GateKeeper: Operator-centric Trusted App Management Framework on ARM TrustZone

Balachandar Gowrisankar
National University of Singapore
e0674491@u.nus.edu

Daisuke Mashima, Wenshei Ong, Quanqi Ye, Ertem Esiner
Illinois at Singapore Pte Ltd
{daisuke.m, wenshei.ong, quanqi.ye, e.esiner}@adsc-create.edu.sg

Binbin Chen
Singapore University of Technology and Design
binbin_chen@sutd.edu.sg

Zbigniew Kalbarczyk
University of Illinois Urbana-Champaign
kalbarcz@illinois.edu

***Abstract*—Employing Trusted Execution Environment (TEE) technology such as ARM TrustZone to deploy sensitive security modules and credentials for secure, authenticated access is the go-to solution to address integrity and confidentiality challenges in untrusted devices. While it has been attracting attention as an effective building block for secure enterprise IT systems (e.g., BYOD), these secure operating systems are often not open-source, and thus system operators and developers have to largely depend on mobile platform vendors to deploy their applications in the secure world on TEE. Our solution, called GateKeeper, addresses the primary obstacle for system operators to adopt ARM TrustZone TEE to deploy their own, in-house security systems, by enabling the operators more control and flexibility on Trusted App (TA) installation and update procedure without mandating involvement of the mobile platform vendors at each iteration. In this paper, we first formulate an ecosystem for enabling such operator-centric TA management, and then discuss the design of GateKeeper, which is a comprehensive framework to enable operator-centric TA management on top of GlobalPlatform specification. We further present a proof-of-concept implementation using OP-TEE open-source secure OS to demonstrate the feasibility and practical resource consumption (less than 1000 lines of code and 500 KBytes on memory).**

I. Introduction

In recent years, mobile devices (smartphones, tablets, etc.) are getting prevalent in both the personal and professional contexts. Such devices are often utilized for sensitive tasks, such as gaining remote access to the office network and performing critical transactions. Consequently, the number of attacks targeting such devices, compromising integrity and confidentiality, also increases. To counter these cybersecurity threats, CPUs in modern mobile devices support a special execution environment, called Trusted Execution Environment (TEE) [1]. GlobalPlatform [2] is the organization that makes specifications for TEE and related technologies. TEE is an execution environment isolated from the rest of the CPU at hardware level. ARM TrustZone [3] is the TEE implementation on ARM processors, which is widely used for mobile and IoT devices. The trusted and non-trusted execution environments correspond to two worlds during execution – *secure world* and *normal world*. The operating systems (OS) that run in the secure and normal worlds are called *secure OS* and *rich OS* respectively. Correspondingly, an application (app for short hereafter) that runs in the secure world is called Trusted App or *TA*. Unlike the normal world, the secure world ensures authenticity, integrity, and confidentiality of TAs against adversaries outside of the secure world [4]. TEE's security features, such as secure boot and isolation of worlds, have attracted surging attentions in both academia and industry. TEE can offer a root of trust for implementing security features for various contexts. For instance, in enterprise IT systems, the secure OS can ensure integrity and confidentiality of security credentials (e.g., an encryption key [5] and other meta data [6], [7]) and security critical modules for secure remote access, user/device authentication, and access control. Therefore, an increasing number of system operators and software developers are interested in using such technology to fortify their devices and apps from being compromised.

However, most of the popular secure OSs in the market (e.g., Trustonic's Kinibi [8], [9], Samsung's KNOX [10], Qualcomm's QSEE[1], and Huawei's TrustedCore[2], just to name a few) are close source. Thus challenge remains for the end users in developing, distributing, installing TAs on their own mobile devices. In other words, because of this reason, system operators (e.g., a company that wants to utilize TEE for secure remote access) or third-party developers are not able to freely (i.e., without involvement of a mobile platform vendor that controls the secure OS) develop and distribute their TAs even for their own systems [18]. This has been a significant obstacle to utilize TEE technologies for securing their in-house systems. Some open-source secure OSs, such as OP-TEE [19], and hybrid licensed secure OS, such as Sierra TEE [20], are expected to address this challenge, but its adoption is limited. This may further imply that the system operators or third-party developers are required to disclose the source codes of their TAs, which may include some corporate confidential logic or information, to the mobile platform vendors. Therefore, the current ecosystem requires additional trust assumption on the

[1]There is no official website for QSEE. It is only known to belong to Qualcomm from papers [11], [12], slides [13], [14] and blogs [1], [9].

[2]There is no official website for TrustedCore. It is only known to belong to Huawei from papers [15]–[17] and blogs [9].

mobile platform vendors (in not misusing or disclosing TA source codes/binary). More notably, such a requirement poses a significant inconvenience for a use case of emerging demand, namely bring-your-own-device (BYOD). Owing to the pandemic, many companies are encouraging their employees to work from home. Employees may often utilize their own commercial-off-the-shelf (COTS) devices to work remotely and access the office network. Because of the high dynamics of BYOD user base as well as the fact that devices are owned and utilized for other daily tasks by each end user (e.g., an employee of the system operator and/or third-party contractor who has business contract with the system operator), it is impractical to hand over devices to the mobile platform vendor for TA maintenance (installation and update).

