TruApp: A TrustZone-based Authenticity Detection Service for Mobile Apps

Sileshi Demesie Yalew[1,2], Pedro Mendonça[1], Gerald Q. Maguire Jr.[2], Seif Haridi[2], Miguel Correia[1]

[1]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
[2]School of Information and Communication Technology, KTH Royal Institute of Technology, Sweden
sdyalew@kth.se, pedromdsmendonca@gmail.com, maguire@kth.se, haridi@kth.se, miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—In less than a decade, mobile apps became an integral part of our lives. In several situations it is important to provide assurance that a mobile app is authentic, i.e., that it is indeed the app produced by a certain company. However, this is challenging, as such apps can be repackaged, the user malicious, or the app tampered with by an attacker. This paper presents the design of TRUAPP, a software authentication service that provides assurance of the authenticity and integrity of apps running on mobile devices. TRUAPP provides such assurance, even if the operating system is compromised, by leveraging the ARM TrustZone hardware security extension. TRUAPP uses a set of techniques (static watermarking, dynamic watermarking, and cryptographic hashes) to verify the integrity of the apps. The service was implemented in a hardware board that emulates a mobile device, which was used to do a thorough experimental evaluation of the service.**

I. INTRODUCTION

In less than a decade, mobile apps became an integral part of our daily lives. In several situations it is important to provide assurance that a mobile app is *authentic*, i.e., that it is indeed the app produced by a certain company and with a certain version. Authenticity is important for many apps provided by service providers, from financial to healthcare organizations, and including public administration. Some of these apps are security-critical, e.g., because they allow access to private information (identity, financial details, health details, etc.).

Android has for some time become the most adopted mobile operating system (OS), with a market share much higher than all its competitors together [1]. However, this OS is prone to security issues. A major source of problems are malicious apps. Some apps, even when downloaded from the official Android market, the Play Store, can be malicious [2], [3] and thus attack other apps by modifying their behavior. Moreover, the open nature of Android has led to the appearance of many alternative app markets [4], which facilitates the distribution of malicious apps. Some of these apps are repackaged apps, i.e., apps that were originally legitimate, but that were unpacked, modified to include a malicious payload, signed again, then placed in a market [2]. Moreover, in some cases it has been possible to inject malicious code into legitimate Android apps without damaging the digital signature of the original app [5].

There are mechanisms in place to avoid these problems. Play Store analyses apps and does not distribute them if it suspects they are malicious, but some malicious apps manage to pass undetected [2], [3]. Repackaging requires reverse engineering, which may be made harder by using code obfuscation [6], but this obfuscation is not entirely effective. This work is not concerned with substituting or improving these mechanisms, but with providing another layer of security.

A. Watermarking

Software watermarking mechanisms allow the verification of the authenticity and the integrity of an app [7]–[10]. These mechanisms can be implemented directly into the app's source code or executable, or can be external to the app (e.g., as a service in the mobile OS). The general idea is to embed a *watermark*, typically numbers or strings, into the target app, in such a way that it does not affect the execution of the app but that it is difficult to remove without detection. Verification consists in extracting and checking the watermark.

Depending on how the watermark is embedded and extracted, watermarking schemes can be classified in two classes: static and dynamic [11], [12]. In *static watermarking*, the watermark is embedded into the code or data of an app and is extracted without executing the app. In contrast, in *dynamic watermarking* extracting the watermark involves executing the app. Adversaries may be able to remove the watermarks by leveraging reverse engineering. It is possible to make these attacks harder but not to avoid them altogether, assuming that the attacker has access to the app's executable code (bytecode, in the Android case). This *malicious host problem* is currently unsolvable, only mitigable [10].

An alternative solution would be to run (part of) the protection mechanism in an external device such as a dongle [13]. A dongle is a small hardware device that is attached to computers using USB ports or other interfaces. Dongles are widely employed to prevent software piracy via apps interacting with the dongle during execution to ensure that the dongle is present. Nevertheless, this mechanism can be defeated by emulating the interaction with the dongle or modifying the app to remove this interaction.

ARM TrustZone [14] is a hardware security extension supported by recent ARM processors, including ARM Cortex A8, A9, and A15. The TrustZone technology provides two isolated execution environments: the *secure world* that runs trusted apps on top of a small trusted OS; and the *normal world* that runs untrusted apps on top of a rich OS, such as Android.

TABLE I
SUMMARY COMPARISON BETWEEN THE THREE TECHNIQUES.

Technique	Protection	Detection	Delay
Measurements	best	best (collision resistance)	worst
Static watermarks	best	high	best
Dynamic watermarks	high	worst	average

The physical core of the processor is divided in two virtual cores, corresponding to these two execution environments. The normal world cannot access the system resources (e.g., memory space and peripherals) that are assigned to the secure world; while the secure world has access to the normal world's resources.

B. TruApp

This paper presents the design and implementation of TRUAPP, a software authenticity and integrity verification service for mobile apps. This service aims to ensure that an app running in a mobile device (e.g., a smartphone or a tablet) is genuine and was not modified in an unauthorized way by a third party or the user. TRUAPP is protected from the mobile OS, apps, and malware by running in the secure world.

The design explores static watermarking, dynamic watermarking, and measurements (i.e., cryptographic hashes over code). An actual implementation of TRUAPP does not have to implement all these mechanisms, only a subset of them. The choice of this subset involves tradeoffs that are summarized in Table I (explained in Section V-C).

We assume that TRUAPP is provided by the *TruApp provider*. Moreover, we assume that an app is provided by an *app vendor*. This vendor may be designated a *service provider*, when the app is an interface for a service, e.g., a home-banking app that provides access to the bank's services. In some cases we use the term *TruApp instance* to clarify that we are talking about TRUAPP running in a particular mobile device.

