Three-Body Problem: An Empirical Study on Smartphone-based TEEs, TEE-based Apps, and their Interactions

Xianghui Dong
College of Computer Science
Beijing University of Technology
Beijing, China
dongxianghui6@gmail.com

Yin Liu
College of Computer Science
Beijing University of Technology
Beijing, China
yinliu@bjut.edu.cn

Xuejun Yu*
College of Computer Science
Beijing University of Technology
Beijing, China
yuxuejun@bjut.edu.cn

***Abstract*—Trusted Execution Environments (TEE) serve as a fundamental trust infrastructure of smartphones. By placing critical code and data inside TEE, smartphone apps ensure their confidentiality and integrity, even with a compromised outside system. However, there is a lack of studies on the usage of TEEs on smartphones. To that end, we conduct a comprehensive empirical study, demystifying smartphone-based TEEs, TEE-based apps, and their interactions. Specifically, our study answers threefold questions: (1) how are TEEs designed for smartphones, (2) what do the TEE-based apps look like, and (3) how do these apps use TEE? To answer these questions, our research investigates 17 TEE systems and more than 10,000 real-world Android apps across over 50 regular and malicious app categories. To automate the investigation, a tool that detects if an app uses TEE (i.e., TEE-based apps) has been provided. Our findings can provide practical guidance for app testers to generate valid TEE test cases, for TEE vendors to optimize their TEE solutions, and for security researchers and developers to enhance their protection mechanisms.**

***Index Terms*—trusted execution, mobile security, mobile apps**

I. Introduction

With the proliferation of mobile device usage, trust has become a pressing concern. The Trusted Execution Environment (TEE) emerges as a trust infrastructure, providing an isolated environment that helps to ensure confidentiality and integrity for both code and data. To seize this opportunity to offer more reliable mobile services, vendors have begun implementing their own TEE solutions and integrating them into smartphones. Nowadays, various smartphone-based TEEs have been rolling out, such as Google's Trusty [1], Samsung's TEEGRIS [2], Huawei's iTrustee [3], and more. Following this trend, a wave of "TEE-based" apps (i.e., the apps using TEE) have emerged, leveraging TEEs to perform trust operations (e.g., face/fingerprint authentication) and prevent external interference with their code and data. Hence, it is crucial to have a clear understanding of TEE-based apps, the TEE they are based on, and their utilization of TEEs on smartphones.

Currently, three primary categories of studies exist: the first group of studies surveys the characteristics of TEEs running not on smartphones [4], as well as how smartphone apps use deep learning models [5], blockchain [6], and third-party libraries [7]. However, these studies neither explore the features of smartphone-based TEEs nor how smartphone apps utilize TEEs. The second group analyzes the architecture and features of smartphone-based TEEs. However, it only focuses on a single TEE solution [8], five different TEE solutions [9], or a single TEE application scenario [10]. Since more than 15 various TEEs exist, these studies have not comprehensively represented smartphone-based TEEs. The third group examines TEE-based apps and their TEE usage scenarios. However, it lacks an analysis of commercial TEEs and real-world smartphone apps [11]. It also overlooks important categories of apps, such as malware and games, and omits essential TEE application scenarios like Gatekeeper, which is responsible for device pattern/password authentication [12], [13]. Furthermore, their API-based TEE operation detector utilizes an insufficient number of TEE APIs (about 200 compared to the approximately 600 APIs in our study) and relies on a simple filename-matching method instead of reverse engineering the native libraries (.so files), which reduces the accuracy of their analysis [12], [13].

To bridge the knowledge gap, this paper presents an empirical study that explores a wide range of commercial TEE solutions, a large number of both regular and malicious apps across various categories, and essential TEE application scenarios. The aim is to draw a comprehensive picture of smartphone-based TEEs, TEE-based apps, and their interactions.

The study will answer the following questions:

RQ1 - How are TEEs designed for smartphones: Is there a common architecture of smartphone-based TEEs? Is there a typical communication channel between apps and TEE operations? Do different TEEs share the same API standards?

RQ2 - What do the TEE-based apps, including both regular and malicious ones, look like: What percentage of apps use or do not use TEE? What categories of apps make the most or least use of TEEs? Does the TEE usage relate to the app's rating scores or installs?

RQ3 - How do regular and malicious apps use TEE:

* Corresponding author.

What are the most common TEE operations in all detected apps and specific categories? What are typical application scenarios of TEE-based apps? Are there similar application scenarios for malicious and regular apps when using TEE?

To answer RQ1, we collected technical documents or source code from 17 smartphone-based TEEs developed by various vendors. We then manually analyzed and extracted their design, architectures, and APIs. To address RQ2 and RQ3, we followed three steps. First, we obtained installation packages of over 10,000 Android apps, belonging to 33 legitimate app categories and 20 malicious categories, respectively. Second, we developed a new tool that automatically examines the obtained packages to identify apps that use TEE. Finally, we gathered metadata for the identified apps and analyzed their characteristics and TEE usage.

We have summarized our key findings as follows: **For RQ1**, (1) The primary mobile system that smartphone-based TEEs target is Android; (2) Global Platform (GP)'s standard APIs for TEEs have been widely adopted, but few TEEs have obtained GP's certification; (3) The technical details of many smartphone-based TEEs remain a mystery.

For RQ2, (1) Overall, TEE is more commonly used in regular apps than malicious ones. However, some types of malicious apps may utilize TEE at a high rate; (2) The use of TEE is positively correlated with the popularity of the app (i.e., rating scores and installs); (3) Both regular and malicious apps apply the `Indirect Channel` to communicate with the secure world instead of the `Direct Channel` (See § IV-D).

For RQ3, (1) The most popular scenario for TEE is the "cryptographic operations"; (2) When it comes to authentication scenarios, "biometrics-based authentication" is more prevalent than the "pattern/password-based authentication"; (3) In TEE application scenarios, there is a clear distinction between different categories of apps, but the difference between regular and malicious apps is not noticeable.

The contribution of this paper is as follows:

(1) A comprehensive empirical study on smartphone-based TEEs, TEE-based apps, and their interactions: We have studied 17 smartphone-based TEEs and analyzed over 50 app categories, covering both regular and malicious apps, totaling more than 10,000 real-world Android apps. Our study summarized a common architecture of smartphone-based TEEs and identified two typical communication channels for TEE-based apps. Additionally, we presented a series of new findings of TEE usage and provided a set of recommendations for TEE vendors, app testers, and security researchers and developers.

(2) A new TEE operations detector on both managed and native levels: We have developed a tool that can automatically identify TEE-based apps in both managed (i.e., code written in managed languages like Java or Kotlin) and native levels (i.e., code written in native languages like C/C++). Given the installation package of an Android app (.apk), the tool first reverse engineers it to the managed and native levels, and then determines if the app performs TEE operations by matching app invocations with APIs among more than 600 TEE direct or indirect APIs.

