DroidPosture: A Trusted Posture Assessment Service for Mobile Devices

Sileshi Demesie Yalew[1,2], 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, maguire@kth.se, haridi@kth.se, miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—Mobile devices such as smartphones are becoming the majority among computing devices. Currently, millions of persons use such devices to store and process personal data. Unfortunately, smartphones running Android are increasingly being targeted by hackers and infected with malware. Anti-malware software is being used to address this situation, but it may be subverted by the same malware it aims to detect.**

We present DROIDPOSTURE, a posture assessment service for Android devices. This service aims to securely evaluate the level of trust we can have on a device (assess its posture) even if the mobile OS is compromised. For that to be possible, DROIDPOSTURE is protected using TrustZone, a security extension for ARM processors. DROIDPOSTURE is configurable with a set of application and kernel analysis mechanisms that enable detecting malicious applications and rootkits. We implemented a DROIDPOSTURE prototype using a hardware board with an ARM processor with TrustZone, and evaluated its performance and security.

I. INTRODUCTION

Mobile devices have gradually become an integral part of our daily lives. Currently, financial and healthcare institutions offer services to their clients using smartphone applications. Although this is convenient, it also leads people to rely on these applications to access and process privacy-sensitive data, e.g., financial data and medical records. Many of these applications run on Android, the most adopted mobile operating system (OS) today [1]. However, the popularity of Android and the open nature of its application marketplaces make it a prime target for malware [2]–[5]. This situation puts data stored in smartphones in jeopardy, as it can be stealthily stolen or modified by malware that infects the device.

Several companies provide anti-malware software for mobile OSs. Moreover, the research community has investigated mechanisms for detecting malware on Android using static and dynamic analysis [6], [7]. However, these tools and mechanisms run in the device and assume that the mobile OS is trusted, i.e., that it is part of the trusted computing base (TCB) [8]. However, current malware often disables anti-malware software when it infects a device or computer. For mobile phones this trend started more than a decade ago with malware such as the Metal Gear Trojan and Cabir.M [9] and continues, e.g., with HijackRAT [10].

Introspection has been proposed in the context of PC virtualization with hypervisors as a solution to protect anti-malware and intrusion detection mechanisms from malware and hackers [11]–[13]. The idea is to run these mechanisms in a thin virtual machine (VM) isolated from the fat VM that runs the OS and the applications. Yan and Yin applied this approach to detect malware in Android smartphones using a customized QEMU hypervisor [14]. However, the size and complexity of hypervisors still make them a target for malware, similarly to what happens with OSs.

TrustZone is a hardware security extension incorporated into recent ARM processors [15]. This extension partitions the system resources (e.g., memory, peripherals, etc.) in two logical parts: the secure world and the normal world. The *secure world* runs trusted applications on top of a small trusted OS, whereas the *normal world* runs the normal applications on top of a normal OS such as Android. TrustZone protects the secure world resources from the normal world, whereas the secure world can access resources of the normal world. This hardware separation protects the confidentiality and integrity of computation and data in the secure world, while permitting the secure world to inspect the normal world.

Most uses of TrustZone in the literature are based on *measurements* of the normal world, i.e., on hashes of the software running in the normal world obtained using a collision-resistant hash function. For example, some mechanisms provide login data [16] and sensor readings [17] together with measurements of the normal world. Software in the normal world obtains the login or sensor data, then calls the secure world to get the measurements and a signature. The recipient of this information can check if the normal world is in a trusted state by checking if the values of the hashes correspond to trusted configurations. This is a direct extension of remote attestation mechanisms that have been proposed earlier for the Trusted Platform Module, a simpler hardware module [18].

This way of using TrustZone is interesting, but the versatility of TrustZone suggests it is possible to obtain richer information about the normal world than just hashes, which are simply numbers with limited semantics. An approach is to analyze the *posture* or *compliance* of the device. The notion of posture assessment was introduced in RFC 5209 [19] for *network access control* [20], which proposed having a software agent running on endpoint devices (such as laptops and desktops) to evaluate and report the posture/compliance of the device to the network owners (e.g., anti-virus software running on the device or not, updates installed or not). The

network owner has validation software that determines the device's compliance with the security policies, allowing it to connect to the company's network, to block it, or to connect it to some lower trust VLAN (e.g., one that connects only to the Internet).

This paper presents the design and implementation of DROIDPOSTURE, a *posture assessment service* for mobile devices. DROIDPOSTURE runs in the devices (e.g., smartphones) and evaluates the security status of their OS (e.g., Android) and applications. DROIDPOSTURE is protected from the OS, applications, and malware by leveraging the TrustZone extension and running in the secure world. DROIDPOSTURE does introspection of the normal world and can be configured with a variety of assessment mechanisms, e.g., static analysis of applications and detection of rootkits. Posture data obtained with DROIDPOSTURE can be sent to *external service providers* like the above-mentioned financial and healthcare institutions, which use it to decide if they provide their service to the device or not (or in what conditions they provide it). We also present a communication protocol for this purpose.

