Beyond Kernel-Level Integrity Measurement: Enabling Remote Attestation for the Android Platform

Mohammad Nauman[1], Sohail Khan[2], Xinwen Zhang[3], and Jean-Pierre Seifert[4]

[1] Department of Computer Science, University of Peshawar, Pakistan
recluze@gmail.com
[2] School of Electrical Engineering and Computer Science, NUST, Pakistan
sohail.khan@seecs.edu.pk
[3] Samsung Information Systems America, San José, USA
xinwen.z@samsung.com
[4] Technische Universität Berlin & Deutsche Telekom Laboratories
jean-pierre.seifert@telekom.de

Abstract. Increasing adoption of smartphones in recent times has begun to attract more and more malware writers towards these devices. Among the most prominent and widely adopted open source software stacks for smartphones is Android that comes with a strong security infrastructure for mobile devices. However, as with any remote platform, a service provider or device owner needs assurance that the device is in a trustworthy state before releasing sensitive information to it. Trusted Computing provides a mechanism of establishing such an assurance. Through remote attestation, TC allows a service provider or a device owner to determine whether the device is in a trusted state before releasing protected data to or storing private information on the phone. However, existing remote attestation techniques cannot be deployed on Android due to the unique, VM-based architecture of the software stack. In this paper, we present an attestation mechanism tailored specifically for Android that can measure the integrity of a device at two levels of granularity. Our approach allows a challenger to verify the integrity of Android not only at the operating system level but also that of code executing on top of the VM. We present the implementation details of our architecture and show through evaluation that our architecture is feasible both in terms of time complexity and battery consumption.

1 Introduction

Mobile devices are becoming more powerful and are offering new functionalities that go well beyond the traditional use of cell phones such as making and receiving calls. More and more services are being deployed on these devices leading them to their use as a PC on the go. However, this rapid growth in smartphone usage and their evolving capabilities have made this technology more vulnerable

A. Acquisti, S.W. Smith, and A.-R. Sadeghi (Eds.): TRUST 2010, LNCS 6101, pp. 1–15, 2010.

to today's sophisticated malware and viruses. PandaLabs [1] has identified applications downloaded from the Internet as one of the main causes of propagation of malware on mobile phones.

According to Gartner Research [2], smartphones sales and usage has increased by 12.7% in the first quarter of 2009. One of the driving reasons of this growth is the introduction of open source platforms for mobile devices. In this arena, Android [3] is the most prominent and leading open source platform which has succeeded in attracting a large number of individuals and organizations. In fact, Android OS share in terms of web requests had already surpassed that of Windows Mobile by June 2009 [4]. The growing popularity of Android is attracting more and more enterprises to deploy their custom applications for Android and to allow employees to download data for viewing or editing on their smartphones. On the other hand, the open source nature of Android is also attracting more and more malware writers. Hence, the growing security problems of smartphones are becoming a real concern for users. Service providers need assurance that if sensitive data is released to a smartphone, it will not be compromised due to the presence of a malware on the phone. Similarly, users save highly sensitive information such as their contacts and personal messages on the phone. In case of Android (and other GPS-enabled devices), the phone also has access to real-time information about the owner's location. A compromised device can lead to severe financial losses or even social threats.

To alleviate these problems, there is a need for the creation of a mechanism that can securely establish the trustworthiness of an Android-based device, providing remote parties assurance that the data released to the phone will not be compromised. The traditional approach towards solving this problem is by signing applications as being trustworthy. This approach is followed by many Symbian- and J2ME-based software stacks. A trusted application can perform all tasks, whereas an untrusted application is either sandboxed or severely restricted from accessing any sensitive resource. However, there are several problems with this approach in the context of Android. First, Android does not distinguish applications as being trusted or untrusted – *all applications are created equal*. Secondly, the open source nature of Android means that Android's infrastructure can be changed arbitrarily, thus making any security infrastructure unreliable. Finally, it has been shown in the past [5] that an assurance of trustworthiness of a device cannot be provided through the use of software-based solutions alone. Software is inherently mutable and can be modified to report inaccurate information about the hosting device. To solve this problem, Trusted Computing [6] provides the mechanism of *remote attestation* that allows a *challenger* to establish the trustworthiness of a remote *target* platform. Existing remote attestation techniques mainly aim to measure all the executables loaded on a platform and reporting them to the challenger during attestation. The challenger can then verify, using the reported measurements, whether any of the applications loaded on the platform were malicious. However, these techniques fail to cater to the unique architecture of Android because of the presence of a Virtual Machine (VM) that is responsible for executing all code. As far as the integrity measurement entity

is concerned, the VM is just another executable. Even if the VM is known to be benign, there is no assurance that the code it loads for execution will behave as expected. Note that it has been shown that user-space code (including that executed by a VM) can also lead to severe vulnerabilities in a system [7,8,9].