(3) Three TEE-related datasets: We have created three datasets: one for installation packages and metadata of TEE-based apps, one for official documents of smartphone-based TEEs, and one for specifications of TEE APIs. We release the prototype of our detection tool along with the datasets at: https:/github.com/WesternHunter/Android-App-Analysis

II. Background

(1) TEE solutions: on-smartphone vs. others. TEE [14] is a hardware-software collaborated infrastructure to ensure the secure processing of code and data. To enhance their products' security, many IT companies design and implement their own TEEs. These emerged TEEs are designed for different scenarios: Intel's Software Guard Extensions (SGX) [15] has already been integrated into multiple product lines of their processors, so that running on an enormous number of desktops and laptops [16]. Similarly, ARM's TrustZone, which incorporates ARM chips, is widely utilized in IoT and embedded systems. As smartphones become increasingly important, mobile vendors start designing and customizing TEEs for their devices, such as Google's Trusty [1], Huawei's iTrustee [3], Samsung's TEEGRIS [2], Apple's Secure Enclave [17], Qualcomm's QTEE [18], and more. These TEEs collaborate with specific mobile platforms for securing apps' code and data. In contrast to desktops, laptops, IoTs, and other embedded devices, smartphones become the most common devices used in our everyday lives, having a vast number and variety of apps and computing on limited resources. Hence, it is crucial to comprehend the design and usage of TEEs on smartphones (we call them *smartphone-based TEEs*).

(2) TEE APIs & TEE-based Apps. Global Platform (GP) [19], a cross-industry international standardization organization, has established a standard specification of TEE APIs. However, using these standard APIs is not mandatory, so vendors can either comply with the GP's APIs or provide alternatives on their own. Hence, both of these situations should be considered when investigating the usage of TEEs.

Since these designated APIs bridge the TEE's normal and secure worlds, a smartphone app can communicate with TEEs to perform secure operations via these APIs. In this paper, we refer to an app that uses TEE as a "TEE-based" app. Besides, due to Android's dominant market share in smartphones, we are focusing on Android apps. Android provides SDK and NDK to control an app's lifecycle and allow native code implementation with languages such as C and C++. Therefore, we hypothesize that an Android app can communicate with TEEs using two methods. The first method is indirect invocation, where an app interacts with the TEE through Android framework APIs (i.e., SDK). The second method is direct invocation, where an app directly calls TEE APIs through native layer functions (i.e., NDK). We will confirm our hypothesis by analyzing smartphone-based TEEs and TEE-based apps (discussed in § IV-D).

III. Methodology

In this section, we first introduce our research goal and scope, and then we present our solution overview.

A. Goal and Scope

(1) Research Goal: This paper aims to explore the smartphone-based TEEs, TEE-based apps, and their interactions. It thus covers two study subjects: (1) *smartphone-based TEEs*, including their standard architecture, communication channels, and APIs (discussed in § IV), and (2) *TEE-based apps*, including their characteristics and application scenarios (discussed in § V).

(2) Scope: For *smartphone-based TEEs*, we explore existing commercial TEE solutions on different smartphone systems, such as Android, Harmony, and iOS, to draw a comprehensive picture of how TEEs are designed and worked. We opted for commercial TEEs instead of academic offerings to showcase mature solutions and systems. We believe that the analysis results generated by these commercial TEEs hold greater value and significance for testers, TEE vendors, and security researchers and developers.

For *TEE-based apps*, we focus on *Android apps*, more specifically, *TEE-based Android apps*. We choose Android for three reasons. (a) Android has the highest market share among smartphones: according to StatCounter, Android held about 70% of the global mobile market as of the end of March 2024 [20]; (b) Android has an enormous number of apps: according to 42matters, there are over 3.2 million Android apps available on Google Play Store in 2024 [21]; (c) Android has been commonly used by many smartphone vendors: due to Android being an open-source system, many vendors customize the original Android system to create their own system. Examples of such vendors include Samsung, Sony, and Xiaomi. Although we focus on Android apps, our analysis approach and results can provide insights for apps running on other mobile systems.

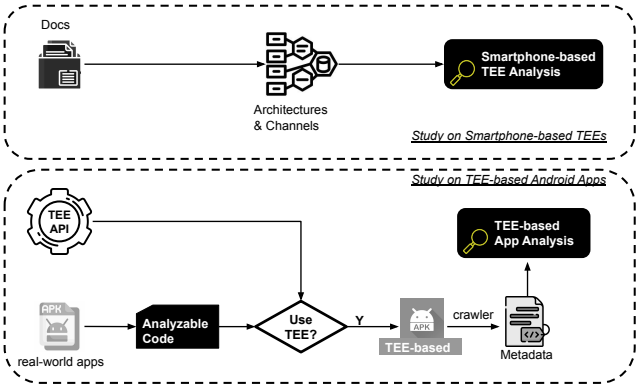

Fig. 1. The workflow of our study

B. Solution Overview

Figure 1 presents the workflow of our solution, which include two parts:

(1) Our study on *smartphone-based TEEs* (shown at the top of the Figure 1) starts by gathering official documents from various TEE vendors. After analyzing these documents, we identify the TEEs that can be integrated into smartphones (i.e., smartphone-based TEEs) and extract a set of architectures and communication channels from each solution. Next, we summarize these different architectures and channels to propose the common and typical ones.

(2) Our study on *TEE-based Android apps* (shown at the bottom of the Figure 1) starts by gathering real-world Android apps for their installation packages (i.e., APKs). These APKs are transformed into analyzable code by using reverse engineering techniques. Besides, we build a dataset of TEE APIs from various data sources, including the above-mentioned documents, Android's source code, and the Android Developers website. Based on this dataset, we check if an analyzable code invokes TEE operations via corresponding TEE APIs. If it does, we add the identified apps to the TEE-based app dataset while crawling and extracting its metadata, such as rating scores and installs. Finally, we perform statistical analysis on the metadata and TEE API's call information to answer aforementioned research questions.

IV. Smartphone-based TEEs

This section illustrates our empirical study on smartphone-based TEEs.

A. Methodology and Assumptions

(1) Methodology. Our empirical study on smartphone-based TEEs including two phases, collection and analysis.

a) Collection: To start our process, we begin by gathering a list of mobile vendors online. Once we have the list, we visit the vendors' web pages and search for documents and source code (if open-sourced) related to topics such as "Trusted Execution Environment," "Secure," and "Privacy." This allows us to obtain any TEE-related materials and open-sourced code available. In the end, we collect official documents such as white papers, technical reports, and developer guidance from 23 vendors. These documents present the vendors' TEE solutions and products.

b) Analysis: We filtered out 17 smartphone-based TEEs from the document collection by eliminating TEEs that cannot be deployed on smartphones. Upon manually examining each document, corresponding source code (if applicable), and GlobalPlatform's website, we obtain critical information for each TEE. This information includes the targeted mobile platform, supported TEE APIs, obtained security certifications, system architectures, and communication channels. After analyzing the information, we uncover a common architecture and two typical communication channels between mobile apps and the secure worlds of TEEs.