In this paper we propose two classes of assessment mechanisms – application and kernel analysis mechanisms – and provide two example of each: signature-based detector, learning-based detector, syscall table checker, and kernel integrity checker. These mechanisms illustrate the forms of posture analysis that can be implemented in DROIDPOSTURE, although others may also be used.

The main contributions of this paper are: (1) the design of DROIDPOSTURE, a posture assessment service for mobile devices that is protected using the TrustZone extension and that is configurable with a set of application and kernel analysis mechanisms; (2) an implementation of DROIDPOSTURE for Android and in hardware, specifically on the NXP Semiconductors i.MX53 Quick Start Board (QSB); (3) an experimental evaluation of DROIDPOSTURE.

II. BACKGROUND

This section provides background on the technologies underlying the design and implementation of DROIDPOSTURE.

A. ARM TrustZone

ARM TrustZone [15] is a security extension supported by recent ARM processors, e.g., ARM Cortex-A and Cortex-M. It provides two logically isolated execution domains: the secure world for security-sensitive computation and storage, and the normal world for conventional processing. A program running in the normal world can make the processor switch to the secure world using a special *secure monitor call* (SMC). This technology is not yet widely-adopted, but it is used in Samsung's KNOX enterprise mobile security solution [21].

TrustZone partitions the system memory into two worlds, i.e., each world has its own address space. The secure world can see all the physical memory in the system, but the normal world can see only its own. Cache memories are tagged as secure or non-secure to protect them from accesses from the normal world. Individual hardware peripherals can be assigned to the secure world, and for these peripherals hardware interruptions are configured to be directly routed to, and handled by, the secure world. As a result, it is possible to secure peripherals such as memory, storage (e.g., an SD card), keyboard and screen to ensure they are protected from software attacks. In general, TrustZone protects the confidentiality and integrity of any computation and data in the secure world, so untrusted code running in the normal world cannot access these resources [22], [23].

B. Android Architecture

Android is a Linux-based open source software stack for mobile devices. It consists of a modified Linux kernel, a middleware layer, and an application layer. The Linux kernel provides OS services like memory management, process scheduling, device drivers, file system support and network access. The middleware layer consists of native Android libraries, the Android run-time environment, and an application framework. The application framework consists of applications that provide system services, e.g., the Activity Manager that manages the life cycle of applications, the Application Installer that installs new applications, and the Package Manager that maintains information about all applications loaded in the system.

Android applications are implemented in Java, but they can also incorporate C/C++ native libraries through the Java native interface (JNI). Java code is compiled to a custom bytecode format, the Dalvik EXecutable (DEX) format. Android applications are comprised of four basic types of components: Activities, Services, Content Providers and Broadcast Receivers.

Android applications are distributed in files in the Android Application Package format (APK or `.apk`). An APK file is essentially a zip archive containing all the application resources: bytecodes (`.dex`), manifest file, media files, etc. After a successful installation of an application, its APK file is stored in the file system (in `/data/app/`). For static analysis of an Android application, one has to unzip the APK file, then decompile the `.dex` file or translate it into Java source code using appropriate tools (e.g., dex2jar, APKtool). Every APK file includes a manifest file (`AndroidManifest.xml`) with essential information about the application: list of components that compose it; permissions that the application needs to run; permissions that other applications must have in order to interact with the application's components.

III. DROIDPOSTURE ARCHITECTURE AND DESIGN

DROIDPOSTURE gives external service providers a mechanism to evaluate the posture of a smartphone and restrict access to critical data based on posture. DROIDPOSTURE is a software component that runs in the secure world of a mobile device. The posture information is requested by external services when smartphone applications request to use a service or by local application to understand the posture of the smartphone.

A. Use Cases

In this section, we describe two use cases for DROIDPOSTURE in order to help understand how it can be used.

In the first scenario, an employee of an enterprise that has adopted BYOD (Bring Your Own Device) uses his smartphone to connect to the corporate network and access corporate data (the enterprise is the external service provider). This scenario is quite realistic as it is essentially an example of network access control [19], [20], frequently used today in large companies, except that we consider mobile devices instead of laptops and that the component running in the device is DROIDPOSTURE, so it is protected using TustZone. The enterprise security policies restrict access to its assets, so it requires adequate posture/compliance of the smartphone in order to ensure that this device conforms to these policies. The security policies may include various combinations of data such as location of the device, OS version, device information (serial number, model, open ports, etc.), and security status of the smartphone. The posture assessment involves four actors: the employee, the enterprise, the smartphone application that allows the employee to connect, and the DROIDPOSTURE service. The employee sends an access request to the enterprise via the application. The enterprise communicates with the DROIDPOSTURE service and requests the smartphone posture. The DROIDPOSTURE service sends a signed block of data with posture data to the enterprise. The enterprise checks the compliance of the smartphone against its security policies and, if needed, sends remediation instructions for the employee to fix the smartphone.

