Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation

Alexandre Bartel	University of Luxembourg / SnT / SERVAL, Luxembourg
Jacques Klein	University of Luxembourg / SnT / SERVAL, Luxembourg
Martin Monperrus	University of Lille / INRIA / ADAM, France
Kevin Allix	University of Luxembourg / SnT / SERVAL, Luxembourg
Yves Le Traon	University of Luxembourg / SnT / SERVAL, Luxembourg

22 May 2012

978-2-87971-111-9

Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation

Alexandre Bartel, Jacques Klein,
Kevin Allix, Yves Le Traon
University of Luxembourg, SnT
Luxembourg, Luxembourg
firstName.lastName@uni.lu

Martin Monperrus
University of Lille
INRIA
Lille, France
martin.monperrus@univ-lille1.fr

ABSTRACT

In this paper we claim that a widely applicable and efficient means to fight against malicious mobile Android applications is: 1) to perform runtime monitoring 2) by instrumenting the application bytecode and 3) in-vivo, i.e. directly on the smartphone. We present a tool chain to do this and present experimental results showing that this tool chain can run on smartphones in a reasonable amount of time and with a realistic effort. Our findings also identify challenges to be addressed before running powerful runtime monitoring and instrumentations directly on smartphones. We implemented two use-cases leveraging the tool chain: FineGPolicy, a fine-grained user centric permission policy system and AdRemover an advertisement remover. Both prototypes improve the privacy of Android systems thanks to in-vivo bytecode instrumentation.

1. INTRODUCTION

Android is one of the most widespread mobile operating system in the world accounting 48% market share [5]. More than 300 000 Android applications available on dozens of application markets can be installed by end users. On the official market of Google (Google Play, formerly AndroidMarket), more than 10 000 new applications are available every month.[1] For the end user, downloading an application on his smartphone is similar to choosing an apple on an apple tree: he only sees the surface and has no evidence that there is no worm in it. Unfortunately there are many worms of different kinds waiting for entering smartphones such as malware leaking private data and adware calling premium-rate numbers.

In this paper we claim that a widely applicable and efficient mean to fight against malicious mobile applications is: *1) to perform runtime monitoring and interception; 2) by instrumenting the application bytecode and 3) in-vivo, i.e. directly on the smartphone*. We present a tool chain to do this and present experimental results showing that this tool chain can run on smartphones in a reasonable amount of time and with a realistic effort. Before further introducing our contribution let us defend our key claim.

Why performing runtime monitoring and interception? Runtime monitoring consists of observing the behavior of an application during execution, for instance by collecting certain metrics, or by intercepting all exchanges at the interface between the application and the rest of the system. Runtime monitoring is often used to achieve better privacy and security. Privacy can be improved for instance by tainting sensitive data [8], or for Android, by enforcing a finer-grained permission model. Security can be improved for instance by detecting anomalies in I/O behavior [13]. In this paper, we discuss two case-studies involving runtime monitoring and interception, including an implementation of a fine-grained permission model on top of the Android software stack as proposed in [19] in order to maximize privacy.

Why performing bytecode instrumentation? There are at least two manners to perform runtime monitoring and interception: modification of the Android software stack or bytecode instrumentation. Modification of the software execution stack consists of altering the operating system or the core libraries to intercept the required information. On Android, it means changing the underlying kernel, the Dalvik virtual machine or the Android framework. Unless convincing the Android consortium, this is rather limited in deployment since normal end-users have neither the rights (jailed phones) nor the ability to do so. Bytecode instrumentation is one of the lightest way to perform runtime monitoring on top of execution platform that can not be modified. In the context of a fine-grain policy enforcement for improving privacy, we are able – thanks to bytecode instrumentation – to enforce a fine-grained permission model of applications already deployed on Android smartphones without any modification of the Android software stack.

Why performing in-vivo instrumention directly on smartphones? Bytecode instrumentation can be done inside an app market, on a PC or even in the cloud. All of them have fundamental limitations. A key point of the success of Android is the sheer diversity of Android markets. Why should users bind themselves on apps of a single market and trust a single monopolistic instrumentation and detection process? About a PC-based instrumentation process, are users able to transfer an app and instrument it on their own PCs? would they be ready to wait hours or days for getting the instrumentation PC before safely using an instrumented app? We think that these are key blocking factors. Let us now consider a cloud-based service providing bytecode instrumentation. This may sound great but this solution is very much hampered by legal considerations: many countries forbid distributing binary applications to third-party services (e.g. France), similarly to certain terms of service of several markets (e.g. Google Play for Android). To us, an instrumentation process that is installed once and directly runs on smartphones is a way to overcome all these issues at once.

