial information leakage and even financial loss. This vulnerability has been accepted as CVE-2024-31695.

Case 5: YouFish (com.jz.youyu). YouFish, a finance management app with over 69 million downloads in the Huawei market, provides a fingerprint lock feature to protect the user's privacy. However, its implementation exists *M4. Mishandled Updates* (removal), specifically failing to offer fallback authentication when users delete all enrolled fingerprints. This oversight grants direct access to sensitive information, posing a risk of unauthorized access and financial data theft. Furthermore, this app developer's intention disallows *PIN Alternative* in FpAuth. However, this misuse undermines that intention, enabling the bypassing of FpAuth by removing all existing fingerprints using a lock screen PIN. This issue has been accepted as CNVD-2023-68954.

> **Finding 6**: Developers often prioritize functionality over security and may lack sufficient knowledge to implement FpAuth securely compounded by ambiguities in samples in Android developer guides, leading to critical security oversights.

C. Responsible Disclosure & Developer Feedback

We responsibly reported the misuse issues to the app developers. If we encounter difficulties in reaching them or if they do not respond promptly, we further report to CVE and CNVD. Specifically, we manually tested the FpAuth functionality of all the identified FpAuth apps to confirm the misuses before reporting. As a result, we have 334 remaining apps that we can test their FpAuth without any constraints.

We first responsibly reported to the developers of these 334 apps, and 15 of them acknowledged our disclosure and fixed the issues. Our detailed reports not only outline the identified misuses but also elucidate the inherent weaknesses in FpAuth security resulting from these misuses. We show the potential consequences of misuses to developers and provide practical and feasible repair suggestions for each misuse to assist them in addressing the misuse problems.

Result. For those we did not receive responses from developers promptly, we reported the issues to CVE (apps from Google Play) and its Chinese counterpart CNVD (apps from Huawei Market). Overall, we have obtained 184 CVE IDs, 19 CNVD IDs, and acknowledgments from 15 developers, as listed in Table XII. The specific details regarding the vulnerabilities refer to https://github.com/FpAuth/FpAuthAnalysis.

Table XII
The acknowledged vulnerabilities after our responsible disclosure.

	CVE ID	CVND ID	Developer
Number	184	19	15

Findings about Developers' Feedback. According to the responses we received, we found that understanding and addressing *M2. Weak Crypto* is more difficult for developers. Some developers express difficulty in fixing M2 in their responses. In addition, apps like 58 Tongcheng (a popular online classifieds marketplace app in China) resolved the problem by canceling FpAuth on rooted devices through risk control measures instead of appropriately using reliable cryptographic key validation. This reflects the challenges developers face in addressing the complexities associated with fixing M2. Furthermore, this defense through risk control may not be reliable due to various means of counteracting root detection.

We reported the *M4. Mishandled Updates* issue to Meta Security's bounty program. They emphasized that fingerprint removal requires prior authentication. However, this prior authentication relies on a weak lock screen PIN, vulnerable to common attacks like shoulder surfing and guessing. Enhancing FpAuth security through proper implementation, as indicated in ISO/IEC standard[31], is crucial. Notably, Element, a similar messenger app, employs a robust implementation, requiring a separate password for unlocking after removing all fingerprints, providing resistance against physical intimate attackers.

VII. Discussion

Mitigation. To address the prevalent FpAuth misuses, we provide several best practices that help mitigate the four types of misuses. For *M1. Obsolete API*, developers should migrate to the newer FpAPIs, i.e., `BiometricPrompt` APIs or adopt `androidx.biometric` library, which ensures compatibility with older devices, to prevent the fingerprint-jacking attack. For *M2. Weak Crypto*, developers for remote services should ensure the proper use, configuration, and validation of the cryptographic key, setting the correspondent parameter of `authenticate` method to a cryptographic key, configuring `setUserAuthenticationRequired` to `True`, and employing it in `onAuthenticationSucceeded` callback to validate with the server. For *M3. Unauthorized Deactivation*, developers should verify the operator's identity when deactivating FpAuth, which is easily overlooked. For *M4. Mishandled Updates*, if developers prefer to use FpAuth without a PIN, which is a more secure way, they should carefully configure the cryptographic key, setting the `setInvalidatedByBiometricEnrollment` to `True`, in line with the best practices for M2. Additionally, Android can provide more customized and accurate guides for apps with different intentions and more unified APIs for safer choices.