In the second scenario, a bank customer uses a mobile banking application on the smartphone to do a transaction. Depending on the sensitivity of the data, the bank, which is the external service provider, requires the posture information to determine whether the data passed to the smartphone will be compromised by malware residing in the smartphone. This mobile transaction scenario involves four actors: a customer, a bank, a bank application, and the DROIDPOSTURE service. To start the transaction, the customer sends a service request to the bank via the bank application. In order to ensure the security of privacy-sensitive data, the bank communicates with the DROIDPOSTURE service via the bank application to request posture information. DROIDPOSTURE performs a posture assessment, sends the signed posture to the bank, and the bank sends the data (or not).

B. Threat Model and Assumptions

We assume that DROIDPOSTURE runs in an ARM processor with TrustZone. We assume that in the normal world the mobile OS and the applications it executes are untrusted, i.e., that they may be malicious or compromised by malware or hackers. In contrast, we assume that the software running in the secure world, including the DROIDPOSTURE software, is trustworthy. In order to reduce the size of the TCB, the size of the software executed in the secure world has to be as small as possible, so it should not include for example a network stack. The size of the API is also as small as possible to reduce the attack surface, and all inputs are validated. These measures make software attacks against the secure world unlikely to be successful, so in this paper we assume they are not. We also assume that the device (i.e., the normal world) is not yet compromised when DROIDPOSTURE is first installed (it can come pre-installed with the device).

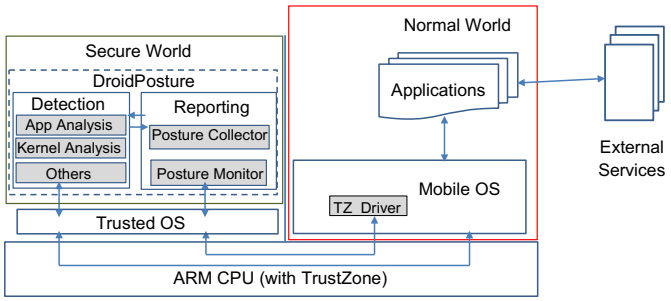

Fig. 1. Architecture of a mobile device running DROIDPOSTURE. The grey boxes are components of the DROIDPOSTURE service.

Each DROIDPOSTURE instance in a device has an identifier id and a public-private key pair (Ku, Kr) for some public-key cryptographic scheme (e.g., RSA). It has also a certificate containing the public key, signed by some trusted certification authority (CA). We also assume the existence of a collision-resistant hash function (e.g., SHA-256).

C. Architecture

Figure 1 represents a mobile device running DROIDPOSTURE and communicating with some external services. The *normal world* runs the usual mobile device software: a mobile OS and applications. It includes also a driver (`TZ_Driver`) that allows software in the normal world to call functions in the secure world, DROIDPOSTURE in our case. This driver allocates a shared memory zone that is used for the application to pass inputs to DROIDPOSTURE, and for DROIDPOSTURE to return outputs to the application (assessment results in our case).

The *secure world* runs the DROIDPOSTURE service. This world includes a small trusted OS that provides basic functions for software running in that world (processes, file access, etc.). Besides its private memory space, it is also configured to have a private persistent storage partition (either part of the internal memory or of an SD card). DROIDPOSTURE itself is composed of two modules, for detecting and reporting the posture of the normal world. The *detection module* is further divided in two modules that we implemented in our prototype – *application analysis* and *kernel analysis* – although others may be designed and used (*others* in the figure). The *reporting module* provides an interface between the normal world and the detection modules. The *posture monitor* receives, validates (for protection against buffer overflows and other input attacks), and replies to requests for posture data from the normal world. The *posture collector* collects information from the detection module(s) and signs it.

The bootstrapping of the device starts by running the kernel of the secure world, so this kernel is the *static root of trust for measurement* [24]. Before passing the control to the normal world and starting the execution of the normal world, the *kernel analysis* module calculates and stores a hash (measurement) of the normal world operating system. This is process is denominated *trusted boot* [24] and may involve storing hashes of other modules, if needed.