However, to our knowledge, there is no proposal of such an instrumentation process on Android nor there is a study describing

[1] http://www.appbrain.com/stats/number-of-android-apps

whether it is feasible to run it on real apps in a reasonable amount of time.

This is the contribution of this paper:

- Section 2 describes a tool chain for instrumenting Android applications directly on smartphones.
- Section 3 describes the design and concrete implementation of kinds of bytecode instrumentation for the security and privacy of smartphones.
- Section 4 demonstrates the feasibility of running the whole tool chain in a reasonable amount of time.
- Section 5 discusses the related work and Section 6 concludes the paper.

2. TOOLCHAIN FOR IN-VIVO BYTECODE INSTRUMENTATION

This section presents a toolchain for performing bytecode instrumentation of Android apps in vivo, i.e. directly on smartphones.

2.1 Android Apps in a Nutshell

Android applications are written in Java, compiled into Java bytecode and finally converted to Dalvik bytecode, a bytecode format optimized for embedded devices. An Android application is a signed zip file (called apk or AndroidPacKage file) containing the Dalvik executable, the `AndroidManifest.xml` (application metadata), data (e.g. images), and the public key needed to check the provided signatures of all files.

2.2 Requirements

Instrumenting and repackaging an Android application so that it can run again is not straightforward. It consists of extracting the executable code from the application code, analyzing and instrumenting it, rebuilding a new working android application and signing it again, since the OS requires code to be signed.

We now devise a toolchain for this, based on the following requirements:

1. The Android OS must be unmodified (for the sake of a broad applicability, see intro);
2. The Dalvik virtual machine that runs Android applications must be unmodified, in particular in terms of configuration values such as the maximum heap size (for the sake of a broad applicability, see introduction);
3. The hardware that is used to instrument bytecode must be representative of common smartphones on the market.

2.3 Toolchain

To execute a whole analysis process of Android applications on smartphones, one needs to: 1) Extract code from Android application apk files; 2) Modify the extracted code with bytecode manipulation tools; 3) Rebuild a new Android application containing the modified code. Each step of this process requires a certain amount of time to be performed.

Those three steps can decomposed in five elementary steps, as shown on Figure 1: i) Extracting and converting the Dalvik bytecode into Java bytecode(step a-b), ii) Manipulating the bytecode (steps b-c), iii) Translating this representation back to Dalvik bytecode (step c-d), iv) Rebuilding a new apk file (step d-e) and v) Finally signing all files with a new private key (step e-f). Let us now discuss the tools that are used in each step.

i) Extracting the Dalvik bytecode..

The first step, as shown in Fig. 1.(a-b), is to extract the `classes.dex` file from the apk file and convert it to Java bytecode classes that can be analyzed with standard unmodified Java bytecode analysis toolkits. For this step, we use the tool `dex2jar`[2].

ii) Instrumenting the bytecode..

In this step, we experiment with two different tools.

ii.a) Soot. Classes are transformed to Jimple with the Soot analysis toolkit. Soot [20] is an open-source analysis toolkit for Java programs. It operates either on Java source code or bytecode. It allows to analyze and transform programs. For instance, an intraprocedural flow analysis could determine if a variable can be *null* at some point in the code. Soot can also perform different callgraph analyzes, useful for specific bytecode instrumentation. Most analyses and transformations in Soot use an internal representation called Jimple. Jimple is a simple stack-less representation of Java bytecode We ported Soot for Android system by converting its Java bytecode to Dalvik and creating a wrapper Android application.

ii.b) ASM. We experienced that Soot is sometimes slow and requires a lot of resources (especially memory). Thus, we also run ASM for bytecode instrumentation. ASM [3] is a Java bytecode engineering library. One of its characteristics is that it is lightweight hence more suitable for running on systems constrained in term of memory or processing resource. It is primarily designed to manipulate and transform bytecode although it can also be used to perform some program analysis. It features a core API to perform simple transformations as well as a tree API to perform more complex bytecode transformations (which requires more CPU processing and memory space).

iii) Translating the modified bytecode back to Dalvik bytecode..

Once the classes are analyzed and modified by the analysis toolkit, they are transformed back into Dalvik bytecode using `dx`[3] which generates the `classes.dex` file from Java class files. This step is illustrated in Fig. 1 as the edge c-d.

iv) Rebuilding application..