To perform verification of an app authenticity, the app vendor provides a TRUAPP instance a certificate called a *verification key* (VK). This VK describes the characteristics of the original app and allows TRUAPP to verify if the app is genuine. VK's content depends on the detection mechanism(s) used, e.g., it can be just a hash if only measurements are used. VK is encrypted and signed in order to ensure its authenticity, integrity, and confidentiality.

The *authenticity verification process* works essentially as follows. When the app starts running, it contacts the entity to which it has to prove to be authentic and obtains VK and a nonce (for freshness, i.e., to avoid replay). Then, the app calls the TRUAPP service in the mobile device and passes it VK and the nonce. Next, TRUAPP verifies the signature of VK, extracts the watermarks and/or the measurements, and checks if they correspond to the information in VK. If they do, TRUAPP returns a signed certificate containing the nonce. The app finally sends this certificate to the entity in order to verify that the app is authentic. Further details of the VK and authenticity verification process are given in Section III-C.

C. Contributions

The main contributions of this paper are: (1) the design of a software authenticity and integrity verification service called TRUAPP, protected using the TrustZone extension; (2) an authenticity verification process based on static watermarking, dynamic watermarking, and measurements, plus system integrity verification; (3) an implementation of TRUAPP running on an NXP Semiconductors i.MX53 Quick Start Board (QSB); and (4) an experimental evaluation of TRUAPP.

II. USE CASES

We believe TRUAPP is useful in several use cases. Here we briefly present two examples.

The first use case is related to service providers, such as the above-mentioned bank and its home-banking app. The bank is concerned that the app may be compromised and used to steal confidential data about the user and user's bank account. Therefore, the app is distributed in a reliable market and includes logic for invoking TRUAPP. When the user starts to login via the app, the app's authenticity verification process explained above is executed. If successful, the service provider can trust the app to execute as expected. On the contrary, if the app has been compromised or it is not the original app (e.g., a repackaged version), then access to the service is blocked. Note that the authenticity verification must be successful before the user provides any login or other sensitive information via the app.

The second use case is inspired by a recent project we were involved in. The PCAS project designed a hardware component called a *secure portable device* (SPD) [15], [16]. This component is essentially a mobile device that is highly secure, supports biometric authentication, has a large memory, has the form of a smartphone sleeve, and is supposed to be connected to smartphone (it does not have communications, other than a USB connection to the phone). The objective is to store private data such as its owner's personal health record and medical exams. The SPD limits the data it provides to the smartphone, but a malicious app might obtain some useful information. The PCAS app that runs in the smartphone could be protected using TRUAPP. The app would contact the SPD, receive a nonce and proceed with the authenticity verification process explained above. Notice that, in this case, the resulting authenticity is known to a hardware component connected to the mobile device, rather than to an external service or backend as in the previous use case.

III. SYSTEM ARCHITECTURE AND DESIGN

A. Threat Model and Assumptions

We leverage the ARM TrustZone hardware protection to run most of TRUAPP in the secure world, isolated from the mobile OS. We assume that the software running in the normal world, including the mobile OS, is untrusted, i.e., it may be malicious or compromised by malware or hackers. In contrast, we assume the secure world software, including the TRUAPP software, is trustworthy.

We minimize the software installed in the secure world as much as possible to minimize the size of the trusted computing base (TCB) [17]. Specifically, we use a small tailor-made kernel (Section IV-A), do not install unnecessary libraries, and have no network stack. We also reduce the size of the API to the secure world to reduce the attack surface and carefully validate inputs to that API, to make software attacks against the secure world infeasible; hence we assume they cannot be successful.

A small part of TRUAPP runs in the normal world. The rest of TRUAPP verifies the integrity of the normal world and, specifically, the part of TRUAPP running there, hence we assume neither malware or attackers can compromise it.

We assume the TRUAPP service provider has a public-private key pair $(\mathtt{Ku}_{tasp}, \mathtt{Kr}_{tasp})$ for some public-key cryptographic scheme (e.g., RSA), with a key size considered secure (3072 bits for RSA [18]). That entity keeps the private key $\mathtt{Kr}_{tasp}$ for itself and installs the public key $\mathtt{Ku}_{tasp}$ in the TRUAPP service in the mobile devices. Furthermore, the service provider also generates a public-private key pair $(\mathtt{Ku}_{ta}, \mathtt{Kr}_{ta})$ for each TRUAPP instance running in a mobile device, plus a public key certificate $\mathtt{C}(\mathtt{Ku}_{ta})_{Kr_{tasp}}$ (e.g., in the X.509 format [19]). Both keys and the certificate are stored in TRUAPP, in the secure world of the device. Finally, every app vendor has also a public-private key pair $(\mathtt{Ku}_{av}, \mathtt{Kr}_{av})$.

We assume the existence of a collision-resistant hash function [20], e.g., SHA-2 with 512-bit output [18]. We also assume a secure symmetric encryption system (e.g., AES with 256-bit keys [18]) with cipher block chaining (CBC) mode.

B. Architecture

Figure 1 depicts the architecture of TRUAPP. The mobile OS and the apps it executes, and parts of TRUAPP (i.e., *syscalls tracer* and *TZ Driver*) are run in the normal world. The *syscalls tracer* is a module of TRUAPP that intercepts and logs kernel level system calls made by an app running in the normal world. This is only used in conjunction with the dynamic watermarking scheme. *TZ Driver* is a kernel driver that supports cross-world communication between software in the normal world and TRUAPP components in the secure world. This driver allocates a shared memory buffer that is used for the software in the normal world to pass selected data to TRUAPP in the secure world, and for TRUAPP to return back verification results to the requesting app.