D. Posture Reports

DROIDPOSTURE provides information about the posture of the device in the form of a *posture report*. The format of a posture report is: $\langle id, posture\_data, nonce \rangle_{S_{Kr}}$, where id is the identifier of the DROIDPOSTURE instance, $posture\_data$ the posture data itself, $nonce$ a nonce (a random number used only once for replay protection) that comes with the request for posture data, and S_{Kr} a signature obtained using the instance private key.

Posture reports can be delivered to applications or transferred to external services via the normal world. In both cases an application running in the normal world requests posture data from the *posture monitor* module. This request contains the above-mentioned nonce. The module then invokes the *posture collector* module that requests the *detection* modules to collect the posture of the device. The posture collector gets the result, creates and signs the posture report, and sends it to the posture monitor. The latter sends the posture report to the application, which may optionally send it to an external service.

As the mobile OS and the applications may be compromised, we do not trust them to deliver the posture report unmodified to the application or external service that requested it. The authenticity and integrity of the report are verified using the digital signature S_{Kr} calculated using the private key of the DROIDPOSTURE instance in the device. Such an attacker might still do a denial of service attack by deleting or modifying all reports, but this would be understood by the service provider as consequence of a compromised device.

Figure 2 illustrates the steps for providing a posture report to an external service. An application starts the interaction with the external service, e.g., the backend of the application (step 1). The external service replies and provides a nonce (step 2). The application forwards the nonce to DROIDPOSTURE in the secure world and asks for posture assessment (step 3). DROIDPOSTURE performs the posture assessment, creates and signs the posture report, then sends it to the application (step 4). The application sends this report to the external service (step 5). Finally, the external service verifies the nonce and the signature, using the certificate of that DROIDPOSTURE instance (Section III-B). If they are correct, it then interprets the posture data. If it finds the posture acceptable it continues to interact the application, e.g., sending it some data (step 6).

E. Application Analysis

We designed two *detection* modules, although others may be used. As mentioned before, these mechanisms illustrate how

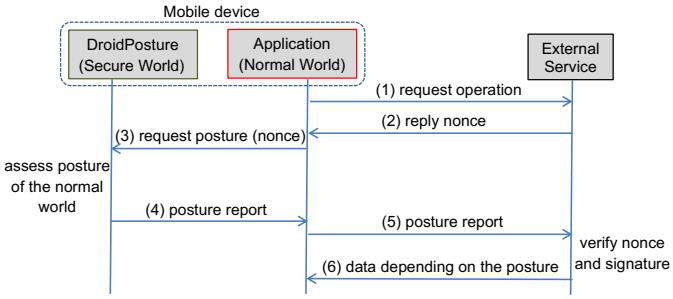

Fig. 2. DROIDPOSTURE providing a posture report to an external service.

posture analysis may be implemented in DROIDPOSTURE.

This section is about the *application analysis* module (Figure 1). This module provides two static analysis mechanisms to detect the presence of malware in Android applications: *signature-based detector* and *learning-based detector*.

By static analysis we mean analysis of code, without executing it. The application analysis module first unpacks the application APK and obtains the bytecodes. Then, the bytecode file is passed as input to the signature-based and learning-based detection mechanisms. Prior to this process, the hash of the APK file is compared with the hashes of the APKs already analyzed stored in the secure world persistent storage partition, in order to avoid re-executing the analysis. If it has already been analyzed, the result obtained previously is returned. Notice that if the APK changes, then its hash also changes (collision resistance property of the hash function).

1) Signature-Based Detector: The first and simplest malware detection technique is based on pattern matching. Malware detectors have a database of distinctive patterns – *signatures* – of malicious code and they look for them in applications. Malware has to be public for a period of time so that signatures can be generated for that specific malware family.

Our *signature-based* mechanism detects malicious applications or injected malicious code based on similarities of control flow graphs (CFGs). A control flow graph represents the control flow of a program. The signature-based mechanism takes the bytecode file and converts each function in the bytecode into a string that represents the CFG of the function. The comparison of similarity of the CFGs is done by using a similarity algorithm such as Kolmogorov distance and normalized compression distance (NCD). The CFG string of each function is compared against the CFGs (the signatures) of known malware in the database to verify if they are similar.

2) Learning-Based Detector: As the *signature-based* detection mechanism only detects malware for which it has signatures, we propose a complementary mechanism to detect malware in applications. The *learning-based* detection mechanism relies on a machine learning classifier. This mechanism has two phases. In the first phase, *training*, the mechanism statically examines and extracts selected features from known malware samples and benign applications to build feature vectors; then, these features are used to train a machine

learning classifier to distinguish malware from normal code. In the second phase, *detection*, the classifier is used to check applications for malware. Notice that the detection phase is the one that is executed by DROIDPOSTURE itself; the training phase is executed beforehand, by whoever manages the DROIDPOSTURE service.