As presented in Fig. 1.(d-e), after the fourth step, a new Andoid application is built. The newly generated `classes.dex`, the data and the Android manifest from the original application are inserted in a new zip[4] file.

v) Signing the modified application..

Android requires applications to bear a cryptographic signature. Hence, all files of the generated zip file are signed using a newly created couple of public/private keys (not represented on the figure), and the new public key is added to the zip (not represented on the figure). For this, we used the `keytool` and `jarsigner` Java programs on the Android platform (Fig. 1.(e-f)).

2.4 Recapitulation

We have devised a bytecode manipulation process on Android using standard tools. The following presents the design and implementation of two concrete bytecode instrumentations and an evaluation of the time required to run the whole process on real applications.

---

[2]http://code.google.com/p/dex2jar/

[3]using `com.android.dx.command.Main.class` from the Android SDK

[4]using the `java.util.zip` library

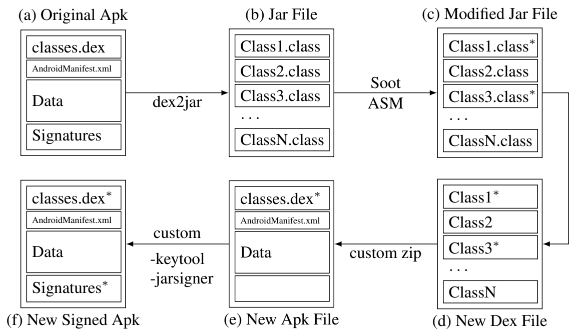

Figure 1: Process to Generate New Apk

3. USE CASES

There are different scenarios in which it would be beneficial to manipulate and analyse Android applications (more specifically the bytecode of the application) directly on smartphone devices. In this Section we present two scenarios we implemented: *AdRemover* and *FineGPolicy*.

3.1 Advertisement removal

Android enforces a per-application policy-based security model: Either all parts of an application benefit from a given permission, or none of its parts. Furthermore, Android applications are distributed as self-sufficient packages, bundling together both specifically developed code and the third-party libraries they may need, such as binary-only advertisement modules.

Put together, these two facts mean that when a user grants permissions to an application, she actually grants permissions to components potentialy written by different entities, with different goals and different security levels. If this user does not want, or is forbiden by her organization's policy to transmit her location to advertisement companies, she is not offered a way to express that, for example, a newspaper app is allowed to send her location back to the app publisher so that she is presented with local news, but this app is not allowed to send location data to a third-party. In such a case, the user faces a dilemma: She has to either reduce her security and/or privacy level expectation, or refrain from using an otherwise valuable application.

Several Ad libraries have been shown to use the dynamic code loading facilities provided by Android to download code from the Internet [12]. This raises potential security issues, since it may enable an attacker to inject malicious code that could leverage permissions granted to the host application.

Advertisement libraries also have a significant impact on mobile handsets stamina. Indeed, third-party advertisement modules can be held responsible for up to 65%-75% of energy spent in free applications [15].

Despite these issues, nearly half of the Android applications embeds third-party code to handle in-app advertisement [16]. A significant proportion of ad-supported apps include at least two advertising libraries [18].

3.1.1 *Overview*

Static analysis can be used to detect privacy and security issues of Android applications [12, 11]. Our tool chain allows these kind of static analysis to be performed, and provides a framework for implementing mitigation of the privacy and security threats detected by static analysis.

To demonstrate sub-application granularity possibilities offered by our program manipulation tool chain, we devised a method to prevent an application subset (*i.e.* a library or a set of libraries) from performing potentialy privacy-violating or security-breaching actions.

In order to avoid the need to reverse engineer every code module one might want to inhibit, we propose a generic, heuristics-based approach to hinder a library's operations. We then employ this approach to remove advertisement from Android applications.

3.1.2 *Implementation*

Our AdRemover tool currently focuses on two widely used Android advertisement modules. We collected the Java package names used by these libraries and we configured AdRemover to operate only on classes that are part of those packages.

Potentialy dangerous code often requires IO operations, Network data transfer for data leakage, GPS polling for location tracking or phone network for sending short messages.

Unlike regular local and repeatable computation, developers should expext IO operations to fail on a regular basis, depending on unpredictable context. For example, all kind of communication will be impossible if the device has no network coverage whatsoever.

Building on this observation, we make the assumption that potentialy dangerous code have been placed by developers inside a Try/Catch block to cater for exceptions raised by IO failures.