Ethics. Our study focuses on the security of FpAuth and thus it is inevitably necessary to test and simulate attacks to attempt to bypass FpAuth. To uphold ethical standards, all aspects of our study and testing have been exclusively conducted using our own devices and our test accounts, avoiding any potential harm to other users' account security.

Other Biometrics. In this paper, we focus on analyzing the security issues within the FpAuth lifecycle, given its prevalence and widespread adoption in authentication systems. However, similar principles and methodologies can be applied to other biometric authentication schemes, such as facial recognition and iris scanning. These biometric technologies may also face similar security issues, including unauthorized access, spoofing attacks, and data breaches. Therefore, the insights gained from this study can be applied to enhance

the security and resilience of other biometric authentication systems. Future research efforts could explore the specific misuse patterns and security risks associated with different biometric authentication, further advancing the understanding and mitigation of biometric authentication vulnerabilities.

Limitation. The limitations of this study are primarily due to the inherent challenges of static analysis tools and the complexity of dynamically executing FpAuth. Our identification of FpAuth apps and detection of FpAPI misuse rely on the ability of static analysis tools to construct accurate and complete control and data flows. However, static analysis can be hindered by techniques such as native calls and code packing commonly employed in Android apps. We have mitigated some of these challenges using state-of-the-art tools for unpacking and by enhancing inter-component communication (ICC) analysis, which has partially addressed these limitations.

Moreover, to validate the results, dynamic execution of the FpAuth functionality is essential. This necessitates manual efforts to complete the login process within apps and fingerprint enrollment on the device. Thus, large-scale testing is limited by the time and manual resources required. These common limitations are orthogonal to the primary focus of our work, i.e., studying the real-world misuses of FpAPIs. Nevertheless, we are optimistic that future advancements in these areas will enhance the findings of this study and similar research.

VIII. Related Work

Fingerprint APIs. Recent research has focused on the security analysis of FpAPIs due to the potential risks associated with FpAuth in critical user operations. To the best of our knowledge, Zhang et al. [7] are the first to demonstrate the security issues in FpAuth, where they present four attack techniques targeting mobile FpAuth frameworks. Among these attacks, GUI hijacking, as explored by Bianchi et al. [32], involves a malicious app replacing the GUI of other apps to mount click-jacking attacks. As Android APIs have evolved over the years, a more recent study Wang et al. [9] proposes five novel attacks about fingerprint hijacking, which can even bypass the latest countermeasures in Android 9+ and be effective against all apps using the APIs. These investigations underscore FpAPI security vulnerabilities, emphasizing the need for robust countermeasures in FpAuth.

Furthermore, observing the misuse of these FpAPIs by app developers, Bianchi et al. [10] present a systematic analysis of the FpAPIs in Android, which is the previous work most relevant to our study. Their research primarily focuses on the cryptographic checks of FpAPIs during the verification stage. In contrast, our work offers a novel research perspective by examining the entire lifecycle of FpAuth, including activation, verification, updates, and deactivation. This comprehensive approach allows us to identify and analyze security issues across different stages, rather than focusing solely on verification. Our study introduces new threat models and identifies new misuse types, providing a broader understanding of FpAuth security. By analyzing the FpAuth lifecycle, we uncover interesting and practical findings that highlight real-world security implications and potential improvements.

Fingerprint Recognition. The widespread use of fingerprint recognition has heightened concerns over its security. Feng et al. [33] investigate fingerprint obfuscation techniques that enable individuals to alter their fingerprints without detection. Roy et al. [3] examine the advantages and disadvantages of employing small-sized fingerprint sensors for authentication and the feasibility of matching synthesized partial fingerprints with stored templates. Additionally, relevant research suggests that the cost associated with forging fingerprints may be relatively low, and even cheap nylon materials can be used to counterfeit fingerprints [7]. Chen et al. [6] propose a zero-knowledge brute-force attack to unlock Android phones. These findings collectively underscore the need for improved security in fingerprint recognition technology.