The selection of features is essential for the efficiency of the detection mechanism. Redundant or relevant features may present several problems such as misleading the learning algorithm, and increasing model complexity and run time. We use the features below, extracted from the `AndroidManifest.xml` file and the `.dex` file. All features are binary, i.e., either the application has it or not:

- Requested permissions. Android uses permissions for restricting access to the device resources. Permissions are granted by the user during application installation, or later in the latest versions of Android. Malicious applications request certain permissions (e.g., `SEND_SMS`, `READ_SMS`) more often than benign applications. Requesting a security sensitive permission is a feature.
- Sensitive function calls. Among thousands of Android API functions, we consider API calls invoked by applications that allow to access sensitive data or resources. For example, APIs for accessing the user's personal information, network details, device ID, and sending SMSs.
- Suspicious intents. An intent is an abstract description of an operation to be performed. Application components are activated using intents. We consider intents that perform sensitive actions as features (e.g., `android.intent.action.CALL`, `android.intent.action.DIAL`).
- Suspicious content URI. A content URI is used to locate data in a content provider. It can be used to leak users personal data or to access another applications data. For example, `content://sms/inbox` can be used to read SMS messages from inbox.
- Arbitrary code execution. Execution of native code using JNI or Linux commands. For example, `Ljava/lang/Runtime;->exec()` executes command `exec()` in a separate process.

In the training phase, to construct feature vectors, we retrieve the selected features from each malware sample and benign application and store them as binary numbers, 1 or 0, respectively for presence or absence of the feature. Furthermore we assign a class to each feature vector, *M* for malware and *B* for benign application. Feature vectors are then provided to the machine learning classifier.

We use the *k-Nearest Neighbours* (kNN) algorithm as classifier [25]. Given vectors of N classes as training samples, kNN classifies an unknown sample by searching the entire set of training samples for the k nearest based on a distance metric, then the unknown sample is assigned to the class most common among its k nearest. k is a positive integer and if $k = 1$, then the unknown sample is simply assigned to the class of the single nearest sample.

F. Kernel Analysis

This section presents the *kernel analysis* module of Figure 1. Again, this module provides two kinds of analysis.

A rootkit is a piece of malware that gains privileged access to a system, hides itself from the user and the OS, then stealthily carries out some kind of malicious activity. Rootkits may be roughly classified in two classes. *User-level rootkits* replace system binaries and libraries with customized versions. *Kernel-level* rootkits modify the kernel, for example by adding code into the running kernel memory image (`/dev/mem`) or by injecting a Loadable Kernel Module (LKM).

The *kernel analysis* module starts by calculating a hash of the normal world kernel and by comparing it with the hash obtained during the boot of the device (Section III-C). If the hashes are different, the kernel analysis fails immediately.

1) Syscall Table Checker: The Android kernel provides system calls (*syscalls*) that allow applications in user mode to interact with the kernel. Syscalls are one of the primary targets for kernel-level rootkit writers. The kernel uses a *syscall table*, an array of pointers mapping each syscall number to the corresponding function in kernel memory. Modifying a syscall table entry is a popular way to intercept the execution flow of any system service. Kernel-level rootkits often modify syscall table entries to point to new, malicious, system calls. Therefore, in order to detect a kernel-level rootkit, the first step is to verify the integrity of the system call table.

Each time the kernel is compiled, a file containing the map of kernel symbols and addresses is created (`System.map`). Comparing the addresses of syscalls in the `System.map` with the addresses in the *syscall table* during runtime detects if system calls have been redirected, which may be an indication that the kernel has been compromised by a rootkit.

When DROIDPOSTURE is installed, our *kernel analysis* mechanism starts by making a copy of the addresses of system calls in `System.map` and storing them in the secure world. Then during runtime the mechanism simply compares that copy with the values in the syscall table in the normal world. Recall that we assume that the system is not compromised when DROIDPOSTURE is installed.

2) Kernel Integrity Checker: Besides syscall table integrity checking, the *kernel analysis* module is capable of checking the kernel code for modifications to detect rootkits. As the kernel is not supposed to change during runtime, changes are probably a sign of malware. For example, a rootkit can replace the first few bytes of some system call functions with a `jmp` instruction that redirects the execution to malicious code.

In order to verify the kernel integrity, the kernel integrity checker calculates a hash of the kernel code memory pages of the Android OS running in the normal world and compares it against a hash calculated when the system was in a pristine state, which is stored in the secure world persistent storage partition. To calculate a hash value, the start address and length of the target memory pages are required. The kernel integrity checker finds the virtual address of the kernel code in the copy of the `System.map` file stored in the secure world and

translates this address to the secure world address space before evaluating the hash value.