As a first step, our tool leverages this assumption and inhibits every Try/Catch[5] section of the targeted packages of the application. For every Try/Catch block it encounters, our tool obtains the type of the handled exception, creates such an exception object, and inserts an instruction that throws this exception at the very begining of the block.

To further hinder the third-party module's operations, our tool then renders inoperant a collection of utility methods. Our tool searches for every methods in targeted packages whose return type is either void or String (respectively), and replaces the body of the method by either a `return;` statement or a `return("");` statement (respectively).

We produced two implementations of AdRemover: One using Soot and one using ASM.

3.2 Fine-Grained Permission Policy

Android framework relies on a permission-based model and follows an *"all or nothing"* policy. At installation time, users must accept or reject permissions requested by application developers. An application is installed only if all the requested permissions are accepted. There is no way to accept certain permissions (such as accessing the geolocalization data) and refuse others (such as connecting to the Internet) not directly related to the main functionalities of the application. Users are doomed to completely trust the application developers. Enck et al. [9] have pointed out that an application with several sensitive permissions is a real security threat. For instance if an application requests the permission to send SMS and to read the contact list, the contact list could potentialy be sent to a remote phone by writing it in a text message and sending this message by SMS. Enforcing a finer-grained permission model is a efficient way to achieve a higher level of privacy for the user.

We propose to give users the ability to specify their own set of admissible permissions (according to their own usage) and that a system enforces this user-defined policy. Running such user-level

[5] Although the soot implementation of AdRemover performs manipulation in the Jimple language, we stick here to Java terminology for simplicity

security policy is possible on unmodified Android systems by manipulating the application bytecode at installation time.

3.2.1 Overview

For a user-centric policy to exist, we need to instrument the bytecode of every application one whishes to control. All API calls which require one or more permissions are redirected to a *policy service* which allows or not the call.

When the instrumented application runs, the user-defined policy is enforced by the policy service. Indeed, for every instrumented method, the running instrumented application calls the policy service and the policy is checked. If the policy allows the original API method call, the API call is performed. Otherwise, a fake implementation is executed and returns a fake default value.

3.2.2 Implementation

Our tool enforces a user-defined policy at the user level (also called application level). It allows users which can not modify the system policy to enforce their own policy for a set of applications.

Instrumenting the Application.

The first step is to instrument the bytecode of an application one whishes to control to limit its permissions. This is illustrated in Figure 2 where application X is represented as a graph of method calls starting from node s. All method calls which require one or more permissions [10, 1] are wrapped with code which:

1. asks the policy service if the application is authorized to call the method

2. according to the answer from the policy service either invokes the original method or the the fake method.

For instance, the `getLocation(p1)` method invocation of node *7* (which requires permission `GPS`) has been wrapped in the figure by a call to the *policy* service. If this call returs *null* the original `getLocation(p1)` is executed, otherwise a fake method is invoked.

There are N instrumentations where N is the number of API calls under consideration present in the application bytecode.

Defining the Policy.

The next step, as shown in Figure 3, is to define the policy regarding the instrumented applications. The user defines a set of allowed methods for each application. In Figure 3, only method `getLocation()` is allowed for application X'.

Note that this step could be performed first to instrument only method calls which are not authorized by the policy. However, instrumenting every API method calls which requires one or more permissions makes it possible to change the policy at runtime.

Policy Service.

Finally, when the instrumented application runs, the policy is enforced by the Policy service as shown in Figure 4. For every instrumented method (here the original/instrumented method is `get Location` and its associated permission `GPS`) the running application calls `policyCheck()` (step *A*) and the policy is checked by calling `policyHas()` (step *B*). Method `policyCheck()` returns `null` if the policy allows the original method, a stub service `reference` otherwise. If the original method is allowed in the policy, the original method is called (this is the case in Figure 4, since step *C* returns *null*). Otherwise, the stub method corresponding to the original method is executed: step *D* would return