(2) Assumptions. We assume that all the documents collected from mobile vendors accurately describe their proprietary TEE solutions. That is, assuming a document explicitly states that their TEE solution targets Android and complies with GlobalPlatform's TEE API, we trust that this information is accurate and dependable. In contrast, for security certification provided by GlobalPlatform for TEE, we track the latest results on their website [22] by the end of September 2024. To put it simply, if a TEE has not been confirmed on GP's website, we assume that it does not possess GP's certification and vice versa. Additionally, we mark a TEE *"Maybe"* receive

certification if its certification is not listed on GP's website, but can be found on other websites.

TABLE I
SUMMARY OF SMARTPHONE-BASED TEES

Solution	Vendor	Targeted Sys.	API	Cert.	Details
iTrustee [3]	Huawei	HarmoryOS [23]	GP	Y	Y
Kinibi [24]	Trustonic	Android	GP & Proprietary	Maybe	Y
QTEE [18]	Qualcomm	Android	GP & Proprietary	N	Y
Trusty [1]	Google	Android	Proprietary	N	Y
Secure Enclave [17]	Apple	iOS	Proprietary	N	Y
OP-TEE [25]	Linaro	Android	GP	N	Y
TEEgris [2]	Samsung	Android	GP & Proprietary	Y	Y
TEEI [26]	Eastcompace	Unknown	Unknown	N	N
ISEE [27]	BeanPod	Android	GP	N	N
WatchTrust [28]	Watchdata	Android	GP	Maybe	N
TEE [29]	Rocky Core Tech.	Android	GP	N	N
T6 [30]	TrustKernel	Android	GP	N	N
Link-TEE [31]	Alibaba	Android	GP	N	N
MiTEE [32]	Xiaomi	Xiaomi HyperOS[1]	GP	N	Y
CoreTEE [33]	Sequitur Labs	Linux-based OS	GP	N	Y
ProvenCore [34]	ProvenRun	Unknown	GP	N	N
SierraTEE [35]	Sierraware	Unknown	GP	N	N

B. Current Situations

Table I shows our studied TEEs, which covers 17 dissimilar smartphone-based TEE solutions (shown in the column "Solution") from different vendors (shown in the column "Vendor"). The column "Targeted Sys." shows the targeted mobile system, and the column "API" shows the supported TEE API, both collected from their official documents and source code. The column "Cert." displays the GP's certificate of security evaluation, which we mark "Y" if the certificate can be found on the GP's website and vice versa. When a certificate can be found online but not listed on GP's website, we label it "Maybe." The "Details" column indicates whether enough technical information about a given TEE is available in their documentation or online.

Based on this information, we summarize the current situations of smartphone-based TEE as follows:

<Finding-1> the primary platform that smartphone-based TEEs target is Android: 71.4% of the analyzed TEEs claimed target Android platform (TEE with "Unknown" targeted systems were excluded). Moreover, Xiaomi HyperOS, targeted by MiTEE, is based on Android. Also, Android is a Linux-based OS that can deploy CoreTEE. Hence, if we consider these two targeted systems to be Android, only 2 out of 14 TEEs (excluding "Unknown" targeted systems) target smartphone systems other than Android.

<Finding-2> GP's standard APIs for TEEs have been widely adopted, but few TEEs have obtained GP's certification: 87.5% of the analyzed smartphone-based TEEs either claimed to support standard GP APIs or to use these APIs with proprietary APIs (TEE with "Unknown" API were excluded). According to GP's website, only 2 out of 16 TEEs (excluding "Unknown" API) have obtained GP's certificate for TEE security.

<Finding-3> the technical details of many smartphone-based TEEs still remain a mystery: Only 52.9% of the analyzed TEEs provided details on their website or other materials (e.g., whitepapers, developer guidances, and source code). In other words, almost half of the analyzed TEEs (marked by "N" in the column "Details") only provided a brief overview of their targeted systems, supported APIs, and TEE architectures/communication channels, without delving into the specifics of how to implement or apply their own proprietary TEE solutions.

[1]An Android-based mobile OS designed and implemented by Xiaomi.

C. Common Architecture

As shown in Figure 2, we find that a smartphone-based TEE is commonly structured around following layers:

(1) Application Layer: The application layer is the topmost layer of the system and allows for the operation of smartphone apps. In the normal world (on the left-hand of Figure 2), the `Client Apps (CAs)` represents the apps aiming to communicate with the secure world, and the `Normal Apps (NAs)` represents the regular smartphone apps (e.g., apps downloaded from app stores online), which may or may not access the secure world. In the secure world (on the right-hand of Figure 2), the `Trusted Apps (TAs)` represents the apps providing secure functionality to CAs or other TAs. Technically, the TAs can be provided by third parties; they also can be pre-installed by vendors (i.e., the `Embedded TAs`.)

(2) Application Framework Layer: This layer provides a set of APIs (usually written in managed languages), simplifying the use of lower layers' components and services. Typically, these APIs can be implemented as a wrapper of necessary TEE-related libraries in the Core Services layer. In particular, the `FW APIs of CAs` is invoked by CAs in the normal world, and the `FW APIs of TAs` by TAs in the secure world. It is worth noticing that providing these APIs in the Application Framework layer is an option, because CAs and TAs can also be implemented to communicate directly with their corresponding lower-layer libraries.

(3) Core Services Layer: In the normal world, `CA Libs`, written mostly in C/C++, aim to establish a communication channel between CAs and TAs. `CA Libs` do this by exposing necessary communication interfaces to upper layers and forwarding message to the Core OS layer. Typically, these interfaces can be exposed to the Application Framework or Application layers. The interfaces exposed to the Application Framework can be implemented through the aforementioned `FW APIs of CAs`; those to the Application layer can be implemented through native interfaces (e.g., JNI and NDK) directly interacting with CAs. In practice, `CA Libs` can contain the GlobalPlatform TEE Client APIs or (and) the proprietary APIs offered by particular vendors.

In the secure world, `TA Libs` can also enable the connection between CAs and TAs, which is similar to `CA Libs`. More importantly, `TA Libs` provide interfaces of secure functionalities to TAs. To enable the communication between CAs and TAs, `TA Libs` can expose interfaces to Core OS for message relaying and processing. To enable the secure functionalities for TAs, `TA Libs` can expose interfaces either to the Application Framework layer through the `FW APIs of TAs` or to the Application layer directly. In practice, `TA Libs` can contain the GlobalPlatform TEE Internal APIs or (and) the proprietary APIs offered by particular vendors.