In this paper, we present an efficient integrity measurement mechanism aimed specifically at Android that allows integrity verification of code loaded on top of the VM as well as that running on the operating system level.

Contributions: Our contributions in this paper are as follows: (1) We design an integrity measurement architecture which ensures that all the executable code loaded on Android is measured, (2) We provide two alternative solutions for the deployment of our integrity measurement mechanism, which cater to different real-world use cases, and (3) We describe the details of implementation of both alternatives and provide evaluation results to show that the technique is highly feasible both in terms of time taken for integrity measurement and battery overhead caused by it.

Outline: The rest of the paper is organized as follows: Section 2 provides real-world use cases for motivating the need for integrity measurement and gives a brief summary of the background on Android. In Section 3, we provide the details of our architecture covering the two alternative solutions in 3.2 and 3.3. Section 4 outlines the verification mechanism. Detailed evaluation results are presented in Section 5. Sections 6 and 7 reflect upon pros and cons of our technique and the conclusions drawn respectively.

2 Background

2.1 Motivating Examples

We motivate the need for the measurement of integrity of an Android-based smartphone through the use of two real-world use cases. The first use case is similar to those presented as a motivation for remote attestation in the PC world, whereas the second is more relevant to the personal nature of a smartphone.

Use case #1: Consider an organization that provides its employee – Alice – with a G1 handset running several applications that she might require for carrying out her job responsibilities. The employer, being the owner of the device, allows Alice to install applications that she might need for her daily use. However, since the organization releases sensitive information to Alice's mobile, it wants to ensure that the integrity of Android is intact and that there is no malicious software or application running on the mobile device.

Use case #2: Emma, on the other hand, is a self-employed IT consultant who has bought her own smartphone running Android. Knowing that a smartphone in general [10] and Android in particular [9,8] is much more likely to be affected by a virus threat, she decides to take preventive measures against such attacks. While the smartphone is better than her old cell phone, it is still dependent on a battery source, and if Emma were to run a dedicated antivirus software on the

device, its battery would drain a lot sooner than she would like. Therefore, she decides to use remote attestation as a virtual antivirus. She remotely attests the integrity of her smartphone periodically and after she installs a new application. This ensures that her mobile device is not running any malicious software while still keeping it free of a battery-hungry antivirus software.

2.2 Android Architecture

Android is an emerging open source platform for mobile devices like smartphones and netbooks. It is not just an operating system but provides a complete software stack including a middleware and some built in applications. Android architecture is composed of different layers, with the Linux kernel layer at the bottom. This layer provides various hardware drivers and acts as a hardware abstraction layer. It is also responsible for memory and power management functionalities of Android. The Android native libraries written in C and C++ sit above the kernel layer. These libraries provide some core functionalities. For example, the Surface Manager libraries are responsible for composing graphics onto the screen, SGL and OpenGL enable graphics processing capabilities, webkit provides HTML rendering and SQLite is used for data storage purposes.

Next is the Android runtime layer which is composed of two principle components namely *Dalvik Virtual Machine* and *Android core libraries*. Android runtime is specifically designed as an optimized environment to meet the requirements of running on an embedded system i.e., limited battery life, CPU speed and memory. Dalvik virtual machine executes its own bytecode represented by *dex files*. The second component of Android runtime is the collection of class libraries written in Java programming language, which contains all of the collection classes and I/O utilities.

Class loaders: Android framework and applications are represented by classes composed of `dex` code. One or more class loaders are used to load these classes from a repository. These class loaders are called when the runtime system requires a particular class to be loaded. All of the class loaders are systematized in a hierarchical form where all requests to child class loaders are first delegated to the parent class loader. The child class loader only tries to handle a request when the parent class loader cannot handle it.

Android comes with several concrete implementations of the abstract class – `ClassLoader` [11] – which implement the necessary infrastructure required by all of the class loaders. Of these, the `PathClassLoader` will be of particular importance to us.

3 System Architecture

In Section 2.1, we presented two real-world use cases for motivating the creation of an integrity measurement system on Android devices. In this section, we present an architecture that provides two levels of granularity, each catering to one of the use cases presented. Figure 1 shows the high-level architecture of our approach.