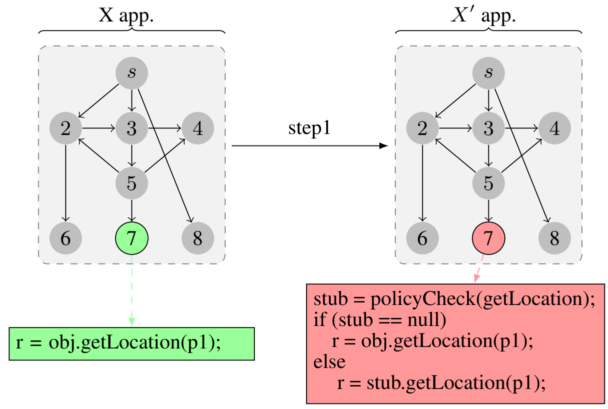

Figure 2: Step 1: Instrumenting Application Bytecode

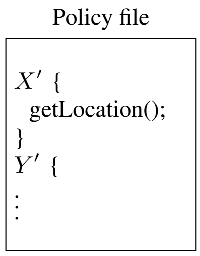

Figure 3: Step 2: Defining the Policy

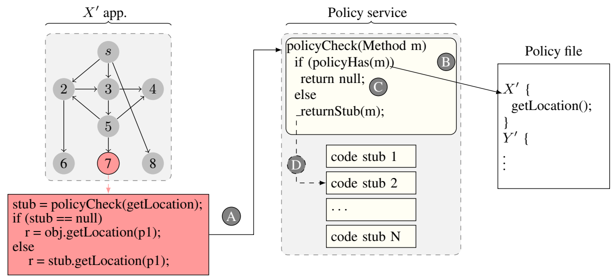

Figure 4: Step 3: Enforcing the Policy

a reference to the stub service 2 which implements the fake getLocation method. Here, `stub2` handling method `getLocation` is not executed .

3.3 Conclusion

AdRemover.

We implemented our advertisement removal tool with both soot and ASM. The whole application modification process takes less than 15 seconds on a recent PC laptop with the soot implementation and is even quicker with the ASM implementation.

We tested that our tool is functional by manually selecting one application from the dataset described in section 4.3. We made sure that the test application was using one of the two advertisement modules currently handled by AdRemover.

First we ran the unmodified test application on an Android devices, and confirmed that it was a working application, and that it displayed advertisements.

We then sent this application through our tool chain (with the soot implementation) running on a PC, and tested the modified application and confirmed it was still functional, and that no more advertisements were displayed. We monitored the network connection during the modified application test, and we found that it did not send any ad request anymore.

Finally, we processed the unmodified application again, this time running the bytecode manipulation directly on the smartphone. Running this modified application yielded the same results as with the application modified on a standard PC.

FineGPolicy.

We implemented the policy service as an Android service and the instrumentation code as a plugin for the static analysis tool Soot. We tested methods of two stub services we implementeted (ActivityManager, LocationManager) on one Android application on a desktop computer. The application is first instrumented and then repackaged into a new application which is then signed.

We ran the instrumented application on an Android device, and tested it with different policies. We were able to validate that the user-supplied policy was enforced as expected.

Those two implemented scenarios are further evaluated in Section 4.

4. EMPIRICAL RESULTS

In this section we present the results of applying the instrumentation process presented in section 2 and summarized in Fig. 1. The goal is to know: 1) whether it is possible to manipulate bytecode on smartphones given the restricted resources of the hardware. 2) whether it takes a reasonable amount of time.

4.1 Measures

We measured the execution time of the five steps of the instrumentation process on a set of 130 Android applications. This Android applications set is described in Section 4.3. We run the instrumentation process on three different Android smartphones whose configurations are presented in Section 4.2.

The feasibility of the whole process is measured by the time to pass every step of the toolchain (1: $dex2jar$, 2: $Soot/ASM$, 3: dx, 4: $customZip$, 5: $signature$). Moreover, the time to run each step is measured as well as the number of applications that successfully go through the step.

For the second step of the process (Step: Instrumenting the bytecode), we evaluate both ASM and $Soot$. For ASM, we measure the time required to instrument Java bytecode on the AdRemover case study. The AdRemover transformation leverages the ASM tree API and parses all classes to transform specific methods returning String or void as well as try/catch blocks. $Soot$ is evaluated by measuring the time it requires to generate Java classes on both AdRemover and FineGPolicy case studies.

4.2 Experimental Material

We conducted the experiment on three Android-based smartphone devices. Their configuration is detailed in Table 1. Indeed, it is a parameter that influences the results of bytecode manipulation tools which are memory demanding. The main differences are the processor clock speed (0.8, 1.2 and 1.4GHz), the total amount of main memory (512, 768 and 1024MiB), the Android version (2.2, 2.3.4 and 4.0.3) and the maximum heap size of the Dalvik virtual machine (24, 32 and 48). Since the heap size controls the maximum memory that can be allocated it also controls the maximum number of objects that can be allocated simultaneously.