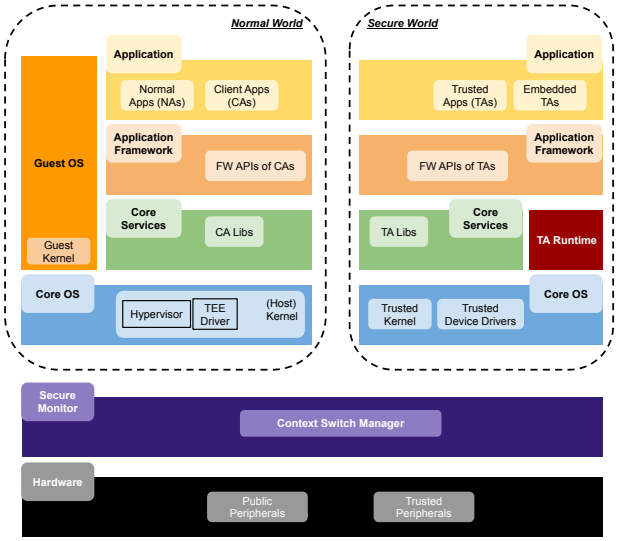

Fig. 2. The common architecture of smartphone-based TEEs

(4) Core OS Layer: In the normal world, this layer provides a `TEE Driver` that enables communication between normal and secure worlds. `TEE Driver` can interact with `CA Libs` in the upper layer and the Secure Monitor in the lower layer, and finally exchange messages with `Trusted Kernel` in the secure world. In addition to communicating with `TEE Driver`, `Trusted Kernel` provides scheduling and other operating system management functions (e.g., task isolation and inter-process communication) for both TAs and `TA Libs`. Furthermore, `Trusted Device Driver` provides communication between TAs and TEE's Trusted peripherals in hardware. In practice, mobile vendors always create their specific `TEE Driver`, `Trusted Kernel`, and `Trusted Device Driver` as the core of their TEE solutions.

(5) Secure Monitor Layer: This layer includes a `Context Switch Manager` that simplifies the transition between normal and secure worlds. To enable such a transition, a specific instruction is utilized known as Secure Monitor Call (SMC). Hence, the `Context Switch Manager` can provide handler services that the Core OS layer of both normal and secure worlds can request via the SMC.

(6) Hardware Layer: This layer provides both public and trusted peripherals. Noticing that the trusted peripherals, such as the trusted UI (touchscreen and keyboard), NFC controller, and secure storage, can only be accessed from the secure world through the `Trusted Device Drivers`.

(7) Other Layers: In addition to the previously mentioned layers, mobile vendors can offer additional layers to support their specific TEE solutions. To host multiple isolated guest operating systems, one can use a `Hypervisor` in the Core OS layer. The guest OS can also provide the aforementioned layers and communicate with the secure world. Besides, a `TA Runtime` can be integrated to support TAs execution inside the secure world. This runtime contains corresponding compilers or virtual machines for certain programming languages.

D. Typical Communication Channels

As shown in Figure 3, two typical communication channels enable an app to communicate with the secure world:

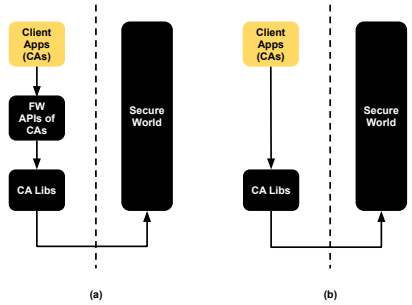

Fig. 3. Two channels between apps and smartphone-based TEE

(1) Indirect Channel: When a client application (CA) initiates a TEE-based operation, it sends a request with necessary data to APIs in the Application Framework layer (i.e., `FW APIs of CAs`). The `FW APIs of CAs` then invokes corresponding functions in the `CA Libs`. Next, the `CA Libs` routes the request to the `TEE Driver`, and then the `Secure Monitor`. The `Secure Monitor` sends the request to the `Trusted Kernel`. After receiving the request, the `Trusted Kernel` determines which TA should handle it and dispatches the control to the TA via the `TA Libs` and/or `FW APIs of TAs`. Once a TA receives the control, it starts executing corresponding TEE operations through `FW APIs of TAs`, `TA Libs`, and `Trusted Kernel`, and obtains return values and any processed data. After completion, the TA returns control to the original CA via the reverse path.

To simplify the above process, Figure 3-(a) illustrates the communication process in higher layers in the normal world. That is, `CAs` calls `FW APIs of CAs`, and `FW APIs of CAs` then calls TEE-related functions in the `CA Libs`. Hence, from the `CAs`'s perspective, it only needs to invoke `FW APIs of CAs` for performing TEE-base operations. In this case, the TEE-related APIs in `CA Libs` are indirectly invoked by `CAs`, we thus call this communication channel the "indirect channel." Note that the `FW APIs of CAs` is usually implemented with managed languages (e.g., Java), which would be the same language used to implement `CAs`.

(2) Direct Channel: As shown in Figure 3-(b), `CAs` can directly interact with the `CA Libs` without `FW APIs of CAs` (the rest of the process is the same as the "indirect channel"). Note that `CA Libs` is typically implemented using native languages, such as C or C++. Hence, `CAs`, especially those that come pre-installed, can integrate a portion of `CA Libs` as their dynamic or static libraries (.so or .a files). Then, `CAs` can invoke with these libraries via native interfaces like JNI/NDK. In this case, the TEE-related APIs within `CA Libs` are directly invoked by `CAs`. Therefore, we call this communication channel the "direct channel."

E. Threats to Validity

The internal validity is threatened by the contents of our collected documents. Some collected documents only provide an overview of their solutions rather than technical details. **The external validity** is threatened by the number and types of our analyzed TEEs. Although we analyzed and summarized a representative sample of smartphone-based TEEs, the collection may still not be large enough to draw definitive conclusions.

To mitigate these threats, we plan to study academic TEEs, compare them with commercial ones, and make our dataset available online for practitioners to contribute more details.

V. TEE-based Apps in the Wild

This section details our approach to detecting and analyzing TEE-based apps in the wild. Please keep in mind that our detection approach is specifically tailored towards Android apps, as stated in § III-A.

A. Methodology: detecting and analyzing TEE-based apps

As shown in Figure 4, our approach includes three phases: TEE-API extraction, TEE-based app detection, and TEE-based app analysis. We discuss each phase in detail below.

(1) Large-scale TEE-API Extraction. The objective of this phase is to establish a comprehensive dataset of TEE APIs. According to Findings 1 and 2 (in § IV-B), APIs provided by GlobalPlatform (GP) are the dominant TEE APIs, and Android is the most commonly targeted platform among the smartphone-based TEEs. Hence, we gather TEE APIs from four different sources, as shown at the top of Figure-4: (a) Global Platform (GP), (b) Trusty (i.e., the built-in TEE provided by Google), (c) the Android Developers website, and (d) the Android Open Source Project (AOSP).

For (a), we extract APIs that can be invoked by CAs from GP's TEE Client API document [36]. For (b), we collect dedicated APIs from Trusty API Reference [37]. For (c), we search for TEE documents on the Android Developers website [38] by using keywords such as "Trusted Execution Environment," "(TEE)," "Trusted Application," and "(TA)." Then, we extract TEE APIs from the searched TEE-related guidance and specifications. For (d), we use the above keywords to search the entire Android codebase [39] for relevant comments regarding TEE. Then, we extract TEE APIs from the matched comments and their corresponding code snippets.