In this paper, we propose a framework, *GateKeeper*, to enable TA maintenance by system operators, without involving mobile platform vendors for each iteration, after the one-time setup of the secure OS (i.e., in a system-operator-centric manner). GateKeeper enables (and also ensures) continuous installation/update of only TAs that are implemented by authorized developers and are approved by the system operator. GateKeeper solves flexibility and openness challenges by means of a "gateway" TA, which is pre-installed with the secure OS upon the one-time setup by a mobile platform vendor. Our framework guarantees confidentiality and integrity of TAs throughout their life cycle against attackers outside of the secure world. We also implemented a proof-of-concept on OP-TEE secure OS [19]. We expect such flexibility and control encourage system operators to adopt TEE technologies for their in-house systems. Our contributions are:

- We formulated an ecosystem for the system-operator-centric TA management.
- We designed the GateKeeper framework for ARM TrustZone devices to enable secure installation of authorized TAs under the operator's discretion and responsibility.
- We developed the proof-of-concept implementation of GateKeeper on OP-TEE secure OS to demonstrate the feasibility as well as to measure overhead.

GateKeeper is designed as an additional security and key management layer on top of the TA installation mechanism defined by GlobalPlatform. While our design and prototype focuses on open-source platform, OP-TEE, our intention is not to develop a technology only for OP-TEE platform. The concept we develop in this paper can be applied to other TEE platforms, and thus we hope that commercial mobile platform vendors evaluate our proof of concept to consider integration of GateKeeper into their mobile devices.

The rest of this paper is organized as follows. We introduce the background knowledge and context related to TEE in Section II. Then we introduce the design and architecture of our framework in Section III. The implementation of the framework is discussed in Section IV. Security analysis and performance/overhead evaluation are done in Section V and Section VI respectively, followed by supplementary discussions in Section VII. Related works are discussed in Section VIII. Finally, we conclude this paper in Section IX.

II. Background and Motivation

To familiarize the readers with TEE-related technologies, we first introduce some background knowledge in this section. We then discuss a challenge that has motivated this work.

A. TEE and Trusted App (TA)

Trusted Execution Environment (TEE) is an isolated execution environment that allows the secure OS and TAs to be stored and executed securely [2]. On the other hand, the usual execution environment is called the Rich Execution Environment (REE) where the Rich OS and normal applications run. TrustZone is a concrete TEE implementation by ARM. When TEE is in operation, the ARM processor will be running in two separated and isolated worlds, the secure world and the normal world. The world that is currently running can be distinguished by the value of the Not Secure Bit (NS bit) in the Secure Configuration Register (SCR) in co-processor CP15 [21]. The NS bit is propagated to the memory address and peripheral devices. Therefore, memory and peripheral devices can be configured to be shared by both worlds or be accessible solely by either the secure OS or the Rich OS. In the usual configuration, the secure OS has higher privilege than the rich OS and it has access to system resources including the registers, memory, peripherals, and so on.

The larger the size of the software, the greater is the possibility that there exist bugs and vulnerabilities in that software. Thus, to ensure the security of the software running in the secure world (trusted computing base or TCB), it is better to keep the size of the software running in the secure world as small as possible. Therefore, to achieve minimal TCB size, secure OSs are developed from scratch rather than reusing the traditional OS kernels and their code base. In addition, normally, the secure OS does not follow the traditional Portable Operating System Interface (POSIX) standards and programming paradigms. GlobalPlatform is responsible for enforcing the standard APIs for the secure OS. The TAs need to follow a special set of APIs and a programming paradigm so that they can be loaded and executed in secure OSes that follow the GlobalPlatform specification.

Secure OS and TAs run in the secure world. The world switching is done via secure monitor call (SMC) that can be raised in both worlds. SMC includes the UUID (Universally Unique Identifier) of the TA to be invoked. SMC can only be handled by the SM, which is responsible for the world switch. Thus, it is critical to ensure the security of its logic and implementation. TAs in the secure world can store data in the storage called secure objects. Secure objects are typically stored in the file system of the normal world, encrypted and signed using the key managed by the secure OS. Installed TAs themselves are also stored and protected in a similar manner.

B. Motivation

There are several stages during the secure boot process where the board loads different firmwares and OS images.

Each stage contains the integrity information for the next stage and it checks, verifies, and authenticates the integrity of the firmware or images using the integrity information known to it. The chain of trust (CoT) is established in this way. However, it is difficult to update or modify an individual component in this chain. Since the inception of ARM TrustZone, many secure OSs, including proprietary ones (e.g., by Samsung and Huawei) and open-source ones (e.g., OP-TEE [19]), have been developed. It is also imperative to ensure that the secure OS only installs and executes the verified and authenticated trusted apps in the secure world. Otherwise, the security guarantee provided by the secure OS is not different from the one offered by rich OS in the normal world. There should be a mechanism to propagate the CoT to the point of executing the trusted apps in the secure OS. However, it is also equally important for the legitimate system operators and software developers that are appointed by the operators, to conveniently develop and distribute their TAs and install them on the end-user devices. Without addressing this challenge, TEE is not readily beneficial for system operators to adopt for securing their in-house systems. For instance, when we consider a system to enable secure BYOD, involvement of a mobile platform vendor whenever installing and updating TAs on user-owned device is highly inconvenient.

One solution to address this challenge is to design and implement a built-in TA that works as a gateway for installing TAs and updates that are authorised by the system operator. Such a TA is to be carefully evaluated and installed by a mobile platform vendor upon setup of the secure OS. This way, once the gateway TA is securely set up by the platform vendor, the system operator (e.g., an enterprise IT operator or a critical infrastructure operator) can conveniently install or update their own TAs at later times without having the mobile platform vendor in the loop. In other words, the mobile platform vendor can let the system operators owe responsibility for the security of the TAs for operational flexibility. Unfortunately, there is no established framework to meet such practical demands. Thus, in this paper, we first define the ecosystem and trust relationship among the entities. We then discuss design of the gateway TA, named *GateKeeper*, and present a proof-of-concept implementation of it.

III. GateKeeper Framework

A. Entities in Operator-centric TA Management EcoSystem

We first summarize the relationship and trust models among the parties involved. As seen in Fig. 1, there are four different entities, namely: a *system operator*, a *mobile platform vendor*, a *software developer*, and a *mobile device user*.