The secure world software architecture is designed to run the TRUAPP modules (*app verifier*, *system verifier*, and *interface*) on top of a small trusted OS that provides basic OS functions (e.g., process management and file access). The *app verifier* module implements a set of techniques to verify the authenticity and integrity of an app running in the normal world. The *system verifier* checks the integrity of the TRUAPP component that runs in the normal world (the *syscalls tracer*) and the Android kernel, as the normal world is vulnerable to attacks. The *interface* module acts as an interface between the normal world and TRUAPP. It is responsible for receiving and replying to requests, e.g., a request from an app in the normal

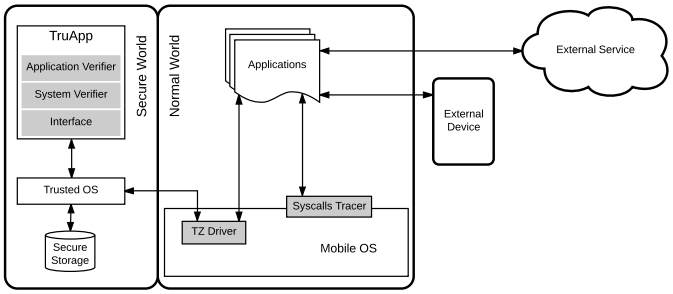

Fig. 1. Architecture of a mobile device running TRUAPP. The grey boxes are components of the TRUAPP service.

world for an authenticity verification. It also validates all the incoming data from the normal world and protects against buffer overflows and other input attacks. In addition to the private memory, a private persistent storage area is reserved in the secure world (shown as *secure storage* in the figure).

C. Authenticity Verification Process

This section describes the authenticity verification process, which mainly involves the *app verifier* module and the *interface* module of Figure 1. In this section we assume the integrity of the *syscalls tracer* module (we defer to Section III-D an explanation on how this assumption is enforced).

1) Verification Key: The authenticity verification process is based on watermarking and measurements. These techniques require information about the app. For this purpose, the app vendor creates a certificate called *verification key* (VK) that contains this information (i.e., hash value, static watermark data, and dynamic watermark data).

VK is *digitally signed*. There are two options for this signature:

1) the signature is issued by the TRUAPP provider using its private key $\mathtt{Kr}_{tasp}$, upon request from the app vendor;
2) the signature is issued by the app vendor using its own private key $\mathtt{Kr}_{av}$. As the private key is not in the mobile device, the app vendor provides also a public key certificate $\mathtt{C}(\mathtt{Ku}_{av})_{Kr_{tasp}}$ signed by the service provider with its private key $\mathtt{Kr}_{tasp}$.

In the first case, the signature is verified in the TRUAPP *interface* module using the public key $\mathtt{Ku}_{tasp}$. In the second, VK comes with the certificate $\mathtt{C}(\mathtt{Ku}_{av})_{Kr_{tasp}}$, the signature of VK is verified using $\mathtt{Ku}_{av}$ from the certificate, and the signature of certificate is verified using $\mathtt{Ku}_{tasp}$.

VK is *encrypted* using hybrid encryption [21]. This scheme consists essentially in obtaining a random secret key $\mathtt{K}_s$ (e.g., a 256-bit key for AES-256), encrypting the content of VK with $\mathtt{K}_s$, and encrypting $\mathtt{K}_s$ with the public key of the TRUAPP instance $\mathtt{Ku}_{ta}$. VK consists of three parts: the encrypted content, the encrypted $\mathtt{K}_s$, and the signature.

When VK is received by TRUAPP in the device, the verification and decryption process consists: verifying the signature as explained above, decrypting $\mathtt{K}_s$, and using this key to decrypt the contents of VK.

This combination of mechanisms ensures the following security properties:

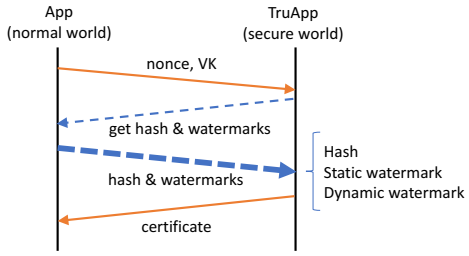

Fig. 2. Authenticity verification scheme.

- *authenticity* – VK must have been created by the TRUAPP service provider or the app vendor, as they are the only entities that have, respectively, $\texttt{Kr}_{tasp}$ and $\texttt{Kr}_{tav}$ to sign the message.
- *integrity* – VK cannot be modified and its signature modified to match its content for the same reason as for authenticity;
- *confidentiality* – the contents of VK cannot be disclosed by entities other than the TRUAPP instance or the entity who generated the VK. As only these entities know the plain text version of the VK or have the private key $\texttt{Kr}_{ta}$ necessary to decrypt the key $\texttt{K}_s$ (that is encrypted with $\texttt{Ku}_{ta}$), and the key $\texttt{K}_s$ is required to decrypt the content of VK.

2) Overview of the Authenticity Verification Process: It is up to the app to show to the external service or external device that it is authentic (Section II). Therefore, the VK is embedded in the app package that is downloaded and installed from the app market. The app typically gets a nonce – a number that is never reused – from the external service/device, which it passes together with VK to the *TZ Driver*, which passes them to the *interface* module within the secure world. This process is equivalent to the app calling a remote procedure

```
cert_t TruApp_verify(nonce_t nonce, vk_t vk);
```

where `nonce_t` is the type of the nonce (e.g., a 64-bit unsigned integer), `vk_t` the type of VK, and `cert_t` the type of the certificate returned by the function showing that the verification was successful (otherwise it returns `null`). This certificate is signed using the private key of the TRUAPP instance ($\texttt{Kr}_{ta}$).

Figure 2 presents this process but shows also that, between the function call and the result being returned, TRUAPP interacts with the app. Specifically it obtains measurements and/or watermarks, which it uses to evaluate the authenticity of the app.

The *app verifier* module is composed of three sub-modules, corresponding to the three schemes for verifying authenticity: *measurement checker*, *static watermarker*, and *dynamic watermarker*.

