T2Droid: A TrustZone-based Dynamic Analyser for Android Applications

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*—Android has become the most widely used mobile operating system (OS) in recent years. There is much research on methods for detecting malicious Android applications. Dynamic analysis methods detect such applications by evaluating their behaviour during execution. However, such mechanisms may be ineffective as malware is often able to disable anti-malware software. This paper presents the design of T2DROID, a dynamic analyser for Android that uses traces of Android API function calls and kernel syscalls, and that is protected from malware by leveraging the ARM TrustZone security extension. In our experimental evaluation T2DROID achieved accuracy and precision of 0.98 and 0.99, respectively, with a kNN classifier.**

I. INTRODUCTION

Android has become the most widely used mobile operating system (OS), with a smartphone market share of more than 85% in Q3 2016 [1]. However, the existence of several Android application marketplaces [2] makes it feasible for attackers to distribute malicious applications, e.g., in the form of repackaged applications [3]. Researchers have shown that Android devices are vulnerable to a large number of attacks, e.g., applications and libraries that misuse their privileges [4], [5], run root exploits that steal private information [6], take advantage of unprotected interfaces [7], [8], do confused deputy attacks [9], and do collusion attacks [10].

There is much research on methods for *detecting malicious Android applications*. These methods can be roughly categorized in two approaches: static and dynamic analysis. *Static analysis* methods detect if an application is malicious by inspecting its code or metadata, without executing the application [11], [12]. This approach is often used by application marketplaces to evaluate applications before starting to distribute them. However, malicious applications may use obfuscation techniques to make static analysis hard.

Dynamic analysis methods, on the contrary, detect malicious applications by evaluating their behaviour during execution [13]–[20]. Most mechanisms of this kind involve extending the mobile OS kernel or the middleware (Dalvik or ART), so their security is based on the assumption that these components are not compromised. However, such mechanisms may be ineffective as malware is often able to disable anti-malware software [21], as in the recent case of HijackRAT [22], sometimes due to vulnerabilities that allow this [23]. An alternative approach is to run the protection mechanism in an hypervisor or a thin virtual machine (VM) isolated from the fat VM that runs the OS and the applications, as in Droidscope [24]. However, current mobile devices do not use this kind of virtualization, which would have a considerable overhead.

TrustZone is a hardware security extension incorporated in recent ARM processors [25]. It partitions the system virtually in two parts: the *normal world* that runs the mobile OS (e.g., Android) and its applications; and the *secure world* that runs trusted applications or security services on top of a small trusted OS. The memory space, peripherals, and interrupts assigned to the secure world are isolated from the mobile OS and its applications. On the contrary, the secure world can access the resources of the normal world.

This paper presents the design of the *Trustzone-based Trace analyser for anDroid applications* – T2DROID. This mechanism does dynamic (runtime) analysis of applications to detect malware on Android-based mobile devices. T2DROID uses traces of Android API function calls and kernel system calls (syscalls) performed by an application to detect whether it is malicious or not. This combination of the two types of calls allows observing operations with a clear semantics (e.g., sending an SMS message), while not letting malware escape this detection by running native code and doing syscalls instead of calling API functions. It uses a machine learning classifier to do the detection, which allows it to be configured without a human to manually develop detection rules, and to be reconfigured easily when new malicious applications are discovered. T2DROID is protected from malware by leveraging the TrustZone extension. The detection itself is performed inside the secure world. The capture of the API function calls and syscalls has to be done by software components running in the Android kernel, in the normal world, but there is a component protected in the secure world that verifies the integrity of these normal world components and of the mobile OS kernel.

T2Droid does not aim to substitute static analysis mechanisms that should be used to test an application before it starts being distributed in an application marketplace [26]. It is a complementary mechanism that provides a second layer of protection at runtime, similarly to anti-virus software. This second layer of defense is important, as many applications distributed in marketplaces are malicious [5], [27]. However, unlike anti-virus software, T2DROID is protected from malware by leveraging the TrustZone extension.

We envisage three main use cases for T2DROID. The first two consider personal mobile devices, typically smartphones. The first is to run T2DROID automatically whenever an application is downloaded from a marketplace and installed. The first time the application is executed, T2DROID would be executed during a configurable amount of time or number of API calls in order to check if the application is trustworthy. The second is to run T2DROID when requested by the backend of the mobile application, i.e., of the part of the application that runs in the cloud or company servers. The objective would be for the backend to assess if the application in the mobile device has been compromised. The third would be to run T2DROID in devices targeted specifically at testing the trustworthiness of applications. T2DROID would run when the application started to run, but would be executed for a larger period of time in order to check the application for a longer period.