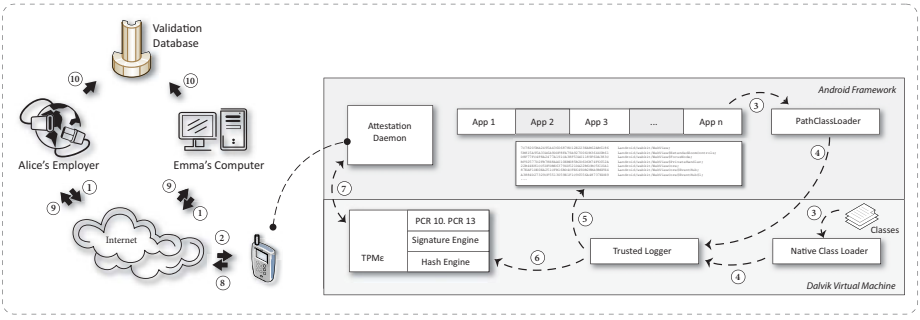

Fig. 1. Android Integrity Measurement Architecture

The attestation challenge begins at Alice's employer's system (or Emma's PC – depending on the scenario). The challenge is sent to the *Attestation Daemon* running on the Android device. On the device, one of the two integrity measurement systems must be in place: (1) Application-level attestation or (2) Class-level attestation. In either case, the measuring entity reports the measurements to a trusted logger that maintains an integrity measurement log and extends a PCR with the hashes of these measurements. When an attestation challenge is received, the attestation daemon reads the log and requests a quote over the PCR in which the measurements have been recorded by the logger. Both of these trust tokens – measurement log and PCR quote – are returned to the challenger as the attestation response. The challenger can then verify the trustworthiness of the platform based on these measurements using a *validation system*.

Both application-level attestation and class-level attestation require the presence of a *root-of-trust*. The chain of trust must be extended from this root-of-trust to the Dalvik VM and then to the measuring entities within the virtual machine. For this purpose, we need: (1) an implementation of a TPM, either hardware or software; (2) a device driver library for communicating with the TPM and; (3) a Trusted Software Stack (TSS) for providing high-level APIs for accessing the low-level functionality of the TPM. Below, we first briefly describe the creation of a minimal subset of the TPM and the TSS that is required for our implementation since a hardware TPM does not exist for mobile phones.

3.1 Chain-of-Trust

For the establishment of a chain of trust, there are two requirements:

1. A root-of-trust that acts as an anchor for the chain. It must be immutable and, according to [5], hardware-based. The TCG has defined a specification for a hardware root-of-trust – called *Mobile Trusted Module* (MTM) [12] – specifically aimed at mobile platforms. To date, no agreed-upon and widely deployed implementation of the MTM exists. We have therefore abstracted away the details of the MTM implementation and built our approach on top of the

TCG specification. This allows us to decouple our integrity measurement architecture from any specific prototype *implementation* and assures forward compatibility by complying with the standards. It should also be noted that since a hardware root-of-trust is currently not available, our implementation cannot, as yet, be deployed in production environments. However, the successful standardization of the MTM and its wide acceptance by the scientific community leaves little doubt that MTM hardware will be made available in the very near future.

2. The second requirement for a chain of trust is making all links in the chain integrity-aware. The BIOS, bootloader and the operating system all need to be modified so that they measure the integrity of every loaded executable before passing control on to it.

Below, we take a look at how we have addressed the aforementioned problems.

Emulating the Trusted Platform Module: One of the most important aspects of our architecture is the presence of a root-of-trust that can securely save the hashes of the measurements and report them to the challenger in a trustworthy manner. The absence of a hardware TPM mandates the creation of a minimal implementation of a software emulator that can act as a prototype until a hardware root-of-trust becomes available. Software emulators of both TPM [13] and MTM [14] already exist. An implementation of MTM has also been proposed recently [15]. However, we decided not to use either of these. The reason is that they are complex softwares that aim to implement the whole TPM/MTM specifications. We, on the other hand, need only protected storage (i.e. PCRs) and the PCR quote operation. Implementing the complete specifications not only gives rise to complexity in the software but also taxes the limited resources of the phone device. We have therefore created a simplified *mini* TPM *emulator* (TPM_ϵ) that provides only these two functionalities and is optimized for use on a mobile device to consume as little computational cycles and battery power as possible.

We implement TPM_ϵ as part of the kernel instead of as a module so that it can measure all the modules loaded by the kernel. TPM_ϵ uses facilities provided by the Linux kernel code for auxiliary operations, such as random number generation.

Each of the entities performing measurements needs to communicate with TPM_ϵ. The communication aspects of each of these entities are discussed in their relevant sections below.