IV. DroidPosture Implementation

We implemented a prototype on an i.MX53 QSB board equipped with a Cortex-A8 single core 1 GHz processor, 1 GB DDR memory, and a 4GB MicroSD card. Most TrustZone-enabled smartphones are locked in such a way that it is not possible to use the secure world, so we opted for this board.

A. Runtime Environment

Genode is a framework for building special-purpose operating systems [26]. It provides a collection of OS building blocks, e.g., kernels, device drivers, and protocol stacks. Genode can reduce OS complexity for security-sensitive scenarios, which makes it an appealing foundation for an OS to run in the secure world. Genode Labs has released a TrustZone virtual machine monitor (VMM) demo for our board, which enables the execution of Genode in the secure world, while a guest OS such as Linux, monitored by a Genode hypervisor, runs in the normal world. We used this demo as a starting point to implement our prototype.

In the secure world, we implemented DroidPosture based on a program called `tz_vmm` that runs on top of the Genode kernel. In the normal world, we run Android for the i.MX53 series from Adeneo, previously freescale (`http://witekio.com/cpu/i-mx-53/`). We used the Linux/Android kernel modified by Genode Labs for this board. The kernel is modified so as to prevent the normal world from directly accessing certain resources such as hardware, persistent storage and memory that are set as secure within the central security unit (CSU) initialization. To create the secure world persistent storage partition, we used the Genode partition manager (`part_blk`) that supports partition tables and provides a block session for each partition of a SD card. This allows the partitions to be addressable as separate block sessions and makes it is easy to grant or deny access. We used this scheme to reserve a partition for the secure world.

We run `TZ_Driver` in the kernel for an application in the normal world to issue a hypercall to exit the normal world and trap into the secure world, using the SMC instruction. A shared buffer in RAM allows passing data between the two worlds. Some of the general purpose CPU registers are used to store information about the shared buffer between the two worlds, including its address and length.

In our prototype we used components written in Python, which required installing Python 2.6 in the secure world using the Genode libports repository. This is undesirable because it increases the size of the TCB. However, this is not a limitation of our proposal, but of the current prototype. DroidPosture itself does not need to use Python code.

B. DroidPosture Modules

Table I shows the code size of each module implemented in the DroidPosture service.

TABLE I
Lines of code for the DroidPosture modules.

Modules	Code Size (LOC)
Application Analysis	30484
Kernel Analysis	142
Posture Collector	86
Posture Monitor	121

The *application analysis* module is based on Androguard [27], an open source tool written in Python. It is able to unzip an APK file, obtain its metadata and bytecodes. Androguard has a module to create the control flow graph (CFG) for each function in a bytecode file. In addition, Androguard has several built-in signatures that are able to detect known malicious applications. Since Androguard is a complete feature-rich framework, we use its modules to disassemble an application's Dalvik bytecode, then create a CFG for each function, and compare these CFGs with the malware CFGs (the signatures) that are stored in the secure world persistent storage partition. In addition to Androguard, we modified Androwarn [28] to extract the features (Section III-E2) from malware and benign applications to build feature vectors for the learning-based detector.

The *kernel analysis* module needs to access the normal world memory. TrustZone configuration within Genode partitions the DDR RAM between the secure world and the normal world using the multi-master multi-memory interface (M4IF) [26]. `tz_vmm` is able to read the normal world's RAM via an IOMEM session during its start-up routine. The memory is mapped as uncached to the secure world's address space, thus the whole normal world memory can be accessed by the kernel analysis module in the secure world.

V. Performance Evaluation

To evaluate the performance of DroidPosture, we used a set of micro- and macro-benchmarks by considering calls to the DroidPosture service that: (i) return immediately (baseline); (ii) do application analysis, only signature-based; (iii) do application analysis, only learning-based; (iv) do application analysis, both mechanisms; (v) do kernel analysis, only syscall table checker; (vi) do kernel analysis, only kernel integrity checker; (vii) do kernel analysis, both mechanisms; (viii) do all the detection mechanisms.

In the *micro-benchmarks*, an application (in the normal world) sends a request for posture and gets a reply back from DroidPosture (in the secure world). The *macro-benchmarks* are used to evaluate the posture assessment transmission protocol. For this purpose, we used a remote server which runs on a standard laptop. The server listens for incoming requests from the application in the normal world and sends requests for posture to the DroidPosture service running in the secure world of our board via the application.

A. Micro-benchmarks: mechanism performance

We used the calls mentioned above to evaluate the overhead of DroidPosture. To measure the time for the baseline (i),

TABLE II
DROIDPOSTURE DELAY WHEN CALLED LOCALLY (IN SECONDS).