We did an experimental evaluation of T2DROID. The first objective was to understand what was the classifier with best performance among a set of those available. The one that performed best was kNN. The second was to understand what was the performance of that classifier. We obtained accuracy of 0.98, precision of 0.99, and recall of 0.99 with our experimental dataset and the combination of the two kinds of calls. Finally we measured times for trace transfer, feature vector preparation, and classification of 2.5s and for integrity checks made by T2DROID of 1.3s.

The main contributions of the paper are: (1) the design of T2DROID, a dynamic analysis system for Android that leverages the ARM TrustZone extension to securely detect malicious applications; (2) a machine learning detection scheme based on tracing both Android API function calls and kernel syscalls; (3) an implementation of T2DROID for Android on the NXP Semiconductors i.MX53 Quick Start Board (QSB); (4) an experimental evaluation of T2DROID, considering both detection and performance.

II. ANDROID AND ARM TRUSTZONE

This section provides background information on the main technologies underlying T2DROID: Android and TrustZone.

A. Android

Android is an OS for mobile devices, originally created by Google. Android was not developed from scratch, but based on the Linux OS and the Java virtual machine (JVM). Android comprises three layers. The first, the *kernel layer*, is essentially a modified Linux kernel. It provides basic OS services such as memory management, process scheduling, device drivers, file system support, and network access. The second, the *middleware layer* (or runtime), was originally a modified JVM called Dalvik, but was substituted by ART (Android RunTime) in Android 5.0 Lollipop. Dalvik did just-in-time compilation, i.e., compiled applications during their execution. ART does ahead-of-time compilation, i.e., compiles applications when they are installed. The middleware provides also a set of libraries and an application framework. This framework provides services to manage the life cycle of applications, to install them, and to maintains information about the applications loaded. The third layer is the *application layer*. This layer includes core (i.e., installed by default) applications such as browser, phone dialler, and contact manager. Moreover, it allows downloading applications from marketplaces. Each Android application runs in its own process, with its own *virtual machine* (VM). They are typically written in Java, although they may also include and run native code, with the assistance of the Java Native Interface (JNI). The applications are distributed in files in the Android Application Package format (APK). An APK file is essentially an archive containing a bytecodes file (.dex), a manifest file, media files, etc.

B. ARM TrustZone

ARM is a company that creates CPU designs. It does not produce CPUs, but licenses the designs to companies that produce them. Starting with ARMv6, these designs include the TrustZone extension. This extension is part of the CPU itself, not an external chip.

The TrustZone technology provides two trust domains, or worlds. The normal world usually runs a common OS – Android in our case – and its applications. The secure world is supposed to run a smaller kernel in which trust can be placed, and security services – most of T2DROID in our case. The context switch between the two worlds is controlled by a higher privilege mode, the monitor mode. Software in the normal world can force a switch to the secure world by calling the secure monitor call (SMC) instruction.

The secure world provides code and data integrity and confidentiality because untrusted code running in the normal world cannot access the resources of the secure world. The memory spaces of the two worlds are isolated. Each world has access to its own memory management unit (MMU) to maintain separated page translation tables. Cache memories are TrustZone-aware, i.e., cache lines are tagged as secure and non-secure so access to secure cached content from the normal world is always denied. Certain hardware peripherals and memory can be assigned exclusively to the secure world.

III. T2DROID ARCHITECTURE AND DESIGN

This section presents T2DROID's architecture and design.

A. Threat Model and Assumptions

T2DROID runs in an ARM processor with TrustZone. In the normal world, the mobile OS and the applications it executes are not trusted, i.e., they may be malicious or compromised by malware or hackers. In contrast, we assume that the software running in the secure world, including the T2DROID software, is trustworthy. The secure world is the Trusted Computing Base (TCB) [28] of our system, the size of the software executed there has to be as small as possible, so it does not include a network stack or a mobile OS. The size of the API to the secure world is also as small as possible to reduce the attack surface, and all its inputs are validated, so we assume attacks against the secure world cannot be successful. Malware or attackers might be interested in disabling T2DROID, as they

do to anti-virus and other anti-malware software, but (1) we assume they cannot compromise the part of T2DROID that runs in the secure world and (2) we use code in the secure world to verify the integrity of the components that run in the normal world. We assume the existence of a collision-resistant hash function (e.g., SHA-256).

B. Architecture

The architecture of T2DROID is shown in Fig. 1. The *normal world* runs Android and applications. It also includes a part of T2DROID, the two *tracer* modules, which obtain information about the behaviour of the application being checked. The *API calls tracer* and *syscalls tracer* are in charge of monitoring Android API calls and kernel syscalls, respectively. These components are tightly integrated with the Android environment, so we place them in the normal world. It would be possible to place them in the secure world, but there are two drawbacks: the implementation would be much more complex; the performance overhead would be high as there would be at least an order of magnitude more context switches between the two worlds.