3) Measurement Checker: The *measurement checker* module calculates the hash of the app using a collision-resistant hash function and validates it against the hash value presented in the corresponding VK, in order to verify that the app was not modified. A small change to the app results in a different hash value due to the collision resistance property, so if the new hash value does not match the hash value in the VK, the app (or VK) must have been modified.

This verification is possible because the secure world can access the resources of the normal world. Android apps are distributed in the *Android app package* (APK) format. In this mechanism, the secure world inspects the app's APK file stored typically in internal memory (in the part assigned to the normal world).

4) Static Watermarker: For static watermarking, we use a scheme in which a watermark is represented by the values of particular bytes in the app bytecode, instead of the usual method of having the value of the watermark stored in a variable (e.g., a string or an integer) of the app source code. The app developer can select which bytes to use and store their positions, i.e., an index to locate a byte, and value of each selected byte in the VK of the app. The higher the number of bytes used and the more scattered they are, the lower the chances of having two apps with the same values for all of these bytes. The *static watermarker* module running in the secure world reads the position of each byte from the VK and compares its corresponding value with the value of the byte at that position in the target app's bytecode. If any of the bytes do not match, then the app is not authentic.

This checking is possible because the secure world can access all the resources of the normal world. Again, the secure world inspects the app's APK file.

5) Dynamic Watermarker: Android is based on Linux, so just as Linux it provides system calls (syscalls) that allow apps in user mode to perform various low level system operations, for example, with files (e.g., `read`, `write`), processes (e.g., `fork`, `exec`), and networks (e.g., `socket`, `connet`). The syscalls made by an app provide relevant information about the runtime behavior of the app [22].

We propose to use *syscall traces* (sequences of syscalls) of the app as dynamic watermarks. The *syscalls tracer* module running in the normal world intercepts and logs syscalls made by the target app, using itself the `ptrace` syscall. When a certain time elapse or number of syscalls has been made (a configuration parameter), the syscalls tracer sends the trace – the sequence of syscalls – to the secure world via *TZ Driver*. We designate this trace $\texttt{T}_{run}$.

The *dynamic watermarker* module running in the secure world does the verification essentially by comparing $\texttt{T}_{run}$ with a *reference trace*, obtained with the genuine app, that comes in VK: $\texttt{T}_{ref}$. This cannot be a trivial comparison of two call sequences because two executions of the same app normally do not produce the same trace, as the exact order of syscalls depends on timing and interaction with other components.

To take these effects into consideration, we do a similarity comparison using the Needleman-Wunsch algorithm [23]. This algorithm was designed for finding similarities in the amino-acid sequences of two proteins, not traces. The algorithm essentially compares sequences of letters: it adds points when a match is found, subtracts points when a gap is found (i.e., a match that requires discarding letters in one of the sequences), and subtracts more points when a mismatch is found. The

algorithm solves the dynamic programming problem of finding and evaluating the best match between two sequences.

In TRUAPP, every syscall in $\mathtt{T}_{run}$ and $\mathtt{T}_{ref}$ is converted to a letter using some deterministic criteria. Next, the algorithm configured with appropriate values is used to compute their similarity (in the experiments, a match had 4 points, a gap -1, and a mismatch -2). Finally, a threshold $\mathtt{Tresh}_{\neq}$ is used to decide if the apps that produced the traces were the same (result above or equal to $\mathtt{Tresh}_{\neq}$) or not (result below).

D. Normal World Integrity Verification

In Section III-C we assumed the integrity of the *syscalls tracer* module, although this module is executed in the normal world. In this section we explain how this assumption is enforced. This enforcement involves three aspects – trusted boot, system integrity verification, and tracer integrity verification – that are implemented by the TRUAPP *system verifier* module (Figure 1). This module is further divided into three submodules: *boot support*, *system verifier*, and *tracer checker*.

The normal world integrity verification is made at two moments. First, when the device is started a *trusted boot* is executed. Second, when an authenticity verification is requested (Section III-C), first a system integrity verification is executed, then a tracer integrity verification is done. If any of these two verifications fails, then the authenticity verification terminates immediately with failure (i.e., returns `null`).

1) Trusted Boot: The boot of a system involves executing a sequence of modules, typically starting at the BIOS, then a bootloader, the OS kernel, etc. In a *trusted boot* process, the first module executed (e.g., the BIOS) is designated *static root of trust for measurement* (SRTM), as the integrity of the whole process depends on the trust we can have in that component [24]. The process consists in each module, starting with the SRTM, measuring the next one and storing this measurement (a hash obtained using a collision-resistant hash function) in a safe place. The best-known implementation of the concept uses the Trusted Platform Module (TPM) [25], a hardware board currently available on the motherboard of many commodity PCs. The TPM contains an array of Platform Configuration Registers (PCRs) where the measurements are stored (e.g., one measurement per PCR).

In our system we implement this basic idea. We assume that when the device boots, the secure world is booted first, then it passes control to the normal world. This is the most common configuration of devices that have TrustZone, such as i.MX53. Therefore, in our case, the secure world is the SRTM, so it computes a hash over the normal world. The module in charge of obtaining this hash is the *boot* module. The boot module boots the Android kernel and computes its hash. When the Android kernel starts to run, it does a measurement of the *init* program, in charge of initializing several elements of Android, and passes this hash to the secure world boot module. To do so it contacts the *TZ Driver* in a manner similar to calling a remote procedure

```
TruApp_store_measurement(hash_t hash);
```

Then, *init* measures the program *app_process*, which when executed becomes the *zygote* process, i.e., the first instance of the Dalvik VM (the VM that executes all Android apps). Again, *init* calls the same function to pass the measurement to the boot module. The secure world does not have PCRs as it is not a TPM, but instead it stores the hashes in a vector that plays a similar role to the array of PCRs. Notice that the function does not take any input other than the hash; the hashes are simply stored by the boot module in the order it obtains them.