Size		Calls						
APK	.dex	ii	iii	iv	v	vi	vii	viii
12KB	5KB	1.81	0.8	2.23	0.15	1.64	1.75	4.01
19MB	39KB	14.12	1.51	14.92	0.14	1.63	1.77	16.26
4MB	67KB	31.84	1.57	32.26	0.14	1.63	1.77	33.19
250KB	103KB	29.03	1.55	29.70	0.14	1.63	1.77	30.40
803KB	153KB	66.28	5.85	69.96	0.14	1.63	1.77	71.21
401KB	305KB	206.21	7.86	206.54	0.14	1.63	1.77	208.42

TABLE III
DROIDPOSTURE DELAY WHEN CALLED BY A REMOTE SERVICE (SEC.).

Size		Calls						
APK	.dex	ii	iii	iv	v	vi	vii	viii
12KB	5KB	1.85	0.89	2.29	0.17	1.67	1.78	4.59
19MB	39KB	14.27	1.56	14.94	0.16	1.65	1.78	16.74
4MB	67KB	31.86	1.60	32.38	0.16	1.65	1.78	34.05
250KB	103KB	29.07	1.57	29.77	0.16	1.65	1.78	31.24
803KB	153KB	66.39	5.87	69.12	0.16	1.65	1.78	72.32
401KB	305KB	206.61	7.88	207.11	0.16	1.65	1.78	208.89

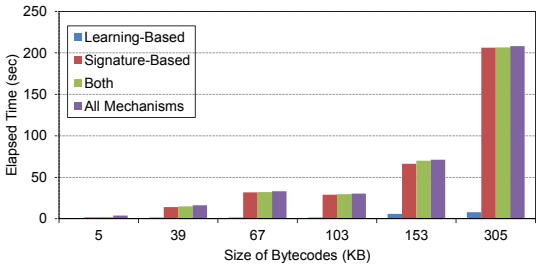

Fig. 3. DROIDPOSTURE delay when called locally with emphasis on the applications analysis modules (in seconds).

the application in the normal world sends a request for posture to the DROIDPOSTURE service in the secure world that does not execute any analysis module. We repeated the experiment 1000 times and obtained an average of 0.082 *ms*, with standard deviation of 0.0061.

For the rest of the calls the process is similar, except that DROIDPOSTURE executes a subset of the analysis modules. We expected calls to the application analysis modules to depend on the size of the applications, so we considered a set of applications with different sizes (downloaded from Google Play Store). We did experiments for the combinations of calls (ii) to (viii) and all bytecode sizes. The results of these experiments are shown in Table II. Moreover, in Figure 3 we represent the same values but only for the combinations of application analysis modules and the total.

These results allow us to extract several conclusions. First, in the table it is clear that calls to the kernel module have a delay that is *independent* of the size of the application, as expected (columns *v-vii*). Second, both the table and the figure show that delay of the signature-based analysis grows with the size of the `dex` file, to the point of becoming unusable (column *ii*). This was expectable as it converts all the functions in the bytecodes into CFGs, which increase with the size of the code. Third, the table and the figure also show that learning-based analysis grows slowly with the size of the `dex` file, showing that this form of analysis is much simpler and faster than the signature-based (column *iii*). Fourth, they also show that these two delays depend on the size of the `dex` files, not on the size of the APK files, which often contain many files that are not analyzed, e.g., images and video (columns *ii-iii*). Fifth, all mechanisms and their combination seem to be usable, except the form of signature-based analysis we considered.

B. Macro-benchmarks: DroidPosture in a company

To evaluate the performance of the DROIDPOSTURE service in the context of a realistic use case, we measured the total time for the remote server to send a request for posture and to get a reply back from the service (see Figure 2). We used a LAN network to emulate the case of posture being provided inside a company.

We measured a round trip time (RTT) between our board and the remote server of 0.497 *ms*. We used the same calls as before. In this case, the time for a baseline call was 1.92 *ms*, with standard deviation of 0.096. The results of these experiments are shown in Table III. The trends are essentially the same that were observed with the micro-benchmarks, with the additional delay of the network.

VI. SECURITY EVALUATION

As previously mentioned, the specific modules we implemented in DROIDPOSTURE serve mainly to demonstrate the kinds of analysis it can make and that it can support several. Nevertheless, we evaluated experimentally the quality of the detection made by our four modules, which we present here.

We used 500 malware samples from the Drebin datasets [29]. These datasets contain samples from 179 different malware families collected between August 2010 and October 2012. We balanced the number of samples from different malware families. For benign applications, we randomly downloaded 30 applications from 8 different categories on Google Play Store and verified them through VirusTotal that runs samples through around 10 anti-virus products, in order to get some confidence that they had no malware (`https://www.virustotal.com`).