The TrustZone driver (*TZ_Driver*) is a kernel level driver, which enables the tracer module to communicate with the secure world. It allocates a shared memory zone that is used for the *tracer* module to pass trace files to the *detector* module in the secure world. It is also used by an application or another module to order T2DROID to inspect an application. For example, in the first use case mentioned in the introduction, the order may come from a modified *application installer*, the Android component that installs new applications.

The *secure world* runs a small trusted OS that provides basic functions for software running in that world (processes, file access, etc.) and modules of T2DROID. The *integrity checker* module is comprised of two modules. The *tracer checker* verifies the integrity of the *tracer* module, whereas the *kernel checker* checks the integrity of the Android kernel running in the normal world. The *detector* module receives trace data of an application from the *tracer* module and performs detection using a machine learning classifier.

Next we explain each component of the architecture.

C. Tracers

T2DROID analyses the behavior of an application by observing the calls it makes. Tracers extract sequences of API calls and syscalls. Next we present these two components, starting with the *API calls tracer*, then the *syscalls tracer*.

1) API calls tracer: Android applications rely heavily on middleware-layer libraries, i.e., they frequently call their APIs. Access to these APIs is protected using Android's permission framework, but users are compelled to give the permissions requested, otherwise they cannot use them. The sequence of API calls performed by an application reveals to some extent its behavior. Malicious applications often make calls that have legitimate uses, but that may be associated with malicious behavior. Examples include sending SMS messages, making phone calls, or accessing the user's contacts. Therefore, analysing Android API function calls is a way of detecting malicious behavior.

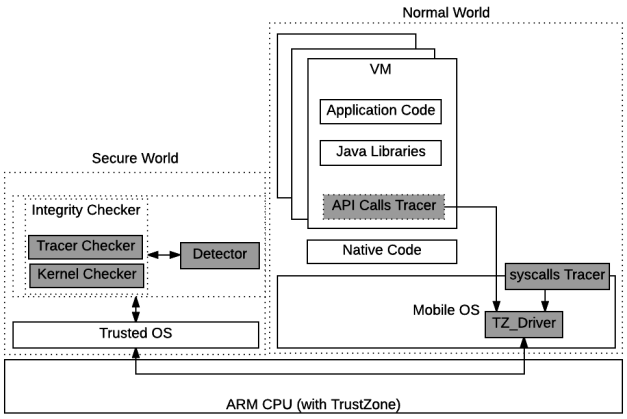

Fig. 1. Architecture of a mobile device running T2DROID (grey boxes).

Dynamic analysis requires applications to be instrumented with inspection code. There are two main instrumentation approaches. *Static instrumentation* involves modifying the application's APK file before the application is installed or executed. With *dynamic instrumentation*, the code is injected into the application process memory by an external process while the application is being launched. This approach does not require modifications to the APK file that might cause reliability issues and would be easier to detect by malware.

For these reasons, the *API calls tracer* module relies on dynamic instrumentation. It runs custom code before and after an Android API function is called. Since every application in Android runs in its own VM, the injected code has access to the VM and is executed inside the VM to hook and call selected Java API methods of the target application. The number of API calls available in Android is large, so it is convenient to limit tracing to a subset of these calls considering if they are used by malware.

The tracer is configured with the number of API calls to collect or the time to collect them. When this threshold is reached, the tracer sends the traces to the T2DROID detector module in the secure world.

2) Syscalls tracer: Android applications may contain native code, so malicious applications may use such code to avoid calling Android API functions and perform malicious operations in a way that is unobservable by the *API calls tracer*. However, such code has to call Android, so we observe its behavior by tracing syscalls.

Android contains a modified version of the Linux kernel. Syscalls are the fundamental API that allows applications to call the OS kernel, usually not called directly but through a library like *glibc*. The Android kernel provides a set of a few hundred syscalls. Examples are syscalls to perform file operations (`open`, `read`, `write`, `close`), process operations (`fork`, `exec`), and network operations (`socket`, `connet`,

`bind`, `listen`, `accept`). Therefore, capturing and analyzing the syscalls performed by an application may provide information about accesses to the file system and network, communication with other processes, etc.

The *syscalls tracer* intercepts and logs syscalls being made by a running application using the `ptrace` syscall. T2DROID uses the number of calls to each syscall to analyse the behaviour of the application. Similarly to the *API calls tracer*, the *syscalls tracer* is configured with the number of syscalls to collect, or the period to collect them.

D. Feature Selection

Unlike the previous and the next, this section does not present a component of the T2DROID architecture, but explains an important aspect of the *detector*.