A *system operator* is an entity that is responsible for the secure operation of its in-house system (e.g., BYOD). Typically, in both the enterprise IT and critical infrastructure operation settings, the organization's IT administration department is seen as the system operator. An Operator is also responsible for secure operation of mobile devices owned and utilized by *mobile device users* (e.g., employees or third-party contractors that work for the system operator). In some cases, the system

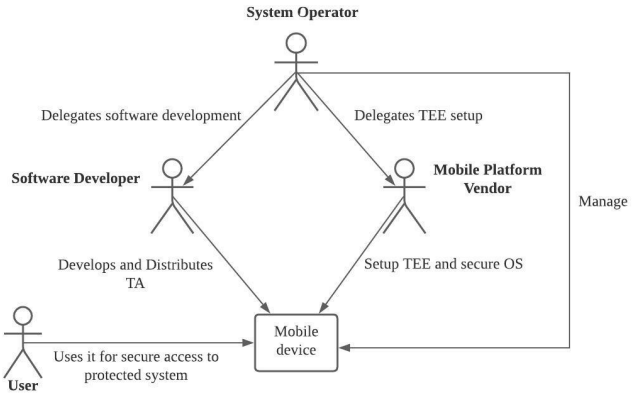

Fig. 1. Relationship among Involved Parties.

operator may establish a business contract to delegate software development to a third party *software developer*. In other cases, the system operator itself (e.g., a development team in the same organization) can develop the software. We assume software developers are legally bound by the contract (e.g., NDA) with the system operator and thus are trusted not to disclose any confidential information, including source codes and binary of TAs, and other secret data such as cryptographic keys, with other entities, including mobile platform vendors and mobile device users.

A *mobile platform vendor* is an entity responsible for setting up the TEE and secure OS on ARM TrustZone devices upon request from the system operator. For instance, in the case of commercial TEE platforms, such as Samsung KNOX, Samsung sets up and installs TEE and secure OS on mobile devices they sell. Such a provider can work with client organizations (i.e., system operators) to implement and set up security applications in the secure world according to the client's needs. On the other hand, the TEE platform is not open to the client, which prevents the client from flexibly installing or updating the applications in the secure world. In this paper, we assume this closed nature of the TEE platform remains. Mobile platform vendors are trusted by a system operator in configuring the secure OS with pre-installed TAs appropriately. However, they can be honest-but-curious adversaries. For instance, if the source code of TA is disclosed, which is the case under the current vendor-centric model, they may attempt to acquire secret information coded/stored in a system operator's TA.

B. Security Goals and Threat Models

In order to maximize flexibility in management by the system operator under the discussed ecosystem, we propose the following operator-centric framework, called *GateKeeper*.

- The *system operator* can choose a *software developer* to implement TAs for their in-house systems (e.g., for remote access) and also distribute the TAs.
- *Mobile device users* (employees or third-party contractors), who want to use the in-house system run by the system operator, require a one-time setup of TEE and

secure OS by a *mobile platform vendor* who is delegated the setup task by the *system operator*.
- Afterwards, the *mobile device user* obtains (e.g., downloads from a web site) TAs authorized by the *system operator* and installs/updates them on their devices anytime.

GateKeeper framework must satisfy a set of security properties. Next, we define our security goals.

Prevention of Unauthorized TA Installation. Since modification of modules in the secure world may affect the overall security guarantee, the GateKeeper system should only allow installation of authorized TAs into the secure world. Thus, the primary security goal of GateKeeper is to prevent any unauthorized TAs from being installed into the secure world. More specifically, GateKeeper only allows installation of TAs that are provided by software developer(s) who are approved by or have contract with the system operator. In addition, it is necessary to check that only the authorized TA binary, which is scrutinized and approved by the system operator, should be installed. Therefore, even if the TA developer becomes malicious (or is compromised), TA binary that is not approved by the system operator is not installed.

Protection of Integrity and Confidentiality of TA. The other challenge is the integrity and confidentiality of TA binary against attackers outside of the secure world, in transit or at rest in the normal world. Integrity protection is important to ensure the first security goal. Confidentiality against any entity other than the system operator and the software developer is also vital since TA binary may include sensitive information (e.g., hardcoded keys and/or corporate secrets). Thus confidentiality of TA binary before installation must be ensured against attackers on the network or in the normal world. Additionally, it is also required that the system operator can control who (and which device) can access the TA code and binary. For example, mobile platform vendors or mobile devices owned by revoked users should not be able to access them.

The above goals are to be met under the following threat models and the trust assumptions discussed in Section III-A. We assume that TEE and components in the secure world (secure OS and TAs) are trusted and securely bootstrapped. While attacks against secure OS (e.g., [22]) may be possible, it is an orthogonal research problem and thus left outside of our scope. On the other hand, components in the normal world or outside of the devices (e.g., entities in the network) are not trusted, and thus attacks could be mounted there.

C. Framework Design

We now elaborate the details of GateKeeper design and step-by-step description of procedures for TA installation. The overview of the framework is shown in Fig. 2. Supplementary discussions on the design decisions are also found in Section VII. GateKeeper framework involves GateKeeper TA, which is split into two TAs (*intermediate TA* and *pseudo TA*), a trusted server run by the system operator, and a software developer. GateKeeper TA is invoked by the *GateKeeper proxy app* in the normal world. The intermediate TA does not have kernel-level privileges and thus cannot directly execute TA installation. On the other hand, the pseudo TA with kernel-level privileges cannot be invoked directly by the proxy app. Below, we describe the procedures of the GateKeeper framework, starting with the preparation phase as Step 0, which consists of 2 tasks.