This process allows later verification of whether the normal world has been compromised.

2) System Integrity Verification: The *system verifier* module detects whether the normal world has been compromised. The module does measurements of the components measured during the trusted boot process: kernel, *init*, and *app_process/zygote*. Then, it compares these hashes with those stored during the boot process. If they differ, then the system must have been compromised due to the collision resistance property of the hash function, hence the verification fails. Otherwise, the verification passes.

3) Tracer Integrity Verification: The *tracer checker* is responsible for verifying the integrity of the code of the *syscall tracer* module to validate the syscall traces collected by the module. When it is called, it calculates a hash $\mathtt{h}$ of the *syscall tracer* module. Then, it compares this hash with the hash of the module it keeps in secure storage ($\mathtt{h}_{st}$). The value of $\mathtt{h}_{st}$ comes with TRUAPP, it is not obtained during the trusted boot, as it is not part of the boot process.

The *syscall tracer* might be maliciously modified just after the tracer integrity verification. However, the system integrity verification provides assurance that the software running in kernel mode (the kernel itself) is not compromised. However, such a modification would still be possible if there was a vulnerability in the kernel or the *syscall tracer* itself. A solution would be to repeat the verification of the *syscall tracer* multiple times during the capture of a trace; however, this scheme might be attacked using a race [26], but the probability would be much lower. A protection that makes such an attack almost impossible is for the external service or device to accept only a limited number of authenticity verification attempts for each mobile device.

IV. IMPLEMENTATION

This section describes the implementation of a prototype of TRUAPP. Most of the implementation is independent of the device, but a few aspects may depend on the specific hardware we used: an i.MX53 QSB board equipped with a Cortex-A8 single core 1 GHz processor, 1 GB DDR memory, and a 4GB MicroSD card.

A. Runtime Environment

The kernel used in the secure world is based on Genode, a framework of components to implement small OS kernels. In the secure world we use a small kernel based on a custom kernel (*base-hw*) provided by the Genome project for our

board. This kernel provides the *tz_vmm* driver to support calls from the normal world to the secure world.

In the normal world, we installed a version of Android for the i.MX53 series from Adeneo/Freescale [27]. The Android kernel is patched to be executed in the normal world. This kernel has to be modified to obtain the measurements and send them to the secure world. Our current implementation still does not support this feature, but it could be implemented following a similar mechanism implemented in the Linux kernel that sends hashes to the TPM (e.g., the TrustedGrub bootloader).

In order to store sensitive data in a persistent way (e.g., VKs from different apps), a part of the SD card or internal memory has to be accessible exclusively by the secure world (the *secure storage* in Figure 1). We use the Genode partition manager (*part_blk*) for this purpose. It supports partition tables such as MSDOS and GPT, and provides a block session for each partition on a SD card. This allows the partitions to be addressable as separate block sessions and makes it is easy to grant or deny access to them.

B. TruApp Components

As shown in Figure 1, TRUAPP has several components, both in the normal and the secure world. In this section, we describe the implementation of these components.

We used *strace* to implement the *syscalls tracer* in the normal world. *strace* is a debugging tool for Linux that can be used to trace the syscalls made by a process. It relies on *ptrace* syscall and can be consider as an interface between the user space and *ptrace* syscall. The *syscalls tracer* module records only the name of each system call. The above-mentioned data is stored in a data structure to be processed and analyzed by the *app verifier* module in the secure world.

In the secure world, we implemented the *app verifier*, *system verifier*, and *interface* modules based on a user level VMM app called *tz_vmm* that runs on top of the Genode kernel. The TrustZone configuration within Genode partitions the DDR RAM between the secure world and *tz_vmm* is able to request the normal world's RAM via an IOMEM session during its start-up routine. The memory is mapped as uncached to the secure worlds address space, thus the whole normal world memory can be accessed by the *system verifier* module running in the secure world. We also configured the Android file system partitions to be accessed by the secure world, so that the *app verifier* module can access the app's APK in the normal world to verify its integrity and authenticity. The implementation of the Needleman-Wunsch algorithm was based on the *seq-align* library [28].

V. EXPERIMENTAL EVALUATION

A. Authenticity Verification

We conducted experiments to evaluate the detection performance of the three authenticity verification techniques individually (measurements, static watermarks, and dynamic watermarks). These experiments assess the ability of TRUAPP to play its role and enables app vendors to choose which verification techniques to use. In the experiments, every non-genuine app provided TRUAPP with the VK of the corresponding genuine app, in attempt to be considered genuine by TRUAPP; otherwise these apps could not possibly show to an external service or device that they were genuine.

We started by using TRUAPP to compare if pairs of apps could be taken to be the same. For that purpose we used a set $\mathcal{A}$ of around 20 random apps downloaded from the Play Store. For every app A in $\mathcal{A}$ with verification key VK_A, we used TRUAPP to verify if all other apps A' in $\mathcal{A}$ could be taken to be A, by providing VK_A as the verification key. In all cases all the three techniques said the app was not genuine, as expected, as the apps were different.

1) Datasets: After that initial phase, we did experiments with apps that were similar, i.e., we compared if a repackaged app A' could be taken for the genuine app A. We used two datasets for that purpose: (1) *manually repackaged apps* and (2) *real repackaged apps*.

For creating dataset (1), we downloaded 41 legitimate Android apps from Google Play Store and repackaged them ourselves. The repackagement involved the following steps: (i) unpack the APK file using the Apktool [29]; (ii) convert the bytecode (DEX file) to Smali code (human-readable bytecode) using the same tool; (iii) add to the Smali code a simple malicious code snippet that deletes the user's contacts (taken from [30]); (iv) modify the file `manifest.xml` to give the app more permissions (in this case to read and write the contacts) and to trigger the code when the mobile device finishes booting; (v) repack the app with the Apktool; (vi) sign the APK file and add a self-signed certificate.