Features are measurable characteristics of a certain phenomenon and play a crucial role in machine learning. In our case, the features correspond to function calls and characterize the behavior of an application (the phenomenon). T2DROID aims to classify applications in two classes – malicious or not – based on a machine learning classifier. This classifier uses a vector of features to assign an application to one of the two classes. The selection of which features to use is important for the detection mechanism to give good results. Redundant or irrelevant features may present problems such as misleading the learning algorithm, or increasing model complexity and run time.

T2DROID uses a vector composed of two types of features: those related to calls to the Android APIs and those related to syscalls. There is one feature per API function and per syscall. The features of the first set (Android APIs) are binary, i.e., they take value 1 if the application made that call, otherwise they take value 0. The features of the second set (syscalls) take a value that is an integer equal or greater than zero, corresponding to the number of times the syscall was issued by the application. We make this distinction because we assume the number of calls made to one among the large number of API calls is not very relevant, whereas the number of calls made to the lower number of syscalls is. Our experimental results seem to substantiate this assumption (Section VI).

Among thousands of Android APIs, we identified the sensitive/suspicious API calls as those that are often invoked by malicious applications. We analyzed a large set of malware and benign applications and generated a list of distinct API calls, then extracted those frequently used by malware. This reduced our features to 121 APIs, which is of the same order of magnitude as the number of syscalls. Our features for Android APIs are these calls in the format: `full-class-name;method`. Some examples are in Fig. 2. For the syscalls we considered as features all the syscalls, instead. The name of the feature is the name of the syscall (e.g., `open`, `read`).

E. Detector

This section presents the *detector* module of the T2DROID architecture (Fig. 1), which is essentially a machine learning classifier and the core of T2DROID. It runs in the secure world.

```
android.telephony.TelephonyManager;getPhoneType
android.telephony.TelephonyManager;getNetworkOperator
android.app.SharedPreferencesImpl;getInt
java.io.FileOutputStream;FileOutputStream
android.app.SharedPreferencesImpl;getBoolean
java.security.MessageDigest;update
java.io.File;mkdir
```

Fig. 2. Examples of API call features extracted from a trace.

There are three important phases of the life cycle of the classifier to consider. The first is the selection and training of the classifier, which we have done and report in this paper. The second is the use of the classifier at runtime, which we validated experimentally. The third is the re-training of the classifier during the life cycle of T2DROID, which we only briefly explain as it is a repetition of part of the first phase.

The first phase is the selection and training of the classifier. This phase is not done inside the device. We first picked a balanced dataset of malicious and benign applications (Section V). Then, we analysed all these applications with VirusTotal, an online malware scanning tool, in order to confirm that they were indeed malicious/benign. Next, we extracted feature vectors from all the applications. These vectors were then provided to a set of machine learning classifiers available in the Weka tool [29], e.g., kNN and SVM, and their detection effectiveness was compared using different metrics. The best classifier was then implemented in the *detector* module and trained with the same dataset (details in Section V).

The second phase is the use of the classifier at runtime to analyse traces. The *tracers* provide the *detector* with traces that it transforms into a vector of features. Then the classifier classifies this vector as characterizing a malicious or benign application. Experimental results are in Section VI.

The third phase consists on re-training the classifier. As malware evolves, we expect the features selected and the training done to become inadequate and the performance of T2DROID to decrease with time. Therefore, the classifier has to be re-trained periodically. Similarly to the training phase, re-training is not done in the device. Re-training involves selecting again the features to be used and repeating the training phase. The existing instances of the T2DROID service running in mobile devices will have to be updated securely using a scheme similar to those used by anti-virus software, e.g., using the cloud [30]. If necessary, the classifier may also be changed, but this is more complicated than updating the classifier configuration as it involves changing its code.

F. Integrity Checker

This section presents the *integrity checker* module of Fig. 1. The components of T2DROID that run in the normal world, *tracers*, are vulnerable to malware that infects this environment. Moreover, the behavior of these components may be compromised if the mobile OS is infected. Therefore, it is important to check the integrity of the *tracers* and the Android kernel, i.e., if they were modified. For this purpose, T2DROID includes two integrity checker modules in the secure world.

The *tracer checker* verifies the integrity of the code of the tracer modules using a hash function (not the data, that changes). The checker stores a hash of each of the modules running in the normal world when they are in a clean state. Then, at runtime, it calculates a hash of every module and compares it with the hashes that are stored. If they match the check is successful, otherwise it fails. This check is possible because the secure world can access the resources of the normal world, as previously mentioned.

The *kernel checker* does something similar but for the kernel. It calculates a hash of the kernel code memory pages and compares it against a hash calculated when the system was first executed. 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 `System.map` file and translates this address to the secure world address space before evaluating the hash value.