Step 0 (TEE/GateKeeper TA Setup). The mobile platform vendor sets up the secure OS as well as GateKeeper TA (both of the intermediate TA and pseudo TA). The intermediate TA is configured with the URL of the trusted server along with its digital certificate. UUID of the pseudo TA is randomly selected for each device and is reported to the system operator. As mentioned above, since the normal world modules cannot access the pseudo TA directly, this pseudo TA's UUID is not visible to normal world components as well as device users and can be known only to the system operator (besides the mobile platform vendor). Thus, it can be used as a secret, *device ID*. In our framework, this device ID is used to identify and manage GateKeeper-enabled devices at a later stage.

Step 0 (Developer Registration). The software developer needs to register with the system operator (specifically, to its trusted server). In a business-to-business relationship, this step can be done as part of the contract phase. During this registration process, a symmetric key K_{enc} and a public/private key pair (K_{priv} and ${K_{priv}}^{-1}$) are generated for each developer by the system operator. The symmetric key K_{enc} will be used by the developer to encrypt the TA binary before sending it for installation. The developer is responsible for securely storing the private key (K_{priv}) that is to be used to sign the developed TA. After this step, the trusted server run by the system operator stores the registered developer's public key to be used in later steps.

Step 1. The software developer implements the TA according to the specification and API of the secure OS. The developer incorporates all the necessary libraries and files by statically linking them into a standalone executable (a.k.a., .ta file). Then, the developer encrypts the TA executable using K_{enc} and signs it using the private key (K_{priv}). The developed TA is submitted to the system operator for its inspection. After the TA passes the inspection, the system operator adds the hash value of the TA executable into the list of approved TAs.

Step 2. The developer disseminates the TA via whatever possible ways. For example, the developer can disseminate the TA on a website and allow mobile device users to download the (encrypted and signed) TA into their devices.

Step 3. After the encrypted and signed TA is placed in the normal world of the mobile device, the user then invokes the proxy app that resides in the normal world.

Step 4. After the user has invoked the proxy app to initiate the installation process, the proxy app loads the encrypted and signed TA executable into the shared memory for transferring the data to the secure world. It then invokes a secure monitor call (SMC) to switch to the secure world.

Step 5. In the secure world, the GateKeeper intermediate TA communicates with the trusted server using TLS with the pre-configured public key of the trusted server, for downloading/updating the trusted developer/TA information. The

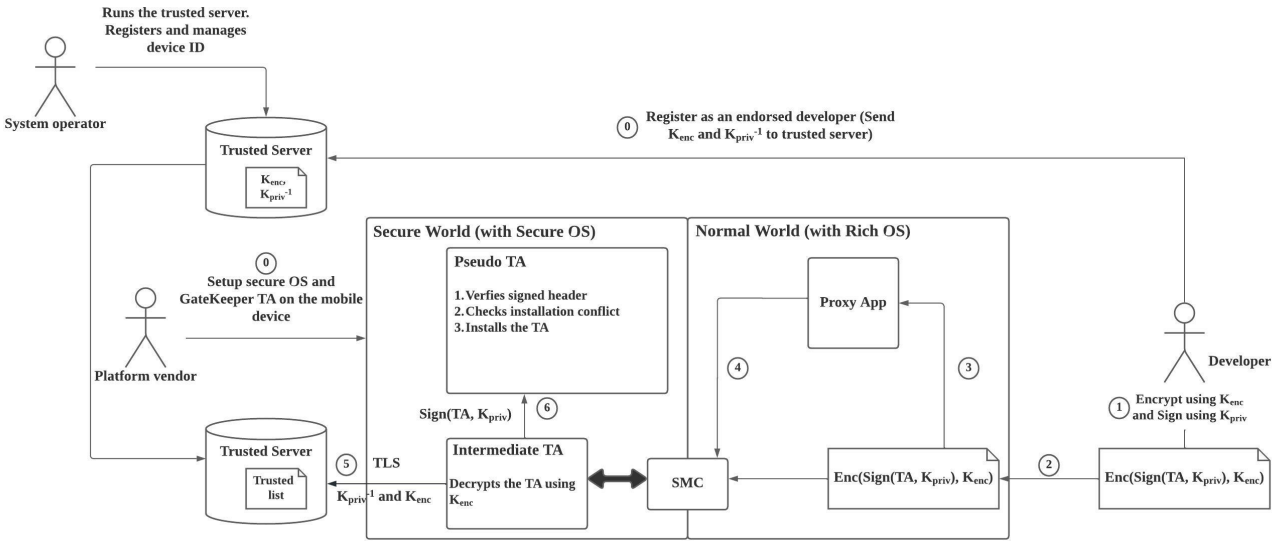

Fig. 2. Overview of GateKeeper

information to be downloaded includes public keys and shared secret keys of authorized developers as well as the list of hash values of the approved TAs. The downloaded information can be stored in the secure world for a limited duration, e.g., a few hours, for potential needs of offline installation. The intermediate TA securely enforces the lifetime of them. When downloading, the intermediate TA sends the device ID (i.e., UUID of the pseudo TA) so that the trusted server rejects requests from unauthorized/revoked devices or entities (i.e., request with unknown or invalid device ID). After downloading is done, the intermediate TA proceeds to decryption of the TA binary using K_{enc}. The decrypted TA binary is then sent to Gatekeeper pseudo TA.

Step 6. GateKeeper pseudo TA checks the developer's signature by using the developer's public key. The hash values of the TA binary is then compared against the list of hash values of the approved TAs. After successful verification, the provided TA executable is registered in the secure world. We should note that, while the TA binary is stored in the normal world file system, it is encrypted and integrity protected by the secure OS [23].