Establishing the Chain-of-Trust: In PC world, the first link in the chain of trust is the BIOS. However, in the case of mobile and embedded devices, there is no BIOS. Device initialization is performed by the bootloader instead. Therefore, the chain of trust in our architecture begins with the bootloader. Moreover, as discussed earlier, no hardware root-of-trust is available on the Android device and consequently, there is no protected storage available for storing the hashes measured before the kernel. Therefore, as yet, the bootloader has to remain outside the chain of trust in our architecture.[1]

[1] We discuss the implications of this aspect in Section 6.

The chain begins at the kernel level with our TPM_ϵ loaded as part of the kernel. Since TPM_ϵ is a part of the kernel itself, it can be used to securely save the hashes of loaded executables. Integrity measurement is performed by Integrity Measurement Architecture that we have ported to the Android kernel. We have tried to keep the changes to IMA at a minimum so as to ensure backward and forward compatibility with IMA code that has now been incorporated in the Linux kernel. However, since our architecture uses TPM_ϵ and not a hardware TPM, we have had to make some changes regarding the communication of the integrity measurement code with the TPM. Other than the aspects concerning the communication with TPM, we have not modified any functionality of IMA. It therefore measures all executables loaded on the Android platform by the Linux operating system. This includes the Android VM as well as any libraries (such as `libdvm.so`, `libandroid-runtime.so` and `libandroid-system.so`). This ensures that all the executables loaded outside the Dalvik virtual machine as well as the native code of Dalvik itself gets measured and stored in the Stored Measurement Log (SML).

Similarly, the semantics of SML are also unmodified. This is because we opt not to interleave the Linux executable hashes with the Dalvik executable hashes but keep the two logs separate. The aggregate up to the point of the Dalvik load is stored in the *Android Measurement Log* (AML).

Once the chain of trust up to the Dalvik virtual machine is established, we provide two alternatives for measurement of code that is loaded on top of the VM. These two alternatives form the core part of our contribution and are discussed at length in the following sections.

3.2 Application-Level Attestation

For coarse-grained attestation of the Android Software Stack that can cater to the requirements of Use case #1 (cf. Section 2.1), we have implemented a binary attestation mechanism that can measure all loaded *applications*. Recall that in the first use case, the employer is only interested in finding out if any malicious application is executing on Alice's phone.

In Android, applications are distributed as `.apk` files that can be downloaded or copied onto the phone and installed through the `PackageInstaller` activity. These *package* files contain the `AndroidManifest.xml` file (that defines the permissions requested by the application), resource files and the `.dex` files that consist of the actual application code. All `.apk` files are stored in the `/system/app` folder in the Android filesystem. Whenever the user starts an application that isn't already loaded, Android looks up the class required for loading that application and calls the `PathClassLoader`. The name of the required class is passed to the class loader that loads the class file from the `.apk` file of that application.

We have inserted an integrity measurement hook in the `findClass()` function of the `PathClassLoader` that ensures that whenever an application gets loaded, the complete `apk` file corresponding to the application is measured and an entry is made to the AML. The hash of the `apk` is extended in PCR-11 to ensure that the log can be trusted at verification time. The implementation of the SHA-1

hashing mechanism is based on the `MessageDigest` algorithm provided by the Java Cryptography Extensions (JCE).

For communicating with TPM_ϵ, the measurement function requires an implementation of the Trusted Software Stack (TSS). As with the TPM and MTM emulators, we have opted not to use any of the existing TSS implementations due to performance concerns. For this coarse-grained measurement, we have implemented a minimal implementation of the TSS specifications – called TSS_ϵ – that allows only two operations: (1) PCR extend – allowing the measurement function to communicate the measured hash to the TPM_ϵ and (2) PCR quote – that allows trustworthy reporting of the PCR values to the challenger. Since the measurement functions operate below the Android application framework layer i.e. in the Java library layer (cf. Section 2.2), TSS_ϵ is implemented as a Java class (`edu.android.aim.TssE`) that exposes two functions for the aforementioned operations – `pcrExtend()` takes a hash and a PCR number as input for extending the PCR and `quote` takes a collection of PCRs, a nonce, an AIK label and the associated authorization secrets as input and returns the quote performed by TPM_ϵ over the PCR values and nonce using the AIK associated with the label. Each PCR extend operation must be matched by an entry made in the AML. This is also implemented as a class in the Java libraries (`edu.android.aim.TrustedLogger`) that exposes two operations – (1) `logEvent()` that creates a new entry in the AML with the provided entry description and hash and; (2) `retrieveLog()` that returns the complete AML. The AML is stored in the filesystem in an unprotected space (`systemdir/aml_measurements`) since its correctness can be ensured through the measurements in the protected storage of TPM_ϵ.