The two checker modules are executed whenever T2DROID is requested to analyse an application, before the extraction of the traces begins. However, there is the risk of malware waiting for the trace extraction to begin, then modify the tracers or the kernel. A solution for this is to check again the integrity when the extraction of the traces ends, but this still leaves the system vulnerable to a race [31] in which the malware modifies the victim component twice. Such attacks are not entirely avoidable, but can be made extremely difficult by repeating the checks.

IV. T2DROID IMPLEMENTATION

We implemented a prototype of T2DROID on an i.MX53 QSB development board. The board is equipped with a Cortex-A8 single core 1 GHz processor with TrustZone and 1 GB DDR memory. Unlike most commercial TrustZone-enabled smartphones, the i.MX platform places no restrictions on the use of the secure world.

A. Secure World Runtime Environment

Genode is a framework for building special-purpose OSs [32]. It provides a collection of small building blocks (e.g., kernels, device drivers, and protocol stacks). Since requirements vary, Genode can reduce system complexity for each security-sensitive scenario. Due to its ability to generate a small TCB, Genode is an appealing foundation for an OS designated to run on the secure world. Genode version 15.11 introduced a TrustZone virtual machine monitor (VMM) demo for our board. It executes a custom kernel (*base-hw*) in the secure world, while a guest OS runs in the normal world. We used this demo as a starting point to implement our prototype. For context switching between the two worlds, Genode provides a VM session interface in Genode's core that enables the VMM to save the CPU state (registers and stack), initiates a switch to the normal world using the SMC instruction, and restores the state after it returns.

In the normal world, we run Android for the i.MX53 series from Adeneo/Witekio [33]. We use the Linux/Android kernel modified by Genode Labs for this board. The Linux/Android kernel is modified to prevent the normal world from directly accessing resources such as hardware and memory that are set as secure within the central security unit (CSU) initialization. We run the *TZ_Driver* driver in the kernel to allow code in the normal world to issue an hypercall to exit the normal world and trap into the secure world, using the SMC instruction.

B. T2DROID Components

We start by explaining the implementation of the components that run in the normal world – the *tracer* modules – then explain those that run in the secure world.

For implementing the *API calls tracer* module we used the *Xposed framework*, which allows modifying the behavior of Android applications without modifying their code and the APK file [34]. There are alternative frameworks, *Cydia Substrate* and *Frida*, but Xposed seemed to be the most stable, with a support community and frequent updates. All application processes in Android have as parent a process called *Zygote*, i.e., every application is created as a fork of that process. *Zygote* is the first process started by `init.rc` after the device boots. This process is launched by the `app_process` executable (`/system/bin/app_process`), which loads all necessary classes and resources. Xposed takes advantage of this mechanism and replaces the *app_process* file with a modified one. Whenever a new VM is created, this *extended app_process* adds an additional jar file (`XposedBridge.jar`) to the classpath. Xposed allows adding hooks to Android API functions and extending them with our own custom code written as a module that is loaded by the *extended app_process* when the target application process is launched. Our *API Calls tracer* is a module of this kind that records the invocation of Android API function calls in a log data structure. The log is later processed and analyzed by the *detector* module in the secure world.

The *syscalls tracer* was implemented based on *strace*, a debugging tool for Linux and related OSs. The *strace* tool can be used to trace the syscalls made by a process. It can be considered to be a user space interface to the `ptrace` syscall. The T2DROID *syscalls tracer* module records the name of each system call, the arguments passed to the system calls and their return values. After the application to be analyzed has been started, a *syscalls tracer* instance is launched and attached to the VM running the application. The above-mentioned data is logged to a data structure to be processed and analyzed by the *detector* module in the secure world.

The *detector* is the main component of T2DROID executed in the secure world. It receives the traces from the two *tracer* modules and runs the detection algorithm. It was implemented based on the Java code of the algorithm in the Weka tool.

The implementation of the *integrity checker* in the secure world follows what was explained in Section III-F. This module needs access to the normal world memory. The TrustZone configuration within Genode partitions the RAM between the secure world and the normal world, so a program called *tz_vmm* in the secure world is able to request the normal worlds RAM via an IOMEM session. The memory is mapped

as uncached to the secure worlds address space, thus the normal world memory can be accessed by the *integrity checker* module in the secure world. We also configured the Android files system partitions to be accessed by the secure world, so the *integrity checker* module can access the files related to the tracer modules in the normal world to verify their integrity.

V. Selection and Training of the Classifier

This section explains how the detection algorithm was selected and trained. For this purpose, we collected 80 Android malware samples up to 3 years-old (2014-16) from the Contagio mobile repository [35]. These samples belonged to 21 different malware families, e.g., FakeInstaller, DroidKungFu, and Opfake. For benign applications, we downloaded from Google Play Store 10 recent applications selected randomly from 8 different categories. Then, we verified these applications with VirusTotal to ensure that no anti-virus product recognizes it as malware. This gave us a balanced dataset of 160 applications, half malicious, half benign.