Finally, we create a dataset for all the extracted TEE APIs above and categorize them into two types: *Direct APIs* and *Indirect APIs.* The former includes the APIs from GP and Trusty, which are placed in the `CA Libs` (as shown in Figure 2.) CAs can integrate with *Direct APIs* as their dynamic or static libraries and invoke them via native interfaces (e.g., JNI/NDK) (i.e., the "direct channel" discussed in § IV-D). The latter includes the APIs from the Android Developers website and AOSP, which are placed in the `FW APIs of CAs`. CAs can invoke these APIs via regular functions (i.e., the "indirect channel" discussed in § IV-D). We plan to release our TEE API dataset as open source, enabling researchers to use and contribute to it.

(2) TEE-based App Detection. To identify whether an Android app uses TEE, we develop a software tool as shown at the bottom of Figure 4. Given an Android app's installation package (.apk), our tool first decompiles it into two parts: the native libs (i.e., .a and .so files), and the smali code (i.e., .smali files). For this purpose, we utilize `apktool` to perform the decompilation process. We then create a `Lib analyzer` and a `smali analyzer` to check for TEE APIs in the native libs and smali code, respectively. If either analyzer detects the presence of TEE APIs, we consider the app to use TEE and add it to the dataset of "TEE-based apps."

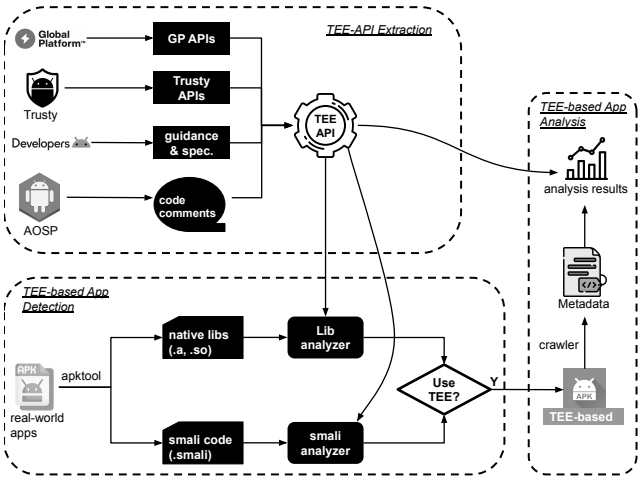

Fig. 4. Our approach to detecting and analyzing TEE-base apps in the wild

Specifically, we build the `Lib analyzer` on `IDA`, a widely used decompilation tool for C and C++. First, the `Lib analyzer` further decompiles the native libs to analyzable code (.i64 file) with `IDA`. Then, it triggers an `IDA` script to gather all the invocations' information (e.g., function name, parameters, return values) from the decompiled code. After that, the `Lib analyzer` examines the TEE API dataset, matching *Direct APIs* with the gathered invocations. If we find an API that matches, we think the app utilizes TEE. The reason why `Lib analyzer` only matches the *Direct APIs* is that, technically, the native libs can be integrated with CAs to provide *Direct APIs*.

We develop the `smali analyzer`. It first examines the smali code files, identifying invocation points and extracting all function calls' information (e.g., function name, parameters, return values). After that, the `smali analyzer` examines the TEE API dataset, matching *Indirect APIs* with the gathered function calls. If we find an API that matches, we think the app utilizes TEE. The reason why it only matches the *Indirect APIs* is that the smali code is placed in `FW APIs of CAs` to provide *Indirect APIs*.

It is worth noticing that we use a conservative approach of matching *Indirect APIs*. That is, we label an app as TEE-based if a given invocation in its smali code meets either of the following conditions. (a) the invocation is matched with a TEE API, based on function name, number of parameters, and class it belongs to, but not parameter type. (b) the invocation's parameter type includes the class to which a TEE API belongs. (c) some dedicated classes occur in the smali code where the invocation is located. These classes are TEE-related and can be used to indicate the TEE use.

(3) TEE-based App Analysis. Once we have acquired the TEE-based app dataset, we can proceed to the analysis phase. The dataset initially contains each TEE-based app's installation package (.apk), package name, and TEE APIs invoked by the app. Then, based on the package name, we create a crawler with `Google-Play-Scraper` [40] to collect metadata for

each app, such as rating scores and installs. On the other hand, we conduct a manual analysis of functional domains for *Indirect APIs*[2].

After studying on information collected from AOSP and Android Developer website, we discovered five domains of *Indirect APIs*: Keystore (i.e., cryptographic operations), Gatekeeper (i.e., device pattern/password authentication), Biometrics-based authentication (including face and fingerprint relevant operations), Android Protected Confirmation (i.e., secure prompt displayed to end-users), and Digital Rights Management (DRM). Then, we map these domains to the app-invoked TEE APIs. With these domains and metadata, we statistically analyze the relationships and trends, as discussed in the following sections. We plan to open-source our app dataset so that researchers can use and contribute to it.

B. Environment Setup

a) Real-world Apps: To carry out our empirical study, we gathered two datasets of real-world apps: one for regular apps and another for malicious apps. The former comes from Google Play Store, where we obtained around 200 "Top Free" apps[3] from each of the 33 categories defined by Google, using `google-play-scraper` [41]. The latter comes from the AndroZoo's malicious apps dataset [42], where we randomly selected 200 apps from each of 20 important types of malicious apps. Most of these types are documented by Google [43].

After removing invalid apps (e.g., cannot be downloaded online), our datasets contain 10,339 apps, including 6,349 regular (33 categories) and 3,990 malicious apps (20 categories).

b) TEE APIs: In total, our dataset of TEE APIs consists of the aforementioned `Direct APIs` and `Indirect APIs`. The former contains 75 functions collected from the specifications of GlobalPlatform and Trusty. The latter contains 619 functions among Android modules of KeyStore (129 functions), Biometrics-based authentication (250 functions), GateKeeper (152 functions), Digital Right Management(86 functions), and Android Protected Confirmation (2 functions).

Our tool identifies the use of TEE in an app by examining matches between the app's invocations and the TEE APIs above. For the `Indirect APIs`, we examine complete datasets of regular and malicious apps; for the `Direct APIs`, we focus on a sample group of apps due to the following reason: after examining 586 apps across three regular categories and 186 apps sampled from all malicious categories, we found no matches for `Direct APIs`. Hence, we hypothesize that Android apps may not directly integrate the `CA Libs` to invoke `Direct APIs`. To speed up the examination and test the hypothesis, we sampled 30 apps from each remaining category and matched them with `Direct APIs`. All the results and findings are discussed below.

[2]We did not analyze the functional domains of *Direct APIs* as it is mainly concerned with establishing communication channels between the normal and secure worlds. The IDs sent through *Direct APIs*, which identify the Trusted app and certain TEE operations, are outside the scope of our work.