This coarse-grained approach has several advantages in the context of a mobile platform. Firstly, it only requires the measurement of `apk` files of applications that are loaded. For a typical smartphone user, this number is usually quite small. This ensures that the computational requirements for integrity measurement are keep to a minimum. Moreover, the AML is fairly small and thus aids in keeping the communication overhead to a minimum. Likewise, the battery consumption during calculation of hashes is also fairly small. In Section 5, we discuss the performance issues associated with this approach.

The major drawback of attestation at this level of granularity is that it is not complete! It does not measure the system classes which form an essential part of Android's trusted computing base. Ignoring these classes removes the possibility of ensuring that, for example, the Android permission mechanism will be enforced by the mobile device – which in turn reduces the level of trust that can be placed in the correct enforcement of the security mechanisms that are expected by the challenging party. To alleviate this drawback, we have implemented a finer-granular integrity measurement approach as defined in the following section.

3.3 Class-Level Attestation

To cater to fine-grained requirements of attestation for the Android platform, we go a step beyond just measuring the applications that are loaded on the device

and propose a solution that provides completeness in integrity measurement. This level of attestation can measure all executables loaded on top of the Dalvik VM and can thus cater to the requirements of Use case #2.

Class-level attestation aims to measure all executables (i.e. classes) loaded on top of the Dalvik VM. While this approach is similar to IMA in essence, it differs significantly in the semantics of measurement. Moreover, since the loading mechanism of Dalvik is, at its core, different from that of the Linux kernel, our binary integrity measurement has major differences in what and how it measures.

As mentioned in Section 2.2, there are two ways in which classes may be loaded into Dalvik. The mechanism mentioned earlier is that which uses `ClassLoader`s executing on top of the VM itself. These class loaders are themselves classes and thus need to be loaded too. Moreover, there are several classes that are 'system classes' and are required for the proper functioning of Java code (e.g. `java.lang.Object`). These classes cannot be loaded by Java-based class loaders and have to be loaded by the native code in the VM itself. Another issue with `ClassLoader`s is their unrestrictive nature. Applications are allowed to write their own class loaders to load classes from arbitrary sources. For example, an application may write a class loader that reads from a byte stream to load a class. This is substantially unlike the Linux/IMA scenario in which all executables are loaded from the filesystem. It is therefore possible in Linux to measure an executable before it is loaded. In case of Dalvik (or any Java-based VM), this is not always possible due to the potentiality of arbitrary class loaders. It is for this reason that the semantics of our binary attestation are that we measure a class *after* it is loaded but before it can be executed.

In Dalvik, the code responsible for calling class loaders is present in three major files – `Class.c`, `InternalNative.c` and `JNI.c` The two broad categories of classes in Dalvik are *system classes* and (what we informally term as) *standard classes*. These are loaded by `dvmFindSystemClassNoInit()` and `dvmFindClassFromLoaderNoInit()` respectively. Both of these functions are present in `Class.c` and are called from a single point – `dvmFindClassNoInit()`. The 'no-init' functions are responsible for loading classes (either directly or by calling a class loader) *without* initializing them. By placing the integrity measurement hooks in `dvmFindClassNoInit()`, we ensure that (1) the measurement is complete i.e. all the classes that are loaded get measured and (2) that classes are measured immediately after they are loaded and before they can be executed.

After a class is loaded, it is returned to Dalvik in a structure that encapsulates the methods, fields, loader details and other information about the class. This is a highly complex structure and includes pointers to many internal structures representing detailed information about the class. Including all this information in the hash of the class would cause severe performance bottlenecks without adding much to the utility of measurement. In our integrity measurement mechanism, we include only those parts of this structure that may influence the dynamic behavior of the class. We define these parts in three categories:

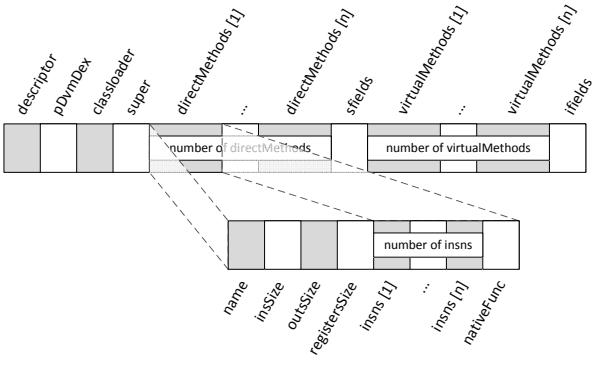

Fig. 2. Subset of a Class Structure for Hash

1. **Meta-information:** This information does not directly influence the execution of a class but is helpful in unique identification of the class. Included in this category are the *descriptor* i.e. fully qualified name of a class, the source `dex` filename, the *class loader* and parent class etc.
2. **Passive entities:** These are static portions of the class that, while non-executable, may affect the execution of the class. Passive entities include *static and instance fields*, *method names* and *instruction and register size* etc.
3. **Executable code:** This is the most important aspect of the measurement and includes the instructions present in the method bodies of a class. Note that, since inner (and anonymous) classes are measured separately, their methods and instructions will be included in their respective measurements and can thus be verified.

Figure 2 shows the precise structure over which the hash is calculated during class-level integrity measurement and Figure 3 shows the integrity measurement log. Each class is represented in the log by its descriptor and is preceded by the hash of the structure described above. Note that since this fine-grained level of integrity measurement computes the hash of *all* loaded classes, it may cause some performance hit but as we discuss in Section 5.2, the performance hit is minimal and with some performance enhancement can be successfully deployed in production settings.

Note that the TSS_ϵ solution proposed in Section 3.2 cannot be utilized at this level of attestation as it operates *above* the VM level, whereas measurement in this fine-grained approach is being done at the VM level. For this level of integrity measurement, we have implemented TSS_ς in the Dalvik VM itself. TSS_ς only performs one operation i.e. saving an entry in the AML. The AML is stored in the same location as in the application-level attestation. It does not provide a function for reading the AML because that functionality is required only at the application level of the Android framework and can be taken care of by the TSS_ϵ's `retrieveLog(()` function. The details of this retrieval operation follow.

```
133A57C0CB942D5F74376BD6A89A3DD98EAB4886 vmaggregate
...
4FC88626E94A631D9FF4BD7C39C57F6EA8847C3F Landroid/widget/AbsListView;
FC060385A2B800175CE68D96AFC4A49E965A8E8F Landroid/widget/AbsListView$CheckForLongPress;
59517950D7280DC0CB4517B40E812D9E2B1BAFB2 Landroid/widget/AbsListView$SavedState$1;
69CEB9E9ED1398EFFF0C2C0705C7D45506481BA1 Landroid/widget/AbsoluteLayout;
457F0C258A8B76B4C03C3A89B1B7BAC8E306ECA1 Landroid/widget/AbsoluteLayout$LayoutParams;
8E84D83A9BFE50BDC7F41714769AB48CE55E208D Landroid/widget/AdapterView;
AE8BB8B2E8585395EB697DC8403C3EC1E2BFF7ED Lcom/android/internal/telephony/Phone;
5CB11877BF82DA663722AFBF19CB3DE2DBC03F3B Lcom/android/internal/telephony/Phone$State;
...
```

Fig. 3. ASCII representation of the Android Measurement Log: Capturing the hash of the class and the class descriptor

4 Verification

Once the attestation tokens i.e. PCR quote and measurement logs are received at the challenger side, they need to be verified to establish the trustworthiness of the remote platform. The first step in the procedure is to validate the digital signature on the quote structure to verify that a genuine TPM vouches for the measurement logs. This is a simple procedure and requires only the knowledge of the AIK which can be provided by a PrivacyCA [16]. Afterwards, the integrity of each loaded executable reported in the measurement log is verified individually. The Android Market [17] is by far the largest and most reliable source of applications. The basic verification mechanism involves creation of a database of *known-good* and *known-bad* hashes of executables retrieved from the Android Market. For instance, currently our database includes information about our own versions of the *Intent Fuzzer* and *Intent Sniffer* tools [9] that may be used to maliciously monitor and/or modify the operation of Android's intent model. If the hash associated with one of these tools is found in the AML reported by the target device, the challenger may conclude that the device is compromised and take preventive measures.

5 Evaluation

In the context of mobile devices, computational complexity and battery consumption are two essential factors that need to be considered when making any changes to the software stack on these devices. We have evaluated both these aspects for the two options of attestation presented in this paper. As a test system, we have taken the Android `cupcake` branch, operating on the HTC G1 handset. Evaluation of the two levels of attestation is presented below.

5.1 Application-Level Attestation

In general, application-level attestation imposes little overhead on both the computational capabilities and battery consumption of Android.

Time: The average time for measurement of an application on our testbed was 1631ms. This is a rather large number but note that we cache the results of

measurement and only measure an application on subsequent loads if it has changed. This caching, coupled with the facts that mobiles are 'always-on' and application `apk`s are unlikely to change frequently, makes the average time fairly acceptable. Moreover, since the largest portion of this time is taken by the hashing algorithm, a faster Java implementation of this function may significantly improve this time.