For dataset (2), we had first to find repackaged apps and the corresponding genuine apps (from Play Store), which was not simple. We found 20 pairs of apps in such conditions (Table II). The repackaged apps were mostly *mod games*, i.e., games modified to give the player some kind of advantage (e.g., unlimited gems in Clash of Clans).

2) Verification: Both *measurements* and *static watermarks* always detected that the repackaged apps were not genuine. In the case of measurements, this happened because repackaging modifies the app bytecode, leading to a different hash value. In the case of static watermarks, the cause was the fact that the repackaging introduces the code snippet and creates a shift of the bytes past the position where the snippet is added, so the original byte values are not found in the expected position. Therefore, we concluded that for evaluating verification the important case are dynamic watermarks.

To measure the detection performance of *dynamic watermarks*, we need to obtain execution traces, which involves executing all apps with the same sequence of inputs. For this purpose, we ran and traced the apps using the Monkey application exerciser [31] to generate a stream of user events such as clicks, key presses and screen touches. If Monkey is re-run with the same seed value, it can generate the same sequence of events. Due to the complexity of using Monkey and executing these experiments in the board, we run the apps in Google's Android emulator [32], setting it to emulate an

TABLE II
DETECTION RATE FOR DATASET (2), REAL REPACKAGED APPS.

APKs \ $\texttt{Tresh}_{\neq}$	1700	1750	1800	1850	1900
Angry Birds 7.5	0	0	0	0	0
Bomb Squad Pro 1.4.121	0.95	1	1	1	1
CCLeaner 1.19.76	0	0	0	0	0
Clash of Clans 9.24.15	0	0	0.05	0.05	0.05
Clash Royale 1.9.2	0	0	0	0	0
Crossy Road 2.4.4	0	0	0	0	0
FIFA Mobile Soccer 6.1.1	0.1	0.1	0.15	0.2	0.2
Flags Quiz 2.4	1	1	1	1	1
Flick Kick FootballLegends 1.9.85	0	0	0	0	0
Last Day on Earth 1.5.6	0	0	0	0	0
Last Hope TD 3.31	0.25	0.25	0.25	0.25	0.25
Magikarp Jump 1.1.0	0	0	0.1	0.1	0.1
Mo n Ki World Dash 1.6	0.15	0.15	0.2	0.2	0.35
Once Upon a Tower 3	1	1	1	1	1
Realm Defense 1.8.4	0.05	0.05	0.1	0.1	0.1
Sniper 3D Assassin 2.0.2	0	0	0	0	0
Super Mario Run 2.1.1	0	0	0	0	0.1
Zombie Castaway 2.8.1	0	0	0	0	0.05
8 Ball Pool 3.10.3	0	0	0	0	0.1
8 Ball Pool 3.10.1	0	0	0	0	0

ARM CPU. We ran and collected 30 traces for each of the manually repackaged apps and 20 traces for each of the real.

We considered different threshold values for the Needleman-Wunsch algorithm to decide whether the traces are the same or not. For each threshold value, we measured the *detection rate*, which is the ratio of non-genuine applications that are detected to be so. For dataset (1), the detection rate was 0, i.e., the dynamic watermarking technique failed detecting non-genuine applications. The reason for this is that the malicious code snippet runs only after a reboot, so it was never executed and the repackaged applications were confused with the genuine ones. We did the same experiment for dataset (2). The results of these experiments are shown in the Table II. Again, the results were quite bad, with values near 0. The reason for this is that these repackaged applications have a behavior that is very similar to the genuine application.

For dataset (1), we also compared the 30 traces of each app with the reference trace of each of the other apps. We measured six common performance metrics for different threshold values. Consider that: a positive (P) is an application detected by TRUAPP to be non-genuine; a negative (N) is an application classified as genuine; a wrong positive or negative are denominated respectively false positive (FP) and false negative (FN); the opposite are a true positive (TP) and a true negative (TN). The definitions of the metrics used are:

$Accuracy = (TP + TN)/(TP + TN + FP + FN)$
$False\ Positive\ Rate\ (FRP) = FP/(FP + TN)$
$False\ Negative\ Rate\ (FNR) = FN/(FN + TP)$
$Recall = True\ Positive\ Rate\ (TPR) = TP/(TP + FN)$
$Precision = TP/(TP + FP)$
$Fmeasure = 2 \times Recall \times Precision/(Recall + Precision)$

The results of these experiments are shown in Table III. The best performance depends on the metric considered, e.g., it was for $\texttt{Tresh}_{\neq}$ = 1830 for the Fmeasure metric. Notice that the accuracy does not vary much with $\texttt{Tresh}_{\neq}$, wereas recall goes down and precision up.

Table IV summarizes the values of the metrics for the three techniques. For dynamic watermarks we consider the values of

TABLE III
EVALUATION OF DYNAMIC WATERMARKING.

$\texttt{Tresh}_{\neq}$	Accuracy	FPR	FNR	Recall	Precision	Fmeasure
1750	0.977	0.019	0.051	0.949	0.894	0.921
1770	0.975	0.017	0.061	0.939	0.902	0.920
1790	0.977	0.015	0.071	0.929	0.910	0.919
1810	0.978	0.012	0.083	0.918	0.928	0.913
1830	0.980	0.009	0.092	0.908	0.947	0.927
1850	0.981	0.003	0.112	0.888	0.978	0.930

TABLE IV
COMPARISON OF THE THREE TECHNIQUES.