[3]In the Google Play Store, "Top Free" refers to the most popular free apps. Note that we obtained a list of 200 "Top Free" apps, but some of their installation packages are unavailable online. Therefore, the actual number of obtained apps for each category may be slightly less than 200.

C. Characteristics of TEE-based apps

(1) Is TEE commonly used in apps?

a) Regular apps: Our result shows that 96.99% of regular apps use TEE. The top three categories with the highest use rates are "SPORTS"(100%), "TRAVEL_AND_LOCAL"(99.50%), and "VIDEO_PLAYERS"(99.49%). The bottom three are "WATCH_FACE"(86.96%), "LIBRARIES_AND_DEMO"(90.40%), and "FAMILY"(91.44%). Indeed, travel apps are more likely to collect sensitive user location data; sports and video player apps may require users to provide personal information and pay for live streaming or other advanced features. All these functionalities need secure operations. In contrast, certain apps such as watch faces, demos, and children's family games may not require extensive TEE operations.

b) Malicious apps: Interestingly, only 38.80% of malicious apps utilize TEE, which sets them apart from regular apps. In general, malicious apps aim to compromise mobile systems and apps instead of utilizing TEE to secure their malicious operations. Specifically, our result shows that the top three categories with the highest use rates are "Ransom" (91.5%), "Spr" (65.5%), and "Smssend" (64.65%), while the bottom three are "Fakeapp" (9.0%), "Clicker" (8.5%), and "Monitor" (2.5%). It is reasonable for ransomware to utilize TEEs to secure its ransom-related operations from interception and tampering. Features such as impersonation ("Fakeapp"), auto-clicking ("Clicker"), and behavior monitoring ("Monitor") may not heavily depend on TEEs.

(2) What is the difference between TEE-based and non-TEE apps? TEE-based apps have higher medians for rating scores (4.32) and real installs (2,481,691) compared to non-TEE apps with medians of 4.12 and 1,096,269, respectively. In fact, both rating scores and real installs can be indicators of an app's popularity. The result shows that the use of TEE is positively correlated with the popularity of the app. This correlation may be due to TEE's effective enhancement of the app's security.

(3) What communication channels are used by TEE-based applications? According to our analysis, we did not come across any apps that directly call upon the `Direct APIs` located in the `CA Libs`, which corresponds to the `Direct Channel`. That is, all functions that match are part of the `Indirect APIs` located in the `FW APIs of CAs`, which corresponds to the `Indirect Channel` (discussed in § IV-C and § IV-D). Therefore, we deem that, when using TEE, Android apps often indirectly invoke TEE APIs in the `CA Libs` via Android's framework APIs instead of directly integrating the `CA Libs` to call the corresponding TEE APIs.

<**Finding-4**> Based on the results mentioned above, we have identified three main characteristics of TEE-based apps: (a) overall, TEE is more commonly used in regular apps than malicious ones. However, some types of malicious apps may utilize TEE at a high rate; (b) the use of TEE is positively

correlated with the popularity of the app (i.e., improve its rating scores and installs); (c) both regular and malicious apps apply the `Indirect Channel` to communicate with the secure world instead of the `Direct Channel`.

D. The roles of TEE in apps

(1) What are TEE's popular application scenarios? After manually analyzing our dataset of TEE APIs and relevant Android documents, we summarized five typical application scenarios for TEE: cryptographic operations (i.e., Keystore), pattern/password-based authentication (i.e., Gatekeeper), biometrics-based authentication (i.e., face and fingerprint operations), secure prompts and warnings (i.e., the process of Android protected confirmation), and digital rights management (i.e., DRM operations).

TABLE II
THE NUMBERS AND RATES OF TEE APPLICATION SCENARIOS.

Application Scenarios	Regular Apps	Malicious Apps
Cryptographic Operations	5950(96.62%)	1449(93.60%)
Biometrics-based Authentication	3954(64.21%)	348(22.48%)
Pattern/Password-based Authentication	11(0.18%)	3(0.19%)
Secure Prompts and Warnings	203(3.30%)	0
DRM	179(2.91%)	108(6.98%)

Table II displays the number of regular and malicious apps for each TEE scenario, respectively. It also shows the ratio of this number to the total number of TEE-based regular and malicious apps, respectively (i.e., values in brackets). Among TEE-based regular and malicious apps, the top two popular TEE application scenarios are "cryptographic operations" and "biometrics-based authentication." At the same time, the bottom one is "pattern/password-based authentication" for regular apps and "secure prompts and warnings" for malicious apps. Specifically, most regular and malicious apps use TEE APIs for "cryptographic operations" (96.62% and 93.60%, respectively) and "biometrics-based authentication" (64.21% and 22.48%, respectively). Whereas, "pattern/password-based authentication" and "secure prompts and warnings" are used very little (0.18% and 0%, respectively).

It is easy to project that the most popular scenario for TEE is the "cryptographic operations," because TEEs can host cryptographic keys and code within the secure world and provide a set of relevant APIs. Surprisingly, the scenario of "biometrics-based authentication" is more prevalent than that of "pattern/password-based authentication." We believe that there are two main reasons: (1) the use of face and fingerprint-based operations has become popular in modern Android apps.; (2) "pattern/password-based authentication" in Android specifically refers to the lock-screen service, which is used by system-level apps (e.g., the Settings app) other than common apps we found on Google Play.

(2) What is the difference between app categories in TEE application scenarios?

Tables III and IV show the top/bottom three regular and malicious app categories in each TEE scenario, respectively. They also show the ratio of apps in a category that applies a specific scenario to the total number of apps in that category (i.e., values in brackets). "NA" means there are more than three categories have no apps that apply a specific scenario.

(a) For regular apps: As shown in Table III, in fact, apps within the "PERSONALIZATION," "SHOPPING," and "ENTERTAINMENT" categories are more likely to perform *cryptographic operations* than those within the "FAMILY," "WATCH_FACE," and "LIBRARIES_AND_DEMO" categories. Also, "FINANCE," "GAME," and "MEDICAL" apps need more *biometrics-based authentication* than "PHOTOGRAPHY," "MUSIC_AND_AUDIO," and "VIDEO_PLAYERS" apps. In addition to the "DATING," "FOOD_AND_DRINK," "PERSONALIZATION," and "LIFESTYLE" categories, we find that none of the apps in the other categories uses *pattern/password-based authentication*. Furthermore, most of the apps in the "BUSINESS," "PRODUCTIVITY," and "SHOPPING" categories make use of *secure prompts and warnings*. Besides that, we find that the majority of *DRM* operations are applied to categories that are intuitively irrelevant to digital rights (e.g., "FOOD_AND_DRINK," "COMMUNICATION," and "MEDICAL"). We plan to investigate this phenomenon in the future.

TABLE III
TOP AND BOTTOM-THREE REGULAR APP CATEGORIES IN EACH TEE APPLICATION SCENARIO.