The same workflow is used for updating an installed TA. Each TA has metadata including the unique identifier and the version number. If a TA binary with the same identifier comes for installation with a newer version, then the secure OS performs update of the corresponding TA.

IV. Proof-of-concept Implementation

We implemented the framework on QEMU [24] based emulated environment. We set up QEMU with OP-TEE according to the official documentation [25]. The QEMU environment is set to emulate an ARM Cortex-A15 processor with TrustZone support. The whole framework and secure OS are implemented using the C programming language. We install Linux (a BusyBox image) in the rich OS and install the OP-TEE in the secure world as the secure OS. The proxy app, which runs on the rich OS, is implemented in C, and offers minimal functionality to do the tasks discussed in Step 4 in Section III-C (around 100 lines of code). We note that the normal world can run a commodity OS, such as Android. In this case, the proxy app can be implemented as an Android app by using JNI (Java Native Interface) to call OP-TEE APIs.

Since our implementation utilizes OP-TEE, an open-source secure OS, let us briefly explain the format of the compiled TA in OP-TEE. Fig. 3 summarizes the structure of the TA binary file (.ta file) [26]. We should note that the format is not standardized in Global Platfrom's specification and may differ on each secure OS implementation. The "signed header" section indicates the algorithms for hashing and signing as well as sizes of the hash value and signature. It is followed by "hash" value of the header and the compiled TA code. In the GateKeeper framework, the signature is made by an authorized software developer's private key. Then the entire binary shown in Fig. 3 is encrypted with the developer's secret key before sending the TA out for installation.

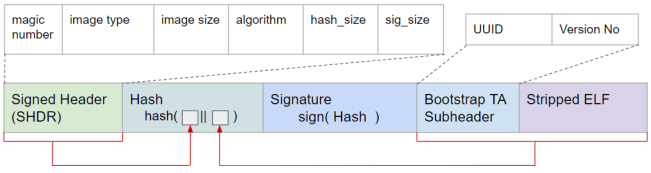

Fig. 3. Structure of TA binary (.ta file)

The GateKeeper TAs (the intermediate TA and privileged, pseudo TA) resides in the secure world and is responsible for installation of the TA binary passed by the proxy app. The decryption, signature verification, and hash calculation are performed by using the cryptographic API provided by OP-TEE. After the decryption and authentication of the TA binary, the secure OS TA invokes the OP-TEE API to register TA to the database managed by the secure OS. Then, the installed TA is stored in the normal world file system in an encrypted and signed form by using keys embedded in the secure OS

(REE filesystem TA [23]).

The GateKeeper intermediate TA also communicates via TLS with the trusted server to fetch or update the authorized developer keys and the list of hash values of approved TAs. In general, to ensure that the secure OS is vulnerability-free, it is necessary to keep the trusted computing base as small as possible so as to keep it easy to test and identify any security vulnerability. Therefore, TLS functionality is not included in the OP-TEE secure OS. To enable the TA in OP-TEE to communicate securely with a trusted server on the Internet, we had to statically incorporate a TLS library to provide secure communication between the trusted server and the secure world. In our proof-of-concept implementation, we incorporated *WolfSSL* [27] as the TLS library because it is designed and developed to be used in the embedded devices environment. Using WolfSSL APIs, the GateKeeper intermediate TA can establish secure, authenticated communication channel to the operator's trusted server to download developer keys and list of approved TAs' hash values, without being affected by potential adversaries in the normal world.

V. Security Analysis

This section provides discussions on the effectiveness of GateKeeper in meeting the security goals (see Section III-B) under various threat scenarios.

(1) Prevention of Unauthorized TA Installation

One of the primary goals of designing GateKeeper is to ensure that only applications approved by the system operator can be installed onto the device. In GateKeeper, this goal is accomplished by mainly two mechanisms: digital signature of authorized software developers on TA binary and a list of hash values of TAs approved by the system operator, which are securely downloaded from the trusted server run by the system operator.

As explained in Section IV, the GateKeeper TA checks the authorized developer's digital signature on the TA to be installed. We should note that the public keys are downloaded upon each invocation of the GateKeeper TA (and cached only for a limited duration). Thus, revocation of the public keys can be enforced in timely manner. An entity without the knowledge of a valid/effective private key cannot pass the signature verification done in the secure world.

GateKeeper contains a component in the normal world, namely the proxy app. The proxy app may be compromised or manipulated by attackers in the normal world to attempt the installation of unauthorized (or tampered) TAs. However, as long as the TAs are not accompanied by the valid signature of the software developer, the TA is not accepted by GateKeeper TA. The same argument also holds against attackers in transit between the developer and the device.

In a scenario where multiple developers are authorized to develop TAs, we also need to prevent a malicious developer from impersonating other benign developers to install their malicious TA onto the devices. GateKeeper requires that all developers pre-register themselves with a system operator, upon which each developer establishes and shares a unique public/private key pair. Thus the malicious developer cannot make a valid signature by impersonating another developer.

While the software developer is trusted in that they don't leak keys, it would still be possible that the keys are stolen or abused to distribute bogus TAs. In such a case, while the bogus TA can be signed and encrypted appropriately, it is never seen by the system operator and thus its hash value is not registered on the trusted server. As the result, it fails in passing the hash value verification done by the GateKeeper TA.

(2) Protection of Integrity and Confidentiality of TA

Another goal is to protect confidentiality and integrity of TA binaries. The encryption by using the software developer's secret key, TLS secure communication with the trusted server initiated from the secure world, and functionality of secure OS together contribute to meet this goal.