More on FpAuth Security. Many prior works have contributed greatly to the development of FpAuth security. Dharavath et al. [34] conduct a comprehensive review of challenges and future possibilities in a systematic manner. Ratha et al. [35] analyze the strengths of FpAuth, and propose a generic fingerprint system framework to identify vulnerable points. Ballard et al. [1] discuss the trade-off between fingerprint data mismatch and privacy concerns at a theoretical level. Tiefenau et al. [36] point out that systems supporting multiple authentication modes are only as secure as the weakest one. Wimberly et al. [37] highlight the impact of perceptible security measures on users' perception of system security. These insights contribute to the expansion of understanding of FpAuth, opening avenues for further in-depth exploration.

As a key part of FpAuth, TEE securely stores fingerprint data. Some studies on TEE have also emerged in recent years. Shakevsky et al. [38] identify vulnerabilities in Samsung's TrustZone cryptographic process, leading to attacks on TEE, affecting FpAuth and FIDO protocol. Imran et al. [39] introduce SARA, a library simplifying TEE-specific API use to facilitate TEE integration and utilization.

IX. Conclusion

This work systematically analyzes FpAPI misuses in real-world Android apps from the perspective of the FpAuth lifecycle. We identify real-world FpAuth apps and perform measurements to understand their characteristics. Through principled analysis, we identify the threat models and summarize four common FpAPI misuse types. Then, we develop a static analysis tool to detect these misuses and unveil that FpAPI misuses widely exist in real-world apps, leading to far-reaching consequences like account takeover, privacy violations, and financial losses. Our responsible disclosures have been acknowledged by app vendors, resulting in 203 CVE/CNVD vulnerability IDs. The detailed analysis of each type of misuses also comes up with several alarming findings that call for more attention to the secure usage of these APIs from both developers and API designers. We have publicly released our code and data at https://github.com/FpAuth/FpAuthAnalysis to facilitate reproduction and subsequent research.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (62102091, 62172104, 62172105, 62472096, 62102093, 62302101, 62402114, 62402116, 62202106), National Key Research and Development Program (2021YFB3101200). Zhemin Yang was supported in part by the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

REFERENCES

[1] L. Ballard, F. Monrose, and D. P. Lopresti, "Biometric authentication revisited: Understanding the impact of wolves in sheep's clothing." in *USENIX Security Symposium*, 2006.

[2] A. S. Rathore, Y. Shen, C. Xu, J. Snyderman, J. Han, F. Zhang, Z. Li, F. Lin, W. Xu, and K. Ren, "Fakeguard: Exploring haptic response to mitigate the vulnerability in commercial fingerprint anti-spoofing," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-230/

[3] A. Roy, N. Memon, and A. Ross, "Masterprint: Exploring the vulnerability of partial fingerprint-based authentication systems," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 9, pp. 2013–2025, 2017.

[4] P. Bontrager, A. Roy, J. Togelius, N. Memon, and A. Ross, "Deepmasterprints: Generating masterprints for dictionary attacks via latent variable evolution," in *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*. IEEE, 2018, pp. 1–9.

[5] A. Abhyankar and S. Schuckers, "Integrating a wavelet based perspiration liveness check with fingerprint recognition," *Pattern Recognition*, vol. 42, no. 3, pp. 452–464, 2009.

[6] Y. Chen, Y. Yu, and L. Zhai, "InfinityGauntlet: Expose smartphone fingerprint authentication to brute-force attack," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2027–2041.

[7] Y. Zhang, Z. Chen, H. Xue, and T. Wei, "Fingerprints on mobile devices: Abusing and leaking," in *Black Hat Conference*, 2015.

[8] X. Wang, Y. Chen, R. Yang, S. Shi, and W. C. Lau, "Fingerprint-jacking: Practical fingerprint authorization hijacking in android apps," *Blackhat, Europe, Tech. Rep. Blackhat*, vol. 2020, 2020.

[9] X. Wang, S. Shi, Y. Chen, and W. C. Lau, "Phyjacking: Physical input hijacking for zero-permission authorization attacks on android." in *NDSS*, 2022.