Log size: Since this coarse-grained attestation only reports the hashes of loaded applications, the log size is extremely small and is dependent only on the number of applications executed on the target device. The size L in bytes of the reported log is given as:

$$L = nL_h + \sum_{i=1}^{n} (L_{a_i}) + L_q + L_s$$

where n is the number of applications loaded, L_h is the size of the application's hash, L_{a_i} is the length of i^{th} application name, L_q is the size of the data structure representing the PCR quote signed by TPM_ϵ and L_s is the size of IMA's SML.

In our evaluation, L_h and L_q were constants (i.e. 20 bytes and 64 bytes) respectively, the average number of applications loaded on the device was 28, the average length of the application name was 11.2 bytes and the size of the SML was 4998 bytes. The total size of the log for application-level attestation was therefore:

$$L = (20 + 11.2) \times 28 + 64 + 4998 = 5935.6$$

which is less than 6KB of data per attestation request for application-level attestation.

Power: Measurement of battery consumption on Android is difficult due to the fact that the battery *charge level* reported by the Android hardware is at a very coarse grained level. Using software for measurement of battery consumption during hash calculation simply yields 'no change' in battery level. However, note that since the attestation techniques only use the CPU and do not tinker with parameters of radio communication, the battery overhead caused by integrity measurement is directly proportional to the time taken. Therefore, using the same arguments as those for time consumption, we can conclude that the battery consumption overhead caused by our integrity measurement mechanism is also bearable.

5.2 Class-Level Attestation

Class-level attestation is performed at a finer-grained level and thus might be expected to have slightly larger overhead in terms of both time and battery.

Time: Figure 4 shows the evaluation results of the time taken for performing this level of integrity measurement. As can be seen, using native C/C++ code for calculating SHA-1 has improved performance by three orders of magnitude. The average time for integrity measurement of a class is 583 μs. Integrity measurement of a few classes took more than a second but these were only around

1% of all the classes measured. Moreover, similar to application-level attestation, caching has been employed for class-level attestation to ensure that after a class has been measured, it is not re-measured on subsequent loads unless it has changed. Moreover, taking only a subset of the structure of the class (cf. Section 3.3) also increases the performance of attestation.

Log size: The length of the log at this fine-grained level of attestation was rather large. The average number of loaded classes during our tests was 1941 and the average length of class names was 35.67. Using the same method of calculation as for application-level attestation, the log size was:

$$L = (20 + 35.67) \times 1941 + 64 + 4998 = 113117.47$$

The log size of around 110KB is not completely insignificant for the a mobile device. However, since we do not require real-time results, attestation can be carried out when the device is connected to the enterprise server or PC through a high-speed connection such as WiFi, thus reducing the time taken for transmission of the log.

Power: Similar to application-level attestation, battery consumption overhead of this finer granular integrity measurement is also directly proportional to the time taken. Moreover, since the time taken by class-level attestation is quite small, battery consumption is also much more acceptable than that for application-level attestation.

6 Discussion

In this paper, we have presented the first attempt at measuring the integrity of the Android platform using the concepts of Trusted Computing. The two levels of granularity presented in the paper both have their pros and cons as discussed earlier. However, there are a few issues that inhibit the deployment of either of the techniques in production environments just yet. First of all, there is the lack of a hardware root-of-trust. A hardware TPM or MTM does not exist for any mobile device. We currently use an emulator for the demonstration of our technique and rely on the assumption that it is only a matter of time before an MTM becomes available for mobile devices. Note that we have designed the architecture in such a way that our technique would be able to use an MTM

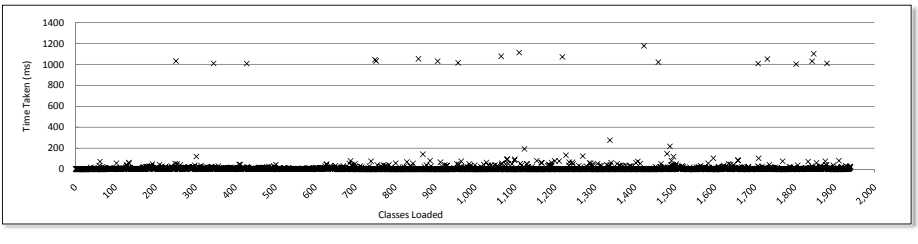

Fig. 4. Class-level Attestation Results

directly without any change to its working. We envision the deployment of our attestation technique probably as a separate trusted sub-system [18] that acts on behalf of either the service provider or the local owner of the device to provide attestation responses.