For installation, the TA binary needs to be temporarily stored in the normal world. However, TA binary before installation is encrypted by using the secret key assigned to each software developer, and thus an attacker in the normal world cannot break confidentiality. Integrity is ensured through the digital signature. Note that both decryption and signature verification are done after the TA binary is passed to the secure world. After the installation, the TA binary is typically stored on the file system of the normal world. However, according to the specification of OP-TEE, installed TA binary is encrypted and signed using the keys generated by the secure OS. Thus, throughout the lifecycle of the installed TA, confidentiality and integrity are ensured.

Under our framework, the trusted server is (and has to be) publicly accessible, and thus, unauthorized or malicious parties may attempt to download the developer's symmetric encryption key. This may result in a situation where the confidentiality of the TA binary is compromised, or the TA is installed on unauthorized devices (e.g., devices that belong to former employees). To prevent this, we introduced verification of the device ID, which is the UUID of pseudo TA installed on each mobile device as discussed in Section III-C, by the trusted server. Device IDs are uniquely set when the mobile platform vendor sets up the secure OS and the GateKeeper TA and is reported to the system operator for registration on the trusted server. We should also note that the device ID is not known to even a legitimate mobile device user. Furthermore, the list of effective device IDs maintained on the trusted server can be updated if a mobile device user is revoked (e.g., when he/she leaves a company). Therefore, in order for a malicious entity to download the keys, he/she is required to include a valid device ID. However, it is not feasible, with typical countermeasures against brute-forcing such as rate limiting, since the space of the UUID is 128 bits. Additional discussions regarding device authentication, including an alternative way, is provided in Section VII. It may be argued that the downloaded keys remain in the secure world of the device even after revocation. However, the lifetime of the cached keys and deletion are enforced by the GateKeeper TA in the secure world.

Speaking of confidentiality, we also need to consider the mobile platform vendors. Mobile platform vendors are trusted

in that they appropriately set up TEE and secure OS as well as install GateKeeper TA. Once that is done, mobile platform vendors are not involved in the TA installation or update procedure. Thus, they are not aware of the content of the system operators' TAs. Thus, any confidential information in the TAs is not accessible to the mobile platform vendors. Besides, since the mobile platform vendors can know device IDs and the address of the trusted server, they could technically download the encryption keys from the trusted server. If the platform vendor somehow obtains the encrypted TA binary, there will be no confidentiality. While this may be considered as a limitation, we consider this is outside of the honest-but-curious adversary model we assumed, which typically exploits information that they legitimately can access.

VI. Performance and Overhead Evaluation

A. Footprint of GateKeeper

GateKeeper TA, along with OP-TEE secure OS, is part of the trusted computing base, and thus its correctness and integrity must be carefully validated. Therefore, its footprint and size of codes should be manageable to undergo rigorous testing to eliminate vulnerabilities.

In order to measure the complexity of the module, we show the lines of code for the intermediate TA and pseudo TA. Note that the count does not include OP-TEE API and WolfSSL library. The intermediate TA consists of 767 lines, and the pseudo TA consists of 193 lines of code. Given this small footprint, it is tractable to investigate the correctness thoroughly to make the codes vulnerability-free.

B. Overhead to Implement GateKeeper

In order to facilitate the integration of the GateKeeper framework into commodity devices, it is crucial that resource consumption is acceptable. Using the prototype implementation on QEMU, we measured the memory and storage overhead required for GateKeeper, which is summarized in Table I. Among the measurements, since we were not able to find the size of the pseudo TA executable, we measured the intermediate object file before generating the executable. Since the pseudo TA only utilizes APIs provided by OP-TEE, we assume that the eventual executable size is not significantly different. Secure objects, in a signed and encrypted form, are visible in the normal world. Thus we measured the total size on the normal world file system, after downloading keys for a single developer and 1 TA's hash value. If we consider multiple developers and approved TAs, then the size is expected to grow linearly, but in the expected use cases, the number of developers and TAs are usually small. Thus, overall storage/memory usage is considered practical.

TABLE I
Memory and Storage Overhead

Component	Size (KByte)
RAM Usage of OP-TEE OS Image	469
GateKeeper Intermediate TA Executable	295
GateKeeper Pseudo TA Object Size	115
Secure Object Size	65

Besides, while GateKeeper is invoked only when TA installation/update needs to be performed, which is considered infrequent and does not require real-time'ness, running TAs in the secure world suspends execution in the normal world modules. Thus, latency for execution of GateKeeper should not be too long. As discussed, the installation process of GateKeeper involves the following tasks:

1) Decrypting TA binary to be installed
2) Downloading from the trusted server (and caching)
3) Verifying digital signature
4) Verifying hash value of TA binary

Among these, 1) and 2) are executed in the intermediate TA, while 3) and 4) are done in the pseudo TA of GateKeeper. Using a simple TA, whose encrypted version is of size 72 KBytes, we measured the latency of each step. The measurement is taken 10 times for each step, and the average is shown in Table II. Our prototype implementation utilized 2048-bit RSA for signing a TA and AES CBC mode for encryption. Besides, the keys and hash values downloaded from the trusted server are stored in the secure storage provided by the secure OS for offline usage.

TABLE II
Average time taken for each phase of installation

Micro Tasks in Installation Process	Average Processing Time (in seconds)
Download keys and list of hash values from server using TLS	1.13
Write decryption key to secure storage	0.26
Read decryption key from secure storage	0.01
Write RSA public key (modulus and exponent) to secure storage	1.18
Read RSA public key (modulus and exponent) from secure storage	0.02
Write hash list (with 1 hash value) to secure storage	0.80
Read hash list (with 1 hash value) from secure storage	0.02
AES-CBC decryption	0.23
Signature and hash verification by Pseudo TA	1.04

In addition, we took three different TAs of different sizes and compared the time taken for decryption and installation by the pseudo TA. The results are summarized in Fig. 4. The details about the TAs used for the measurement are summarized in Table III.