To extract the features, we obtained execution traces by executing all these applications. For this purpose, we used Android Monkey [36] to generate different kinds of events for the application. Monkey is a program running on Android provided by the Android SDK, which automatically feeds an application with pseudo-random streams of user events such as clicks, key presses and touches, as well as a number of system-level events. We executed and traced each of the applications using Monkey to generate 500 events with a delay of 1 second between each pair of events, leading to more than 8 minutes of execution, which is enough to extract reasonably long traces (100-200 KB for syscalls and 1-3KB for API calls). This time is a tradeoff between how long we monitor the application (the lower the time the better) and how much of its behavior we observe (the larger the time the better). Due to the complexity of using Monkey and executing these experiments on the board, we executed the applications in the Android emulator [37], set to emulate an ARM CPU.

For each application, we then extracted the features. For the API calls, we extracted 121 values 1 (call issued) or 0 (not issued). For the syscalls, we extracted the number of calls made to each. We assigned a class to each feature vector, M for malware and B for benign application. We created three feature vector sets: (1) API calls only (only features extracted from API calls traces); (2) syscalls only (only features extracted from syscall traces); and (3) all features extracted (both API calls traces and syscall traces). The purpose of having variants with these three sets is to allow compare the results and understand if there is a benefit in using more than one of the traces.

The feature vector sets were then inserted in Weka. Weka allows training different machine learning classifiers with a set of feature vectors and obtaining metrics of their performance. For each feature set, we conducted experiments using six widely used machine learning classifiers: Bayes Net, Naive Bayes, SMO (SVM), Ibk (kNN), J48, and Random Forest. In each experiment, we used 10-fold cross validation to evaluate the classifiers without having a training and a testing dataset. Our sets of applications (both malicious and benign) were divided into 10 different sets/groups. In each of the 10 rounds, one set of malware and benign applications was used as the testing datasets and the remaining 9 as the training datasets.

For each classifier and feature set, we measured five common performance metrics. Consider that *TP* (*True Positives*) is the number of malware samples correctly identified as such, *FN* (*False Negatives*) is the number of malware samples classified as benign applications, *TN* (*True Negatives*) is the number of benign applications correctly identified, and *FP* (*False Positives*) is the number of benign applications identified as malware. We consider the following metrics:

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

The two most interesting metrics are *precision* (*Precis.*), which measures the confidence we can have when T2Droid says an application is malicious it is indeed so, and *accuracy* (*Accur.*), which measures the correctness of the mechanism in terms of the rate between correct results and the total.

The results of this evaluation are shown in Table I. Comparing the precision and accuracy it is possible to conclude the following. First, the best classifier with API calls only is SMO, which is an implementation of support vector machines (SVM). Second, with syscalls only the best classifier is Random Forest, and the results are slightly worse than with API calls. Third, the best performance was obtained with both API calls and syscalls with the Ibk algorithm, an implementation of the *k-nearest neighbors* (kNN) algorithm, with an accuracy of 0.98 and a precision of 0.99, although the accuracy was always 0.85 or greater and the precision 0.87 or greater independently of the classifier used. This lead us to the conclusion that *the detector should use the two types of traces and use Ibk/kNN as classification algorithm.* Therefore, this was the algorithm implemented in T2Droid.

VI. Experimental Evaluation

This section presents the experimental evaluation of T2Droid in terms of detection and performance.

A. Detection

Our experimental evaluation of the detection performance of T2Droid was presented in the previous section, together with the study for the selection of the classifier. The experimental results for the selected algorithm (Ibk/kNN) are shown in bold in Table I, showing that it had excellent performance in all metrics, with all equal to 0.98 or 0.99, except FPR that was 0.02 (but in this case values close to 0 are better).

Receiver operating characteristics (ROC) curves are a well-known tool to visualize the performance of classifiers. Therefore, we plotted the ROC curve for the T2Droid detector in Fig. 3. Moreover, we plotted two curves for detectors with the same algorithm but with API call features only and systcall

TABLE I
EVALUATION OF 6 CLASSIFIERS WITH 160 APPLICATIONS AND 3 FEATURE VECTOR SETS.