Finally, we discuss the issue of *time of measurement, time of use race conditions* [19] that was a major concern in the original IMA technique. The issue is that when reading from a filesystem, the file may change after it is measured but before it gets loaded for execution. Since we measure classes or applications only after they are loaded and not from the filesystem, our architecture does not suffer from this drawback.

7 Conclusion and Future Work

The personal and ubiquitous nature of mobile phones poses serious security concerns regarding data that is stored on these devices. Measuring the integrity of a smartphone can ensure that sensitive information accessible to applications running on the device will not be compromised. Android is among today's most popular smartphone platforms. It is backed by a vast majority of industry leaders and is made available as open source, thus leading to wide adoption of this software stack. In this paper, we have proposed the design and implementation of an integrity measurement mechanism aimed specifically at the unique architecture of Android's software stack. We have described our architecture at two levels of granularity catering to different real world use cases. We have shown our architecture to be efficient both in terms of time complexity and battery consumption – two critical factors for any architecture targeting mobile devices.

One of the more important usages of our attestation technique, that we can foresee, is for ensuring 'copy protection' of paid applications for Android phone. Paid applications that are not allowed to be moved from one device to another are protected by the Android system. However, due to the presence of 'rooted' phone devices, it is possible for a malicious user to bypass copy protection [20]. Using our attestation technique before releasing a copy-protected application may provide assurance to Android Market that the target device is in a trusted state and will thus enforce copy protection as expected. Formalizing the semantics and procedure of this mechanism forms part of our future work.

References

1. PandaLabs: PandaLabs Q1 2008 report (2008), http://pandalabs.pandasecurity.com/blogs/images/PandaLabs/2008/04/01/Quarterly_Report_PandaLabs_Q1_2008.pdf
2. Gartner Research (2009) Press Release, http://www.gartner.com/it/page.jsp?id=985912
3. Google: Android Home Page (2009), http://www.android.com.
4. AdMob Mobile Metrics: Mobile Metrics Report (June 2009), http://metrics.admob.com/2009/07/june-2009-mobile-metrics-report/.

5. Pearson, S.: Trusted Computing Platforms: TCPA Technology in Context. Prentice Hall PTR, Upper Saddle River (2002)
6. TCG: Trusted Computing Group (2010), http://www.trustedcomputinggroup.org/
7. Zovi, D.A.D.: Advanced Mac OS X Rootkits. In: Black Hat Technical Security Conference USA (2009), https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html
8. Miller, C., Mulliner, C.: Fuzzing the Phone in your Phone. In: Black Hat Technical Security Conference USA (2009), https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html
9. Burns, J.: Exploratory Android Surgery. In: Black Hat Technical Security Conference USA (2009), https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html
10. Evers, J.: Russian Phone Trojan Tries to Ring Up Charges – Zdnet Australia (2006), http://www.zdnet.com.au/news/security/soa/Russian-phone-Trojan-tries-to-ring-up-charges/0,130061744,139240795,00.htm
11. Google: Android Abstract ClassLoader (2009), http://developer.android.com/reference/java/lang/ClassLoader.html
12. Mobile Phone Work Group Mobile Trusted Module Overview Document, http://www.trustedcomputinggroup.org/resources/mobile_phone_work_group_mobile_trusted_module_overview_document
13. Strasser, M., Stamer, H., Molina, J.: Software-based TPM Emulator, http://tpm-emulator.berlios.de/
14. Ekberg, J., Kylaanpaa, M.: Mobile Trusted Module (MTM)–An Introduction (2007)
15. Ekberg, J.E., Bugiel, S.: Trust in a Small Package: Minimized MRTM Software Implementation for Mobile Secure Environments. In: STC 2009: Proceedings of the 2009 ACM workshop on Scalable trusted computing, pp. 9–18. ACM, New York (2009)
16. IAIK: About IAIK/OpenTC PrivacyCA (2010), http://trustedjava.sourceforge.net/index.php?item=pca/about.
17. Google: Android Market (2009), http://www.android.com/market.html.
18. Schmidt, A., Kuntze, N., Kasper, M.: On the deployment of Mobile Trusted Modules. Arxiv preprint arXiv:0712.2113 (2007)
19. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: SSYM 2004: Proceedings of the 13th Conference on USENIX Security Symposium (2004)
20. Oberheide, J.: A Look at a Modern Mobile Security Model: Google's Android Platform. In: Annual CanSecWest Applied Security Conference (March 2009), http://jon.oberheide.org/research/