Application Scenarios	Top Three	Bottom Three
Cryptographic Operations	PERSONALIZATION(100%), SHOPPING(99.49%), ENTERTAINMENT(99.48%)	FAMILY(90.06%), LIBRARIES_AND_DEMO(84.92%), WATCH_FACE(58.75%)
Biometrics-based Auth.	FINANCE(91.19%), GAME(90.58%), MEDICAL(88.64%)	PHOTOGRAPHY(41.18%), VIDEO_PLAYERS(40.21%), MUSIC_AND_AUDIO(39.89%)
Pattern/Password-based Auth.	DATING(4.44%), FOOD_AND_DRINK(0.53%), PERSONALIZATION(0.52%) and LIFESTYLE (0.52%)	NA[4]
Secure Prompts&Warnings	BUSINESS(14.44%), PRODUCTIVITY(13.37%), SHOPPING(10.26%)	NA
DRM	FOOD_AND_DRINK(12.30%), COMMUNICATION(11.92%), MEDICAL(9.66%)	NA

TABLE IV
TOP AND BOTTOM-THREE MALICIOUS APP CATEGORIES IN EACH TEE APPLICATION SCENARIO.

Application Scenarios	Top Three	Bottom Three
Cryptographic Operations	Addisplay(100%), Clicker(100%), Hacktool(100%), Monitor(100%), Smssend(100%)	Exploit(61.02%), Fakeapp(66.67%), Malware(83.54%)[5]
Biometrics-based Auth.	Downloader(84.81%), Fakeapp(66.67%), Exploit(45.76%)	Monitor(0%), Hacktool(1.11%), Smssend(5.47%)
Pattern/Password-based Auth.	Spr(1.53%), Dropper(0.91%)	NA
Secure Prompts&Warnings	NA	NA
DRM	Exploit(77.97%), Monitor(20.0%), Malware(7.59%)	NA

(b) For malicious apps: In our analyzed malicious apps, if they use TEE and belong to the "Addisplay," "Clicker," "Hacktool," "Monitor," or "Smssend" categories, they will always invoke *cryptographic operations* 100% of the time (as shown in Table IV). In addition, the "Hacktool," "Monitor," and "Smssend" prioritize *cryptographic operations* over *biometrics-based authentication* more than other categories, while the "Fakeapp" and "Exploit" do the opposite. Besides, only "Spr" and "Dropper" leverage *pattern/password-based*

[4]"NA" means there are more than three categories have no apps that apply a specific TEE scenario.

[5]"Malware" is an independent category in AndroZoo's collection of malicious apps.

authentication, and none of malicious app categories employ *secure prompts and warnings*. Further, we find the "Exploit", which aims to exploit software or system vulnerabilities to perform attacks, is more likely to use *DRM* operations than other categories.

<**Finding-5**> Based on the results mentioned above, we have identified three primary features of TEE scenarios: (1) the most popular scenario for TEE is the "cryptographic operations;" (2) when it comes to authentication scenarios, 'biometrics-based authentication" is more prevalent than the "pattern/password-based authentication" in Android TEE-based apps; (3) in TEE application scenarios, there is a clear distinction between different categories of apps, but the difference between regular and malicious apps is not noticeable.

E. Threats to Validity

The internal validity is threatened by (1) obfuscated app packages and (2) our conservative approach.

For (1), after decompiling obfuscated packages, the resulting code may lose original textual information, such as method and class names. Without this information, our detection algorithm's TEE API matching process would not function properly. Fortunately, upon manual inspection of the decompiled code, we discovered that the obfuscated app cases are not common. Moreover, we found that in many of the obfuscated apps, the function's call sites retain the original callee's textual information in their resulting code (i.e., smali code) even after decompiling. After careful consideration, we believe that this obfuscation case will only slightly impact our final results. Therefore, we decide not to consider it for now.

For (2), when collecting TEE APIs from Android's documentation, we consider an Android function a TEE API if the corresponding documents claim it "should" or "can" interact with the secure world of TEE. This conservative process means that whether the claimed interaction with TEE is mandatory or not, we mark the involved functions as the TEE API. Hence, our analysis may result in false positives. To mitigate this threat, we plan to design a fine-granded approach to identify the TEE APIs more accurately in the future.

The external validity is threatened by the limited number of TEE APIs we have collected. That is, although we gather TEE APIs from various sources such as official documentation, websites, and source code, it is impossible to exhaust all existing TEE APIs. To mitigate this threat, we will open-source our datasets to allow continuous contributions from other researchers to the TEE APIs dataset.

VI. Guidance for TEE practitioners

In this section, we provide suggestions to TEE practitioners according to our study results and findings.

(1) For testers: based on our findings, all the analyzed apps interact with TEE via functions in the Application Framework Layer. Hence, most TEE operations in an app can be tested by running test cases of the business logic in that layer.

(2) For TEE vendors: since the top two popular TEE application scenarios are "cryptographic operations" and "biometrics-based authentication," we recommend vendors to optimize their TEE operations in these scenarios to improve performance. Besides, a cost-effective way to create a smartphone-based TEE on the vendor's own is to target the Android platform and use standard APIs provided by GlobalPlatform.

(3) For Security researchers and developers:

a) About the interaction between TEE and apps: All of the TEE-based apps we examined use the `Indirect Channel` to communicate with the secure world, as opposed to the `Direct Channel`. We believe that opening more channels for third-party apps would enhance the development of both TEEs and TEE-based apps. To that end, security researchers and developers can create novel middleware to enable easy communication between `CAs` and `TAs`, as well as a mechanism for deploying them.

b) About malicious apps: Despite being less common than in regular apps, our findings indicate that a significant number of malicious apps make use of TEE. An app could conceal malicious activities using TEE to evade secure analysis or anti-virus tools. Therefore, security practitioners should not always assume that TEE operations are harmless. Rather, TEE-related operations from apps should be carefully examined.

VII. Related Work

(1) Empirical study on mobile apps: Xu et al. empirically studied real-world deep learning-based apps [5], and Wu et al. studied blockchain-based ones [6]. Zhan et al. systematically summarized the literature on third-party libraries' (TPL) usage in Android Apps [7]. Although these studies examine various aspects of apps, none of them focuses on those using TEE. In a latest research, Bove analyzed TEE-based apps to study their common usage patterns [12], [13]. However, this study did not take into account important app categories, such as malware or games, nor did it consider the architecture of smartphone-based TEEs and the interaction between TEEs and apps.

(2) Empirical study on TEEs: Liu et al. studied real-world SGX (i.e., Intel's non-mobile TEE solution) software [4]. Busch et al. analyzed the design/architecture of TrustedCore, a TEE in Huawei's earlier smartphones [8]. Cerdeira et al. studied the architectures and security features of five different TEEs [9]. Nevertheless, since more than 15 existing smartphone-based TEEs exist, these studies have not yet comprehensively covered them and their scenarios. Recently, Paju et al. surveyed academic papers and GitHub repositories related to TEE, summarizing various application scenarios [11]. However, their work lacks an analysis of commercial TEEs and real-world mobile apps.