Name	Processor	Memory	Android	Heap Size
smartphone1	ARM 800MHz, 1 core	512MiB	2.2	24MiB
smarthpone2	ARM 1.2GHz, 2 cores	768MiB	2.3.4	32MiB
tablet1	ARM 1.4GHz, 4 cores	1GiB	4.0.3	48MiB

Table 1: The Hardware used in our Experiment

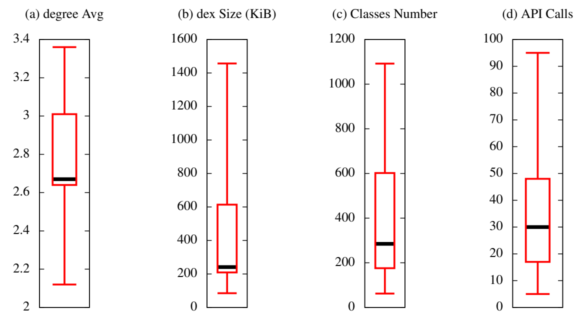

Figure 5: Descriptive Statistics of our Dataset (boxplots)

The number of cores also differs. However, we do not take advantage of multiple cores during the experiments. This hardware complies with requirement #3 mentioned in 2.2.

4.3 Dataset

The choice of a dataset is important because it reflects real world applications. Choosing a poor dataset may yield to better results (for instance if the code within applications is always small) but non useful results since they are non applicable on most real world applications. We apply the whole experimental protocol on a set of 130 Android applications randomly selected among the top 500 applications from the Android market[6]. They span various domains such as finance, games, communications, multimedia, system or news.

To give a better overview on these applications, Fig. 5 shows the key application metrics as boxplots. They indicate that most (75%) of Android applications have less than 614KiB of Dalvik bytecode, less than 602 classes, an average method degree smaller than 3. These applications also have less than 48 calls to a method of the Android API which require a permission. The most important results are the two former. Indeed, one of our hypothesis is that the processing time of a bytecode manipulation is highly dependent on the size of the bytecode. The other states that the time to create a new apk file from an original apk file also depends on the size of the original apk file. Hence, the lower the code and application size, the lower the processing time.

4.4 Dalvik to Java Bytecode Conversion

The conversion time from the Dalvik executable code to Java bytecode using *dex2jar* is shown in Fig. 6.

Observation 1 The `classes.dex` file size ranges from 7 to 4061 KiB. All dex files can be successfully converted to Java classes using a desktop computer. However, *smartphone1* is unable to process any dex file. When using *smartphone2* and *tablet1*, 26 and 11 dex files, respectively, caused the conversion Android applica-

[6] `http://play.google.com`

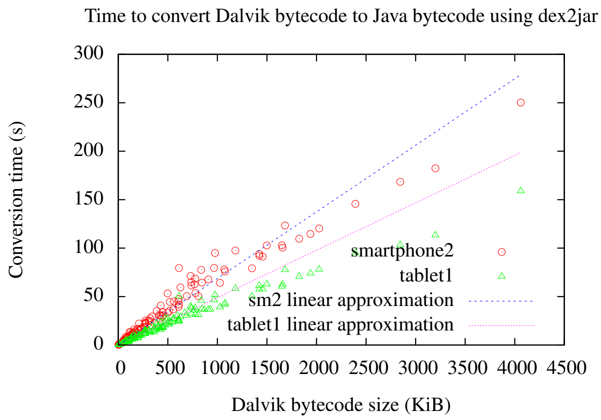

Figure 6: Dalvik to Java Bytecode Conversion Time.

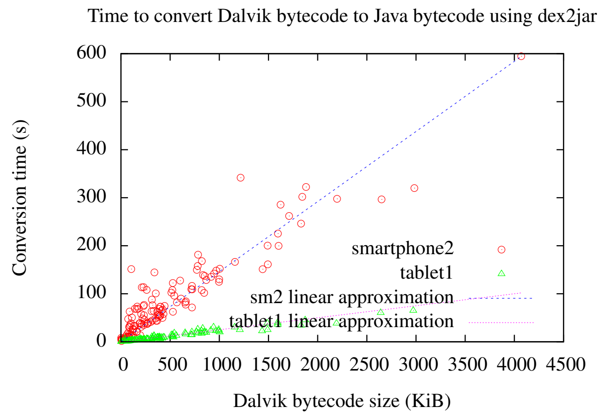

Figure 7: Transformation Time of Java Bytecode with ASM

tion dex2jar to crash. This crash is either an `OutOfMemory` or a `StackOverflow` exception.

smartphone1 result is explained by the hard-coded maximum heap size of Android (32MiB or 48MiB). For the two other devices, crashes are to be attributed to the default 8KiB stack size. In total, 104 (80%) Android applications were successfully converted to a jar file on *smartphone2* and 119 (91%) on *tablet1*.