[10] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S. P. H. Chung, and W. Lee, "Broken fingers: On the usage of the fingerprint api in android." in *NDSS*, 2018.

[11] Google, "Android biometric authentication guide," https://developer.android.com/training/sign-in/biometric-auth, 2023.

[12] "Strongbox," https://developer.android.com/privacy-and-security/keystore#HardwareSecurityModule, 2021.

[13] F. Alliance, "Fido - fido alliance," https://fidoalliance.org/fido2/, 2023.

[14] hluwa, "Frida-dexdump - a frida tool to dump dex in memory to support security engineers analyzing malware." https://github.com/hluwa/frida-dexdump, 2022.

[15] iBotPeaches, "Apktool - a tool for reverse engineering android apk files." https://github.com/iBotPeaches/Apktool, 2020.

[16] Soot, "A java optimization framework," https://github.com/soot-oss/soot, 2023.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[18] A. Sudhodanan and A. Paverd, "Pre-hijacked accounts: An empirical study of security failures in user account creation on the web," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1795–1812.

[19] S. Ma, R. Feng, J. Li, Y. Liu, S. Nepal, Diethelm, E. Bertino, R. H. Deng, Z. Ma, and S. Jha, "An empirical study of sms one-time password authentication in android apps," in *Proceedings of the 35th annual computer security applications conference*, 2019, pp. 339–354.

[20] Y. Chen, R. Tang, C. Zuo, X. Zhang, L. Xue, X. Luo, and Q. Zhao, "Attention! your copied data is under monitoring: A systematic study of clipboard usage in android apps," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 742–754.

[21] Google, "Android apps on google play," https://play.google.com/store, 2022.

[22] Huawei, "Top android apps on huawei market," https://appgallery.huawei.com/Top, 2022.

[23] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 627–638.

[24] X. Zhang, H. Ye, Z. Huang, X. Ye, Y. Cao, Y. Zhang, and M. Yang, "Understanding the (in) security of cross-side face verification systems in mobile apps: A system perspective," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 934–

950.
[25] M. Eiband, M. Khamis, E. Von Zezschwitz, H. Hussmann, and F. Alt, “Understanding shoulder surfing in the wild: Stories from users and observers,” in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 4254–4265.
[26] P. Markert, D. V. Bailey, M. Golla, M. Dürmuth, and A. J. Aviv, “This pin can be easily guessed: Analyzing the security of smartphone unlock pins,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 286–303.
[27] F-Droid, “Free and open source android app repository.” https://f-droid.org/, 2023.
[28] Google, “Android biometric authentication guide - allow for fallback to non-biometric credentials,” https://developer.android.com/training/sign-in/biometric-auth#allow-fallback, 2023.
[29] ——, “Android fragments,” https://developer.android.com/guide/fragments, 2023.
[30] Frida, “Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.” https://frida.re/, 2023.
[31] “Information security, cybersecurity and privacy protection — Security and privacy requirements for authentication using biometrics on mobile devices — Part 1: Local modes,” International Organization for Standardization, Geneva, CH, Standard, Nov. 2022.
[32] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, “What the app is that? deception and countermeasures in the android user interface,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 931–948.
[33] J. Feng, A. K. Jain, and A. Ross, “Fingerprint alteration,” *submitted to IEEE TIFS*, 2009.
[34] K. Dharavath, F. A. Talukdar, and R. H. Laskar, “Study on biometric authentication systems, challenges and future trends: A review,” in *2013 IEEE international conference on computational intelligence and computing research*. IEEE, 2013, pp. 1–7.
[35] N. K. Ratha, J. H. Connell, and R. M. Bolle, “Enhancing security and privacy in biometrics-based authentication systems,” *IBM systems Journal*, vol. 40, no. 3, pp. 614–634, 2001.
[36] C. Tiefenau, M. Häring, M. Khamis, and E. von Zezschwitz, “” please enter your pin”–on the risk of bypass attacks on biometric authentication on mobile devices,” *arXiv preprint arXiv:1911.07692*, 2019.
[37] H. Wimberly and L. M. Liebrock, “Using fingerprint authentication to reduce system security: An empirical study,” in *2011 ieee symposium on security and privacy*. IEEE, 2011, pp. 32–46.
[38] A. Shakevsky, E. Ronen, and A. Wool, “Trust dies in darkness: Shedding light on samsung’s {TrustZone} keymaster design,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 251–268.
[39] A. Imran, H. Farrukh, M. Ibrahim, Z. B. Celik, and A. Bianchi, “{SARA}: Secure android remote authorization,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1561–1578.