	API calls only					syscalls only					API calls and syscalls					
	Accur.	**TPR**	**FPR**	**Precis.**	**Fm.**	**Accur.**	**TPR**	**FPR**	**Precis.**	**Fm.**	**Accur.**	**TPR**	**FPR**	**Precis.**	**Recall**	**Fm.**
BayesNet	0.84	0.84	0.16	0.86	0.84	0.91	0.91	0.09	0.91	0.91	0.95	0.96	0.05	0.96	0.96	0.96
NaiveBayes	0.78	0.78	0.22	0.78	0.78	0.76	0.77	0.24	0.78	0.77	0.85	0.86	0.15	0.89	0.86	0.85
SMO (SVM)	**0.98**	**0.99**	**0.02**	**0.99**	**0.99**	0.86	0.87	0.13	0.87	0.87	0.97	0.97	0.03	0.97	0.97	0.97
Ibk (kNN)	0.94	0.94	0.06	0.95	0.94	0.91	0.91	0.09	0.93	0.91	**0.98**	**0.99**	**0.02**	**0.99**	**0.99**	**0.99**
J48	0.92	0.93	0.08	0.94	0.93	0.92	0.93	0.07	0.93	0.93	0.86	0.87	0.13	0.87	0.87	0.87
RandomForest	0.97	0.97	0.03	0.97	0.97	**0.94**	**0.94**	**0.06**	**0.95**	**0.94**	0.94	0.94	0.06	0.94	0.94	0.94

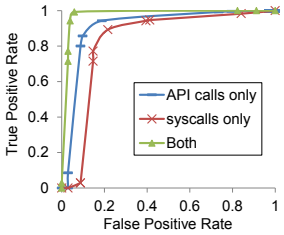

Fig. 3. ROC curve for the detector in T2DROID (Both) and the detector with only one of the types of traces (the other two lines).

TABLE II
TIME FOR TRACE TRANSFER, FEATURE VECTOR PREPARATION, AND CLASSIFICATION.

Num. events	**syscall traces (B)**	**API call traces (B)**	**Trace transf. (s)**	**Feat. vect. prep. (s)**	**Classif. (s)**	**Total (s)**
100	13.2K	488	0.000012	0.082	2.14	2.22
200	69.3K	1.3K	0.000036	0.17	2.21	2.38
500	*157.2K*	*2.1K*	*0.000053*	*0.26*	*2.21*	*2.47*
1000	252.9K	2.6K	0.000065	0.34	2.23	2.57
1500	531.8K	4.4K	0.00011	0.62	2.22	2.84

features only. The ROC curve is obtained by plotting the TPR versus the FPR with various threshold settings. The figure confirms that Ibk/kNN is indeed a good classifier when both types of features are used, as the curve rises fast to values close to 1, then stays there. Moreover, the figure shows that the results with both types of features are better than the results with API calls only, which are better than syscalls only.

B. Performance Overhead

As mentioned above, we did not measure the time of the whole analysis as the time to extract the traces is configurable. However, we measured the total time required for the tracer modules to send trace data (both API calls traces and syscalls traces) to the detector module. This time includes the performance delay introduced by the context switching, and copying and sending of data between the two worlds using the shared buffer. In addition, we measured the time for the detector module to prepare a feature vector from the trace data and classify the examined application as malicious or benign (with the kNN classifier). For this, we used Monkey to generate different kinds of events and run the examined application. We repeated this experiment for different numbers of events. The results are in Table II. For each number of events, the total time to complete the above operations is shown in the last column of the table. We emphasize the row for 500 events as this was the case considered in the detection experiments.

TABLE III
TIME TO DO INTEGRITY VERIFICATION.

File name	**Size (KBytes)**	**Time (ms)**
`XposedBridge.jar`	115	71.62
`app_process`	22	13.54
API calls tracer module	2909	2511.21
syscalls tracer module	1126	829.55
Android kernel	8324	932.66
Total	*12496*	*4358.58*

We also evaluated the performance overhead incurred by the integrity verification of the components of T2DROID executed in the normal world. The *integrity checker* modules verify the integrity of the tracer modules and the Android kernel by calculating their SHA-256 hash and comparing them against their known-good values. The times for these operations are shown in Table III. The table shows both the size and the time to check the integrity of the modules. For the *API calls tracer*, we show separately the values for two modules provided by the *Xposed* framework but that are also important for the integrity of the system: `XposedBridge.jar` and `app_process`. The last line shows the total. It is possible to observe that the time needed to check the integrity is slightly above 1s in our board.

VII. RELATED WORK

As mentioned in the introduction, there is research on both static analysis [11], [12] and dynamic analysis [13]–[19] mechanisms for Android and mobile devices. In this paper we use *dynamic analysis*. Some work on dynamic analysis modifies Android (kernel or middleware) to control the information flow in real-time, e.g., in order to prevent the flow and exposure of privacy-sensitive data [13]–[15], [20]. This is an interesting approach but requires human knowledge about the ways malware violates security properties, whereas in the paper we are interested in using machine learning to extract such knowledge automatically. It also requires modifying Android, which is something we want to avoid.