TABLE III
TAs used for Latency Measurement

Name	Functionality	Size (KByte)
Hello_World	Default TA included in OP-TEE distribution. Increments counter for each invocation.	72
Hash_Gen	Calculates a hash value of a given password using SHA-256.	92
File_Download	Downloads a text file from a server using TLS. (with WolfSSL library)	299

In Fig. 4, we can see that the average time required for decryption and installation increases with the size of the TA. However, the difference in the installation time of 'Hello_World' and 'File_Download' is less than 1 second. We

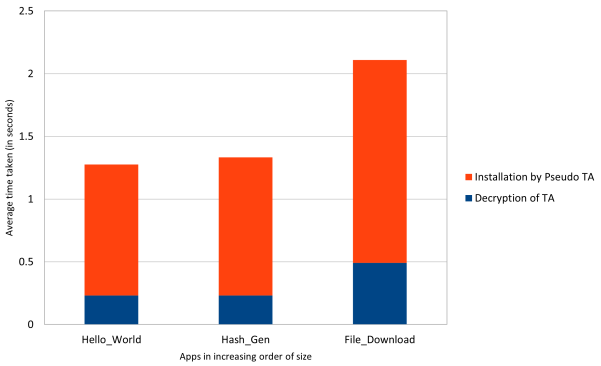

Fig. 4. Latency comparison between TAs of different sizes

also see that the installation latency is roughly linear to the TA size. This shows that the latency using GateKeeper is low enough even for applications of a larger size.

VII. Discussion

Device Authentication by Trusted Server: For the trusted server to authenticate devices, we utilized device ID (UUID of GateKeeper pseudo TA). An alternative (or additional) option is to utilize mutual authentication of TLS using a client certificate and key pair of each device. In this case, a mobile platform vendor is responsible for issuing a certificate and private key upon TEE setup and sharing the client certificate with the system operator. Then the system operator can configure it on the trusted server for device authentication. While we do not see any security drawback with this approach, it is necessary to assume an additional responsibility on the mobile platform vendor. Yet another approach for the device authentication would be a per-user password entered via the proxy app in the normal world. However, this would not only require password registration on the trusted server but also require trusted user interface to protect the user-entered password from a potential adversary in the normal world (e.g., a key logger), which is not feasible or not supported on all platforms.

Encryption of TA: In order to encrypt TA, our design decision is to assign a single symmetric key to each developer. The main reason for this decision is to facilitate key management and distribution of an encrypted TA, allowing the same encrypted TA binary to be processed by all devices that belong to the same system operator. Using different key (either symmetric or asymmetric) for each device would cause significant complexity in distribution of the TA, since a developer needs to prepare a signed, encrypted TA for each device separately.

Integrity Protection of TA: In our design, we utilized digital signature for ensuring integrity. This can be replaced with MAC using the shared secret key for the developer. This way, we could not only simplify the key management (i.e., only one key is used for both encryption and integrity) but also reduce the computational overhead. However, this design decision may result in using the same shared secret key for all devices. (Otherwise, the software developer needs to prepare TA binary for each device, which increases the workload of the system developer.) Moreover, this could increase the risk of key leakage/misuse, similar to a group key setting.

Validation of TA by System Operator: In the proposed framework, we assume that the system operator can verify the integrity and correctness of a TA developed by the authorized developer before adding the hash of the TA to the list of approved TAs. If the verification is insufficient, it would result in approving a TA with a security loophole or backdoor. We admit that such verification is non-trivial. However, this is an orthogonal problem. Given that the source code is made available to the operator for checking and compilation, we assume static and dynamic software analysis can help.

VIII. Related Works

GateKeeper is designed based on GlobalPlatform [2] and OP-TEE secure OS [19], [23] specifications, which define procedure and API to install TAs into the secure world. However, the defined mechanism is not operator-centric. In particular, while a TA to be installed can be signed and optionally encrypted, the keys to be utilized must be embedded in the secure OS. These keys must be set up by a mobile platform vendor, and their update requires involvement of the platform vendor. Therefore, for instance, it does not allow operators to flexibly select and change authorized software vendors. GateKeeper does not intend to re-invent the existing TA installation scheme, but we designed and implemented an additional management layer for operator-centric control.

GateKeeper is most related to secure OS trusted service development. In Truz-Droid [28], the authors developed a trusted app that provides secure services in the secure OS for the apps in the rich OS to use. The secure service provides direct interaction between the end user and the secure OS to ensure that the credentials and the confirmation information entered by the user is secure even in a situation where the rich OS or the underlying hypervisor is compromised. They also incorporate SSL functionality inside the secure OS module to empower the secure OS to securely communicate with a remote server. The notable difference between GateKeeper and Truz-Droid [28] is that GateKeeper provides a framework for the developers to develop the trusted apps and for the users (system operators) to choose whether or which trusted app should be installed onto their mobile devices.

Li et al. [29] proposed a trusted data vault which serves as a small trusted engine that stores and manages security sensitive data in an untrusted mobile OS and device. Similar to Truz-Droid, DroidVault also guarantees that the end user can securely interact with the trusted vault by leveraging TrustZone technology. It also enforces the policy of allowing only the authorized code to access sensitive data. GateKeeper is different from DroidVault in terms of the motivation and purpose of the app. We aim to alleviate the difficulty that arises during development and installation of trusted apps on secure OS while they aim to provide a secure service for storing and managing security sensitive data.

Jang et al. [30] presented a solution to enable developers to utilize TrustZone resources without needing to collaborate with TrustZone OS vendors and manufacturers. They create a private execution environment using a memory region that is isolated from both the rich OS and secure OS. Security critical applications are then moved from rich OS to the private execution environment during runtime. The integrity of the applications is ensured through hash verification before copying them to the private execution environment. The difference between our work and PrivateZone lies in the fact that we aim to provide a framework for TA installation/distribution/update in an operator-centric manner. In contrast PrivateZone focuses on developing a prototype for app developers who can make use of the benefits of TrustZone without involving the OS vendors.