Technique	Accur.	FPR	FNR	Recall	Prec.	Fmeas.
Measurements	1	0	0	1	1	1
Static watermarks	1	0	0	1	1	1
Dynamic watermarks (Table II)	–	–	1	0	–	–
Dynamic watermarks (Table III)	.980	.009	.092	.908	.947	.927

Tables II (the worse case; some metrics are not filled as there is no value for TN) and III (for $\texttt{Tresh}_{\neq}$ = 1830). The table shows essentially that measurements and static watermarks provide better detection results than dynamic watermarks.

B. Performance Overhead

We also evaluated the performance overhead incurred by the authenticity verification techniques to study which technique is best in this aspect.

In order to evaluate the overhead of the measurements technique, we evaluated the time for the *measurement checker* module to calculate the hash of the app using SHA-512 and compare it against the hash value present in VK. We repeated this experiment for different sizes of APK files. The results in Table V show that the time (t) grows linearly with the file size (s). The trend observed is $t = 0.9388 \times s - 0.0562$.

We also evaluated the overhead of static watermarking. The *static watermarker* module running in the secure world reads the position of each byte listed in the *VK* and compares its corresponding value against the value of byte in that position in the target app bytecode. Therefore, we measured the time (t) to perform these operations. Since this time depends on the number of bytes (n) used as watermark, we repeated this experiment for different numbers of bytes (Table VI). The trend is $t = 3.0056 \times n - 90.24$.

For dynamic watermarks, we did not measure the time of the whole detection process as the time to extract the traces is configurable. We measured the total time required for the

TABLE V
TIME TO DO MEASUREMENTS.

Size (MBytes)	Time (ms)
3.3	3,090
4.9	4,580
13.3	12,430
17.5	16,350
18.6	17,370
25.6	23,970
28.3	26,510
37.7	35,160
59.0	55,540
91.5	85,790

TABLE VI
TIME TO DO STATIC WATERMARKING.

No. Bytes	Time (ms)
4	66
8	68
16	72
32	81
64	97
128	130
256	196
512	1,556
1024	3,059
2048	6,071

TABLE VII
TIME TO DO TRACE CONVERSION AND COMPARISON.

No. Letters	Conversion (ms)	Comparison (ms)	Total (ms)
200	96.97	107.43	204.4
400	114.68	108.72	223.4
600	132.53	188.42	321.0
800	159.73	312.63	472.4
1000	168.89	415.21	584.1

syscall tracer module in the normal world to transfer the trace data (syscalls traces) to the *interface module* in the secure world. This time includes the performance delay introduced by context switching between the two worlds, copying the trace data into the shared buffer, and sending it to the secure world. For this, we measured the time for copying different sized chunks of data into the shared buffer and sending them into the secure world. The average throughput to perform the above operations is 17.51 MB/s. We also measured the time for the dynamic watermarker to convert a syscall trace into a sequence of an alphabetical letter and to compare the traces to detect if they are same or not. We repeated this experiment for different number of letters (l) in the converted sequence of letters (200. 400, 600, 800, and 1000). The results are in Table VIII. For each number of events, the total time (t) to complete the above operations is shown in the last column of the table. The trend is $t = 0.5042 \times l + 58.534$.

Finally, we evaluated the performance overhead incurred by the normal world integrity verification process. The *system verifier* checks the integrity of the normal world by calculating hashes of the Android kernel, init, and app_process using SHA-512, and comparing them against their known-good values. The *tracer checker* also does the same operations for the *syscall tracer* module running in the normal world to verify its integrity. Therefore, we measured the time to perform those operations (Table VII). The results are the average of 1000 repetitions. The table shows both the size and the time to check the integrity of the modules. The last line shows the total for the two values. The total time required to check the integrity of the normal world is around 2 seconds in our board.

C. Tradeoffs on Detection Techniques

Table I presents a comparison of the three techniques. In relation to protection from the normal world (second column), it is now clear that the measurements and static watermarks techniques are executed only in the secure world, so they have the best protection from the normal world. On the contrary, the dynamic watermarks technique requires running a module in the normal world (*syscalls tracer*), so its degree of protection is lower, although still high due to the use of the integrity verification mechanisms.

TABLE VIII
TIME TO DO INTEGRITY VERIFICATION.

File name	Size (Kbytes)	Time (ms)
`app_process`	5.7	7.48
`init`	90.1	48.71
syscalls tracer module	1126	829.55
Android kernel	8324	932.66
Total	*9545.8*	*1818.4*

In relation to their ability to detect if an app is not authentic (third column), measurements are clearly the best technique because they leverage the collision resistance property of hash functions. For the other two, their capacity to detect if an app is not authentic depends on the modifications made to the app, modifying, respectively, the bytes checked with the static watermarks, or the syscall sequence for the dynamic watermarks. Our experimental results show that this is high for static watermarks, but lower for dynamic watermarks.

In relation to the time to execute the technique (third column), the highest overhead is from obtaining cryptographic hashes, so measurements tend to be the worst, then dynamic watermarks. Static watermarks are lightweight.

This discussion and the table show clearly that there are tradeoffs that the designer of a TRUAPP implementation can explore, allowing the optimization of different metrics.

VI. RELATED WORK

In the context of mobile devices, Jang et al. [33] proposed a static watermarking mechanism based on steganography techniques. It embeds watermarks into Android apps by reordering the sequence of instruction in the basic blocks of the app DEX files. AppMark [34] and Droidmarking [11] are examples of dynamic watermarking mechanisms that embed a watermark generator into Android apps that are guaranteed to be executed and dynamically create a watermark instance at runtime. The watermark code is combined with the original app with strong data dependency in such a way that it is difficult for attackers to identify and tamper the code by analyzing data dependency AppInk [12] has adopted traditional graph-based dynamic watermarking, implementing it on Android apps. However, adversaries can remove the watermark code embedded in the target app using reverse engineering tools or disable the protection implemented external to the app (e.g., in the mobile OS). In contrast, TRUAPP runs the protection or the watermark code inside the secure world, isolated from the mobile OS, apps and malware by leveraging the TrustZone.