APPENDIX A
DATASET FOR PILOT STUDY

To investigate the security issues present in the real-world FpAuth apps, we conduct an in-depth manual analysis on 40 representative apps. The details of the apps are shown in Table XIII.

Table XIII
The dataset for our pilot study. Note that the origin refers to FpAuth implementations origins used in our dataset, including SDKs, open-source repositories, etc.

No.	Package Name	Category	Source	Detail
1	org.telegram.messenger	Social	Origin	org.telegram.messenger
2	com.smsrobot.callu	Productivity	Origin	com.github.ajalt
3	com.adventhealth.flagship	Health & Sports	Origin	com.epic.patientengagement
4	edu.aiuniv	Education	Origin	de.niklasmerz.cordova
5	net.baptisthealth.android.bhsf.careondemand	Health & Sports	Origin	com.americanwell.android
6	com.divessi.ssi	Health & Sports	Origin	io.flutter.plugins
7	com.coca_cola.android.cep	Food & Drink	Origin	com.salesforce.androidsdk
8	com.yahoo.mobile.client.android.fantasyfootball	Health & Sports	Origin	com.oath.mobile
9	com.usablenet.mobile.walgreen	Shopping	Origin	com.visa.checkout
10	com.bgmobilenga	Lifestyle	Origin	com.oblador.keychain
11	com.baidu.BaiduMap	Maps & Navigation	Origin	com.baidu.sapi2/wallet
12	com.wangc.bill	Finance	Origin	com.wei.android
13	com.taobao.trip	Travel & Local	Origin	com.alipay.security
14	com.tynet.huiliao.doc	Health & Sports	Origin	cn.org.bjca
15	com.tuya.smartiot	Productivity	Origin	com.tuya.security/smart
16	com.wuba	Lifestyle	Origin	com.tencent.soter
17	com.systec.umeeting	Business	Origin	us.zoom.androidlib
18	com.ctrip.jd	Finance	Origin	ctrip.android.pay
19	com.sankuai.meituan.takeoutnew	Food & Drink	Origin	com.meituan.android
20	cn.yingmi.qieman.hermione	Finance	Origin	com.rnfingerprint.FingerprintHandle
21	com.abhinavmarwaha.wrotto	Productivity	F-Droid	-
22	com.zell_mbc.medilog	Health & Sports	F-Droid	-
23	sushi.hardcore.droidfs	Productivity	F-Droid	-
24	org.koitharu.kotatsu	Entertainment	F-Droid	-
25	de.jepfa.yapm	Productivity	F-Droid	-
26	com.glitterware.passy	Productivity	F-Droid	-
27	com.velas.mobile_wallet	Finance	F-Droid	-
28	com.standardnotes	Productivity	F-Droid	-
29	com.soumikshah.investmenttracker	Finance	F-Droid	-
30	com.kunzisoft.keepass.libre	Productivity	F-Droid	-
31	com.beemdevelopment.aegis	Productivity	F-Droid	-
32	com.simplemobiletools.gallery.pro	Productivity	F-Droid	-
33	com.nextcloud.client	Productivity	F-Droid	-
34	com.simplemobiletools.filemanager.pro	Productivity	F-Droid	-
35	com.huawei.holosens	Entertainment	Market	Huawei
36	com.tencent.gallerymanager	Photography & Editors	Market	Huawei
37	com.cns.qiaob	News & Books	Market	Huawei
38	com.sharkdevelop.platincoins	Libraries & Demo	Market	Google Play
39	com.whatsapp	Social	Market	Google Play
40	com.fca.myconnect.alfaromeo.nafta	Auto & Vehicles	Market	Google Play