IX. Conclusion

There is an increasing demand to implement robust mobile device management in various working environments, including enterprise IT and critical infrastructure operations. While trusted execution environment (TEE) is a promising building block, its closed nature and lack of flexibility for installing and updating trusted software have limited its broader adoption. To address this challenge, in this paper we presented a framework, called GateKeeper, that enables system operators to choose third-party developers to implement a trusted application and then install it onto mobile devices it manages without involving mobile platform vendors at each iteration. GateKeeper reduces dependency on mobile platform vendors and thus provides system operators with more control. We also developed the proof-of-concept on OP-TEE secure OS for extensive performance and overhead evaluation.

Some research challenges remain for the GateKeeper ecosystem to be more effective. For instance, it is necessary for a system operator to be able to rigorously validate the integrity (e.g., without any malicious code) and correctness of a trusted application developed by the third-party developer. Furthermore, it is an interesting future work to extend GateKeeper framework to allow a single device to be enrolled in multiple system operators, which is often the case in a BYOD context. Lastly, we plan to open-source GateKeeper for evaluation by the TEE user community and mobile platform vendors.

Acknowledgment

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

[1] J. Guilbon, "Introduction to Trusted Execution Environment: ARM's Trustzone," https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html, Jun. 2018, accessed: Feb. 2019.

[2] *TEE System Architecture Version 1.2*, Nov. 2018.

[3] ARM, "Introducing arm trustzone," https://developer.arm.com/technologies/trustzone, accessed: Feb. 2019.

[4] *TEE Protection Profile Version 1.2.1*, https://www.commoncriteriaportal.org/files/ppfiles/PP%20TEE%20v1.2.1_20161215.pdf, GlobalPlatform Device Committee, Nov. 2016, accessed: Feb. 2019.

[5] E. Esiner, D. Mashima, B. Chen, Z. Kalbarczyk, and D. Nicol, "F-pro: a fast and flexible provenance-aware message authentication scheme for smart grid," 10 2019, pp. 1–7.

[6] E. Esiner, U. Tefek, H. S. Erol, D. Mashima, B. Chen, Y.-C. Hu, Z. Kalbarczyk, and D. M. Nicol, "Lomos: Less-online/more-offline signatures for extremely time-critical systems," *IEEE Transactions on Smart Grid*, vol. 13, no. 4, pp. 3214–3226, 2022.

[7] U. Tefek, E. Esiner, D. Mashima, B. Chen, and Y.-C. Hu, "Caching-based multicast message authentication in time-critical industrial control systems," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 1039–1048.

[8] "Trustonic Secured Platforms (TSP), The industry's most complete device security solution," https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp/, accessed: Feb. 2019.

[9] D. Berard, "KINIBI TEE: TRUSTED APPLICATION EXPLOITATION," https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html, accessed: Feb. 2019.

[10] "SAMSUNG Knox," https://www.samsungknox.com/en, accessed: Feb. 2019.

[11] D. Rosenberg, "Qsee trustzone kernel integer over flow vulnerability," in *Black Hat conference*, 2014, p. 26.

[12] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.

[13] D. Rosenberg, "Reflections on Trusting Trustzone," *Black Hat USA*, 2014.

[14] E. Evenchick, "Rustzone: Writing Trusted Applications in Rust," *Black Hat Asia*, 2018.

[15] *EMUI 8.0 Security Technical White Paper*, https://consumer-img.huawei.com/content/dam/huawei-cbg-site/en/mkt/legal/privacy-policy/EMUI%208.0%20Security%20Technology%20White%20Paper.pdf, HUAWEI Technologies CO., LTD., Oct. 2017, accessed: Feb. 2019.

[16] D. Shen, "Exploiting trustzone on android," *Black Hat USA*, 2015.

[17] A. Grygoriev, "Security issues with ARM TrustZone," *TESTING STAGE*, 2018.

[18] Q. Ye, H. C. Tan, D. Mashima, B. Chen, and Z. Kalbarczyk, "Position paper: On using trusted execution environment to secure cots devices for accessing industrial control systems," 2021.

[19] "Open Portable Trusted Execution Environment," https://www.op-tee.org/, accessed: Feb. 2019.

[20] "SierraTEE for ARM®Trustzone®and MIPS," https://www.sierraware.com/open-source-ARM-TrustZone.html, accessed: Feb. 2019.

[21] "ARM1176JZ-S Technical Reference Manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/Chdfjdgi.html, accessed: Mar. 2019.

[22] R. Stajnrod, R. Ben Yehuda, and N. J. Zaidenberg, "Attacking trustzone on devices lacking memory protection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–11, 2021.

[23] "Op-tee documentation," https://optee.readthedocs.io/en/latest/index.html, accessed: Aug. 2022.

[24] "QEMU the FAST! processor emulator," https://www.qemu.org/, accessed: Apr. 2019.

[25] "Qemu v7," https://optee.readthedocs.io/en/latest/building/devices/qemu.html, accessed: Feb. 2019.

[26] "REE-FS TA binary formats," https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html#ree-fs-ta-binary-formats, accessed: Feb. 2019.

[27] "Emedded TLS Library," https://www.wolfssl.com/, accessed: Apr. 2019.

[28] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 14–27.

[29] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena, "Droidvault: A trusted data vault for android devices," in *2014 19th International Conference on Engineering of Complex Computer Systems*. IEEE, 2014, pp. 29–38.

[30] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 797–810, 2018.