To evaluate the detection performance of the *learning-based detection* mechanism, we randomly split our datasets into a training set (66%) and a test set (33%). The training set was used to determine the classification model, whereas the test set was used for measuring the detection performance. We use as metric *accuracy*, which evaluates the ratio of applications correctly classified (it is given by the number of applications correctly classified as good or bad, divided by the total number of applications evaluated). The result shows that the learning-based mechanisms using *kNN* with $k = 3$ had accuracy of 89.4% with a false positive rate (i.e., percentage of samples wrongly identified as malware) of 4%. The detection performance is relatively good, although our dataset is not large. This suggests that our features effectively model malicious code.

The *signature-based detector* achieves better detection performance for samples that have signatures in the database. To

test its performance, we created signatures from over 100 different malware families, such as DroidDream, DroidKungfu, DogoWar and foncy. The signature-based detector was able to detect malware samples from those malware families correctly with approximately 100% accuracy. However, the learning-based mechanism is more effective than the signature-based mechanism for applications that contain unknown malware.

To illustrate the effectiveness of *kernel analysis* modules, we deployed the Mindtrick kernel-level rootkit on our board [30]. The Mindtrick rootkit replaces the entry for the read syscall (`sys_read`) to instead point to the address of a malicious function injected into the kernel. It allows attackers to obtain a reverse TCP shell on Android devices. Our kernel analysis module in the *secure world* is able to detect this rootkit by reading each address in the system call table from the *normal world* memory, and compare it with each syscall address listed in `System.map`. It inserts an error for the `sys_read` syscall entry in the posture report.

VII. Conclusion

With the strong adoption of mobile devices, new malware is emerging to steal or manipulate sensitive data processed in mobile applications. Anti-malware applications have been proposed to overcome this problem, but these applications are often unprotected from the malware itself. This paper presents the DroidPosture service, which is protected by the ARM TrustZone extension. The service aims to securely detect intrusions in an Android device and report posture information for external services. We implemented a set of application and kernel analysis mechanisms to exemplify the kind of posture assessment that our service can do, although the specific analysis to do are probably specific to different scenarios. The performance of these mechanisms seems to be adequate for many applications, with the exception of the signature-based analysis that is slow for large applications.

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). We warmly thank Nuno Santos commenting this work.

References

[1] M. Kitagawa *et al.*, "Gartner inc. market share: Final PCs, ultramobiles and mobile phones, all countries, 3Q16," 2016.

[2] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012, pp. 101–112.

[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 (NDSS)*, 2012, pp. 5–8.

[4] 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.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 2011, pp. 239–252.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 1–6.

[7] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 639–652.

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

[9] N. Leavitt, "Mobile phones: the next frontier for hackers?" *IEEE Computer*, vol. 38, no. 4, pp. 20–23, 2005.

[10] A. Greenberg, "Sneaky Android RAT disables required anti-virus apps to steal banking info," Jul. 2014, SC Magazine, https://www.scmagazine.com/sneaky-android-rat-disables-required-anti-virus-apps-to-steal-banking-info/article/538770/.

[11] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the 10th Symposium on Network and Distributed System Security*, 2003, pp. 191–206.

[12] Y. Fu and Z. Lin, "Bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 586–600.

[13] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011, pp. 297–312.

[14] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 569–584.

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

[16] D. Liu and L. P. Cox, "Veriui: Attested login for mobile devices," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 2014.

[17] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, 2012, pp. 365–378.

[18] D. Grawrock, *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[19] P. Sangster, H. Khosravi, M. Mani, K. Narayan, and J. Tardo, "Network endpoint assessment (NEA): Overview and requirements," Internet Requests for Comments, RFC Editor, RFC 5209, June 2008.

[20] D. V. Hoffman, *Implementing NAP and NAC Security Technologies: The Complete Guide to Network Access Control*. John Wiley & Sons, 2008.

[21] Samsung, "Samsung Knox," https://www.samsungknox.com.

[22] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to build a trusted language runtime for mobile applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 67–80.

[23] 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.

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

[25] N. S. Altman, "An introduction to kernel and nearest-neighbor non-parametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[26] Genode Labs, "ARM TrustZone, an exploration of ARM TrustZone technology," http://genode.org/news/an-exploration-of-arm-trustzone-technology, 2014.

[27] Androguard, http://code.google.com/p/Androguard.

[28] Androwarn, https://github.com/maaaaz/androwarn.

[29] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket." in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2014.

[30] Mindtrick rootkit, https://github.com/ChristianPapathanasiou/defcon-18-Android-rootkit-Mindtrick, 2014.