Observation 2 We notice that the conversion time is linear with the size of the dex file (of the form $a \cdot X + b$) for a Dalvik bytecode size less than 4000KiB. Using linear regression, we find that for *smartphone2* a equals 0.069 and b equals 0.3. For *tablet1* we have, 0.049 and -0.4.

Observation 3 The time to convert dex files to jar does not exceed 60 seconds on *smartphone2* and *tablet1* for most of the applications (75%).

Observation 4 The application with the biggest dex file (4000KiB) was successfully converted both on *smartphone2* and on *tablet1*.

Conclusion 1 Converting Dalvik bytecode to Java bytecode is feasible and realistic since the conversion time does not exceed 250 seconds on our dataset of Android applications. Furthermore, there is a linear relation between the conversion time and the size of the Dalvik bytecode which enables us to theoretically predict the necessary amount of time to convert any size of Dalvik bytecode (if we extrapolate for size bigger than 4000KiB). For instance, the time to process the Android application with 10MiB of Dalvik bytecode would be 700 seconds for *smartphone2* and 500 seconds for *tablet1*.

As we can see by looking at the result from *smartphone1*, the heap size has a real impact on the result and dex2jar needs a minimum heap size ($24 <$ minSize $<= 32$) to process dex files. Since Android devices become more and more powerful the default heap size of the Android system grows. Indeed, in Android 2.2 the heap size is 24MiB, in Android 2.3.4 32MiB and in Android 3.0 48MiB. This continued growth allows to convert Android applications which have bigger Dalvik bytecode size.

Last but not least, the size of dex files is not the only factor which impacts the correct processing of dex2jar (the biggest application of our dataset was correctly processed), but its complexity also plays a role. The more complex the code is the more memory dex2jar requires.

4.5 Bytecode Manipulation and Use Cases Evaluation

4.5.1 Manipulation With ASM

Transformation time of Java bytecode using ASM is represented Fig. 7. In this experiment ASM does the AdRemover transformation described in 3.1. This transformation is based on the ASM tree API.

Observation 5 All 104 applications that were successfully transformed with dex2jar on *smartphone2* were successfully processed by ASM. It processes every jar (up to 4MiB in size) in less than 60 seconds.

Observation 6 We notice that the transformation time is linear with the size of the jar files (of the form $a \cdot X + b$) for a Dalvik bytecode size less than 4000KiB. Using linear regression, we find that for *smartphone2* a equals 0.146. For *tablet1* we have, 0.025.

Conclusion 2 Manipulating bytecode on smartphones using ASM is feasible. For all the bytecode analyzes and transformations that can be implemented with ASM, this is a promising result.

4.5.2 Manipulation With Soot

Recall that Soot was ported on the smartphone without any modification and that it was not designed to run on such platforms.

Observation 7 Out of the 130 Android applications 3 are correctly processed by Soot on *smartphone2* and 19 on *tablet1*.

Observation 8 It takes less than 30 seconds to convert any jar which size is less or equal to 20KiB on *smartphone2* whereas it takes less than 18 minutes to convert any jar which size is less or equal to 80.5KiB.

Conclusion 3 Using Soot directly on smartphones is feasible only for the smallest applications. However, when it successfully runs on a jar file it means that all classes of the jar files are loaded within Soot and that their bytecode can be manipulated. Soot successfully established the hierarchical relations between classes enabling ad-