Another trend on dynamic analysis is detecting malware evaluating calls [16], [38], [39]. An issue is the semantic gap between syscalls and high level behavior like sending an SMS message. An alternative that greatly reduces this gap is to trace Android API calls, which is the approach followed in some recent work [17]–[19]. T2DROID follows this trend but evaluates both syscalls and Android API calls. Moreover,

T2DROID is a system, whereas existing work proposed only detection approaches based on machine leaning.

As mentioned before, in terms of security these dynamic analysis mechanisms have the limitation of running in the same environment as Android, exposed to some attacks [21]–[23]. In this work we leverage *ARM TrustZone* to protect our mechanism. TrustZone has been adopted to secure a number of services, none similar to T2Droid [40]–[42].

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] R. Chang, "10 alternative Android app stores," https://code.tutsplus.com/articles/10-alternative-android-app-stores–cms-20999, 2014.
[3] P. Yan, "A look at repackaged apps and their effect on the mobile threat landscape," Jul. 2014, trendLabs Security Intelligence Blog.
[4] 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.
[5] 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.
[6] 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.
[7] 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.
[8] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012, pp. 113–124.
[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proceedings of the 13th International Conference on Information Security*, 2011, pp. 346–360.
[10] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 51–60.
[11] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 235–245.
[12] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 627–638.
[13] 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.
[14] 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.
[15] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, 2013, pp. 1043–1054.
[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for Android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011, pp. 15–26.
[17] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying Android malware using dynamically obtained features," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 9–17, 2015.
[18] D. Su, W. Wang, X. Wang, and J. Liu, "Anomadroid: Profiling android applications' behaviors for identifying unknown malapps," in *Trustcom/BigDataSE/ISPA, 2016 IEEE*, 2016, pp. 691–698.
[19] Z. Lin, R. Wang, X. Jia, S. Zhang, and C. Wu, "Classifying Android malware with dynamic behavior dependency graphs," in *Trustcom/BigDataSE/ISPA, 2016 IEEE*, 2016, pp. 378–385.
[20] M.-K. Yoon, N. Salajegheh, Y. Chen, and M. Christodorescu, "Pift: Predictive information-flow tracking," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 713–725.
[21] N. Leavitt, "Mobile phones: the next frontier for hackers?" *IEEE Computer*, vol. 38, no. 4, pp. 20–23, 2005.
[22] 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/.
[23] K. Kwang, "Android flaw can disable, corrupt AV tools," Sep. 2011, ZDNet, http://www.zdnet.com/article/android-flaw-can-disable-corrupt-av-tools/.
[24] 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.
[25] ARM, "ARM security technology, building a secure system using TrustZone technology," http://www.arm.com, 2009.
[26] ENISA, "Appstore security, 5 lines of defence against malware," 2011.
[27] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *USENIX Security Symposium*, 2011.
[28] National Computer Security Center, "Trusted computer systems evaluation criteria," Aug. 1983.
[29] Weka 3, http://www.cs.waikato.ac.nz/ml/weka/.
[30] L. Subramanian, G. Q. Maguire Jr, and P. Stephanow, "An architecture to provide cloud based security services for smartphones," in *27th Meeting of the Wireless World Research Forum*, 2011.
[31] M. Bishop and M. Dilger, "Checking for race conditions in file access," *Computing Systems*, vol. 9, no. 2, pp. 131–152, 1996.
[32] Genode Labs, "ARM TrustZone, an exploration of ARM TrustZone technology," http://genode.org/news/an-exploration-of-arm-trustzone-technology, 2014.
[33] Witekio, "NXP i.MX 53 reference BSP," http://witekio.com/cpu/i-mx-53/.
[34] Xposed framework, http://repo.xposed.info/.
[35] Contagio mobile, http://contagiominidump.blogspot.pt/.
[36] Monkey, https://developer.android.com/studio/test/monkey.html.
[37] Android Developers, "Run apps on the Android emulator," https://developer.android.com/studio/run/emulator.html.
[38] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android application sandbox system for suspicious software detection," in *5th International Conference on Malicious and unwanted software*, 2010, pp. 55–62.
[39] X. Su, M. Chuah, and G. Tan, "Smartphone dual defense protection framework: Detecting malicious applications in Android markets," in *Mobile Ad-hoc and Sensor Networks, 2012 Eighth International Conference on*, 2012, pp. 153–160.
[40] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Čapkun, "Smartphones as practical and secure location verification tokens for payments," in *Proceedings of the Network and Distributed System Security Symposium*, 2014.
[41] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena, "DroidVault: A trusted data vault for Android devices," in *Proceedings of the 19th International Conference on Engineering of Complex Computer Systems*, 2014, pp. 29–38.
[42] 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.