(3) Detecting apps using certain modules: Xu et al. developed a detection tool that can effectively identify an Android app that uses deep learning models [5]. Busch et al. created a tool that can identify apps using TEE by analyzing TEE libraries provided by TEE vendors [44]. The former focuses on identifying the modules other than TEE. The latter is limited by the vendor's TEE libraries, which are not only difficult to obtain, but also challenging to reverse engineer. Besides, Bove developed a TEE operation detector based on TEE APIs [12], [13]. However, their work did not collect a sufficient number

of TEE APIs (≈200 vs. our ≈600) and relied on a simple filename-matching method instead of reverse engineering the native libraries (.so files), which reduced the accuracy.

VIII. Conclusion

We have conducted an empirical study on smartphone-based TEEs, TEE-based apps, and their interactions. In addition, we developed a detection tool that can identify TEE-based apps. By leveraging this tool, we have analyzed over 10,000 real-world apps, both regular and malicious, across 53 categories, and identified five findings that comprehensively depict both corresponding TEEs and apps. Our analysis results and tool can assist TEE vendors, app testers, and security researchers and developers with their TEE-related work.

Acknowledgements

The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. This research is supported by Beijing Natural Science Foundation (Grant No. 4232019).

References

[1] Google, "Trusty TEE," 2024, https://source.android.com/docs/security/features/trusty.

[2] Samsung, "Samsung TEEGRIS," 2024, https://developer.samsung.com/teegris/overview.html.

[3] Huawei, "Huawei iTrustee," 2020, https://globalplatform.org/certified-products/huawei-itrustee-v3-0-on-kirin-980/.

[4] Y. Liu, S. Dhar, and E. Tilevich, "Only pay for what you need: Detecting and removing unnecessary tee-based code," *Journal of Systems and Software*, vol. 188, p. 111253, 2022.

[5] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, 2019, pp. 2125–2136.

[6] K. Wu, Y. Ma, G. Huang, and X. Liu, "A first look at blockchain-based decentralized applications," *Software: Practice and Experience*, vol. 51, no. 10, pp. 2033–2050, 2021.

[7] X. Zhan, T. Liu, L. Fan, L. Li, S. Chen, X. Luo, and Y. Liu, "Research on third-party libraries in android apps: A taxonomy and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4181–4213, 2021.

[8] M. Busch, J. Westphal, and T. Mueller, "Unearthing the TrustedCore: A critical review on Huawei's trusted execution environment," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/busch

[9] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1416–1432.

[10] 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.

[11] A. Paju, M. O. Javed, J. Nurmi, J. Savimäki, B. McGillion, and B. B. Brumley, "Sok: A systematic review of tee usage for developing trusted applications," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, 2023, pp. 1–15.

[12] D. Bove, "A large-scale study on the prevalence and usage of tee-based features on android," *arXiv preprint arXiv:2311.10511*, 2023.

[13] ——, "A large-scale study on the prevalence and usage of tee-based features on android," in *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ser. ARES '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3664476.3664486

[14] GlobalPlatform, "GlobalPlatform, TEE system architecture, technical report," 2022, https://globalplatform.org/specs-library/tee-system-architecture/.

[15] Intel, "Intel Software Guard Extensions (SGX)," 2024, https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html.

[16] ——, "Intel Processors Supporting Intel SGX," 2024, https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-processors.html.

[17] Apple, "Secure Enclave," 2021, https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web.

[18] Qualcomm, "Guard your Data with Qualcomm Snapdragon Mobile Platform," 2021, https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf.

[19] GlobalPlatform, "Global Platform," 2024, https://globalplatform.org/.

[20] StatCounter, "Mobile Operating System Market Share Worldwide," 2024, https://gs.statcounter.com/os-market-share/mobile/worldwide.

[21] 42matters, "Google Play Statistics and Trends 2024," 2024, https://42matters.com/google-play-statistics-and-trends.

[22] GlobalPlatform, "Certified Products," 2024, https://globalplatform.org/certified-products/.

[23] Huawei, "HarmonyOS 2 Security Technical White Paper," 2021, https://consumer-img.huawei.com/content/dam/huawei-cbg-site/common/campaign/privacy/whitepaper/harmonyos-2-security-technical-white-paper-v1.0.pdf.

[24] Trustonic, "Kinibi-520a: The latest Trustonic Trusted Execution Environment (TEE)," 2024, https://www.trustonic.com/technical-articles/kinibi-520a-the-latest-trusted-execution-environment-tee/.

[25] Linaro, "OP-TEE documentation," 2024, https://optee.readthedocs.io/en/latest/index.html.

[26] EASTCOMPACE, "TEEI security terminal," 2023, https://www.eastcompeace.com/product/18.html.

[27] Beanpod, "ISEE Trusted Execution Environment Platform (Secure OS)," 2023, https://www.beanpodtech.com/en/products/.

[28] Watchdata, "Terminal-connected Security Solution," 2023, https://www.watchdata.com/terminal-connected-security-solution/.

[29] Rocky Core Technology, "TEE," 2022, https://www.rockycore.cn/contents/70/549.html.

[30] TrustKernel, "T6," 2021, https://www.trustkernel.com/products/device/tee/t6.html.

[31] Alibaba, "What is Link-TEE," 2019, https://help.aliyun.com/document_detail/126299.html?spm=5176.cniotid_tee_pre.0.0.604d2d17PZK8tJ.

[32] Xiaomi, "Xiaomi HyperOS," 2019, https://hyperos.mi.com/.

[33] SecEdge Inc., "Sequitur Labs' Enhances IoT and Embedded Security with CoreTEE," 2015, https://www.live.secedge.com/media_portfolio/sequitur-labs-enhances-iot-and-embedded-security-with-coretee/.

[34] ProvenRun, "ProvenCore: The essence of Trust," 2023, https://provenrun.com/provencore/.

[35] Sierraware, "Secure OS and Hypervisor - TEE for MIPS," 2017, https://www.slideshare.net/sgopu/trustzone-secure-os-tee-for-mips.

[36] G. Platform, "TEE Client API Specification v1.0," 2010, https://globalplatform.org/specs-library/tee-client-api-specification/.

[37] Google, "Trusty API Reference," 2024, https://source.android.com/docs/security/features/trusty/trusty-ref.

[38] ——, "Android Developers," 2024, https://developer.android.com/.

[39] ——, "Android Code Search," 2024, https://cs.android.com/android/platform/superproject/main.

[40] JoMingyu, "Google-Play-Scraper," 2024, https://github.com/JoMingyu/google-play-scraper.

[41] F. Olano, "google-play-scraper: Node.js module to scrape application data from the Google Play store." 2024, https://github.com/facundoolano/google-play-scraper.

[42] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 425–435.

[43] Google, "Malware," 2024, https://developers.google.com/android/play-protect/phacategories.

[44] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, "Teezz: Fuzzing trusted applications on cots android devices," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1204–1219.