Recent research has begun to use the ARM TrustZone security extension to provide security guarantees in mobile devices despite the mobile OS being compromised. Several works have designed framework to separate small components that do security sensitive computations from the apps and the OS, by running these components in the secure world [16], [35]. Another line of research provides secure storage that is

only accessible to the secure world to protect sensitive data such as private keys [36]. ARM TrustZone has been recently used to implement watermarking [37], [38]. TrustICE [37] uses TrustZone-based watermarking to dynamically protect memory regions in the normal world. The other work uses watermarking for video, not software. We use ARM TrustZone in a very different way: for assessing the authenticity and integrity of apps and keeping the assessment mechanisms isolated from Android and its apps.

VII. Conclusion

We present TruApp, a software authenticity and integrity verification service for mobile devices. TruApp is protected using ARM TrustZone. It is executed mostly on the secure world, which also verifies the integrity of part of the normal and of the TruApp code that runs in that environment. TruApp uses watermarking and measurements to assess the authenticity of mobile apps. We present an implementation of TruApp for the i.MX53 QSB and an experimental evaluation.

Acknowledgements This work was supported by the European Commission through the Erasmus Mundus Doctorate Programme under Grant Agreement No. 2012-0030 (EMJD-DC) and project H2020-653884 (SafeCloud), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

References

[1] IDC, "Smartphone OS market share, 2016 Q3," http://www.idc.com/promo/smartphone-market-share/os, 2016.

[2] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[3] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012, pp. 5–8.

[4] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. L. Spina, and E. Moser, "FSquaDRA: Fast detection of repackaged applications," in *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*, 2014, pp. 130–145.

[5] SecurityLedger, "Exploit code released for Android security hole," https://securityledger.com/2013/07/exploit-code-released-for-android-security-hole/.

[6] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *ACM Workshop on Digital Rights Management*, 2001, pp. 141–159.

[7] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, pp. 311–324.

[8] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *International Workshop on Information Hiding*, 2001, pp. 157–168.

[9] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *Computer*, vol. 36, no. 7, pp. 64–71, 2003.

[10] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[11] C. Ren, K. Chen, and P. Liu, "Droidmarking: resilient software watermarking for impeding Android application repackaging," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 635–646.

[12] W. Zhou, X. Zhang, and X. Jiang, "AppInk: watermarking Android apps for repackaging deterrence," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013, pp. 1–12.

[13] U. Piazzalunga, P. Salvaneschi, F. Balducci, P. Jacomuzzi, and C. Moroncelli, "Security strength measurement for dongle-protected software," *IEEE Security & Privacy*, vol. 5, no. 6, pp. 32–40, 2007.

[14] ARM, "ARM security technology, building a secure system using TrustZone technology," http://www.arm.com, 2009.

[15] PCAS, "PCAS project," https://www.pcas-project.eu/.

[16] S. D. Yalew, G. Q. Maguire, and M. Correia, "Light-SPD: a platform to prototype secure mobile applications," in *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*, 2016, pp. 11–20.

[17] National Computer Security Center, "Trusted computer systems evaluation criteria," Aug. 1983.

[18] ENISA, "Algorithms, key size and parameters report – 2014," Nov. 2014.

[19] "International Telecommunication Union. ITU-T Recommendation X.509: The Directory: Public-Key and Attribute Certificate Frameworks," 2000.

[20] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[21] R. Cramer and V. Shoup, "Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack," *SIAM Journal on Computing*, vol. 33, no. 1, pp. 167–226, 2003.

[22] S. D. Yalew, G. McGuire, S. Haridi, and M. Correia, "T2Droid: A TrustZone-based dynamic analyser for Android applications," in *Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Aug. 2017, pp. 25–36.

[23] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[24] B. Parno, J. M. McCune, and A. Perrig, *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[25] Trusted Computing Group, "TPM main specification level 2 version 1.2, revision 116," 2011, https://trustedcomputinggroup.org/tpm-main-specification/.

[26] M. Bishop and M. Dilger, "Checking for race conditions in file access," *Computing Systems*, vol. 9, no. 2, pp. 131–152, 1996.

[27] Witekio, "NXP i.MX 53 reference BSP," http://witekio.com/cpu/i-mx-53/.

[28] I. Turner, "seq-align: Smith-Waterman & Needleman-Wunsch alignment in C," https://github.com/noporpoise/seq-align.

[29] C. Tumbleson and R. Wisniewski, "Apktool," https://ibotpeaches.github.io/Apktool/.

[30] W. Du, "SEEDlabs: Android repackaging attack lab," http://www.cis.syr.edu/~wedu/seed/Labs_Android5.1/Android_Repackaging/.

[31] Monkey, https://developer.android.com/studio/test/monkey.html.

[32] Android Developers, "Run apps on the Android emulator," https://developer.android.com/studio/run/emulator.html.

[33] J. Jang, H. Ji, J. Hong, J. Jung, D. Kim, and S. K. Jung, "Protecting Android applications with steganography-based software watermarking," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1657–1658.

[34] Y. Zhang and K. Chen, "Appmark: A picture-based watermark for Android apps," in *Proceedings of the 8th IEEE International Conference on Software Security and Reliability*, 2014, pp. 58–67.

[35] M. Pirker and D. Slamanig, "A framework for privacy-preserving mobile payment on security enhanced arm trustzone platforms," in *Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2012, pp. 1155–1160.

[36] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, 2009, pp. 104–115.

[37] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: hardware-assisted isolated computing environments on mobile devices," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 367–378.

[38] P. A. del Pino, A. Monsifrot, C. Salmon-Legagneur, and G. Doërr, "Secure video player for mobile devices integrating a watermarking-based tracing mechanism," in *Proceedings of the 11th IEEE International Conference on Security and Cryptography*, 2014, pp. 1–8.