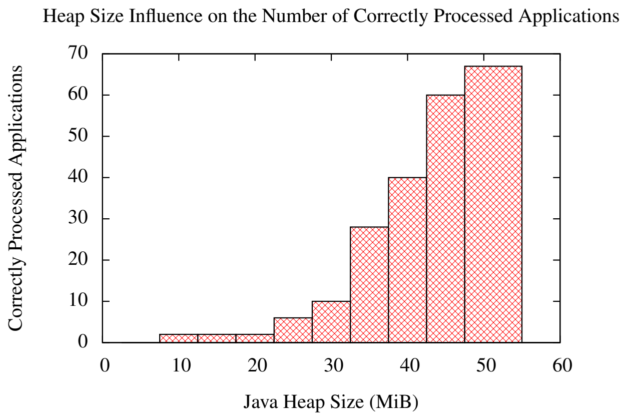

Figure 8: Influence of the Heap Size on Jimple Transformation

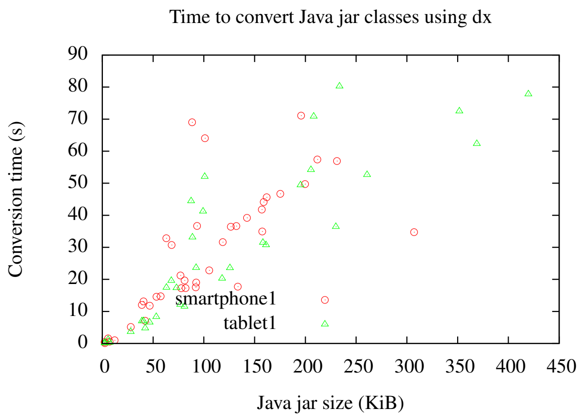

Figure 9: Conversion Time of Java Bytecode to Dalvik

vanced manipulations on bytecode and/or on abstractions of the bytecode.

We assume that the heap size is the main blocking factor of using Soot on smartphones directly. To check this assumption, we conducted an experiment on a desktop computer consisting of analyzing our dataset of Android applications with different maximal heap sizes (from 5Mib to 50Mib by steps of 5Mib). Results are presented Fig. 8. Soot was able to process 67 applications with a heap size of 50Mib. Those results clearly indicate that maximum half of the Android applications could be processed with a heap size of 50MiB. Under the assumption that the heap usage (hence the maximum required size) is similar on the Java and Dalvik virtual machines, it means that the memory is actually the main blocking factor of using Soot on Android.

We also notice that Soot takes much more time on *tablet1* which is running Android 4 than on *smartphone2* which is running Android 2. We hypothesise that the problem emerges because of different implementations and/or configurations of the heap and memory management of the Dalvik virtual machine between different Android versions.

4.6 Java Bytecode to Dalvik Conversion

Once an instrumented application has been produced at the Java

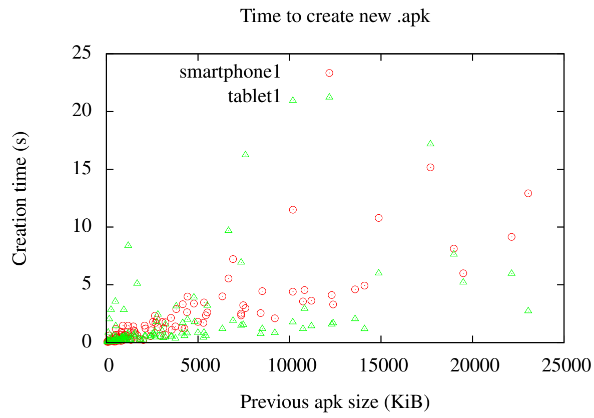

Figure 10: Creation Time of a New `apk` File

bytecode level, it has to be transformed back into Dalvik bytecode. Conversion time from Java classes to the dex file using the *dx* tool is shown in Fig. 9.

Observation 9 Java Bytecode of 33 and 39 applications on *smartphone2* and *tablet1*, respectively, have been successfully converted to Dalvik bytecode.

Observation 10 Conversion time for jar files ranging from 20 to 400KiB does not exceed 80 seconds.

Conclusion 4 The Dx tool is one bottleneck. It can only correctly process 25 to 30% of the applications. This tool is used off the shelf and can be optimized to run on devices where resources are limited. For instance, instead of puting every Java classes in memory it could process class after class to limit memory consumption.

4.7 Creating a New `apk` File

The time taken to create an `apk` file from the instrumented Dalvik bytecode is shown in Fig. 10. Note that for this step, the input set is not the output of the previous step. For applications where the Java bytecode can not be converted to Dalvik code, we take as input the original dex file of the application. In this way, the problems of the previous step do not interfere with the results of this fourth step.

Observation 11 Nearly all 130 inputs were successfully processed. There is no clear relation between the size of the previous `apk` file and the creation time of the new `apk`. Some applications generate an exception because their size is too big and can thus not be processed by the zip utility.

Observation 12 For 95% of the applications it takes less than five seconds regardless of the device and of the size of the original `apk` file. There is no significant difference in the computation time between applications as it is the case in Fig. 6 and 9.

Conclusion 5 It is feasible to create `apk` files on smartphones. The time to create a new `apk` file is negligible compared to the time to convert bytecode or to manipulate bytecode with Soot. The difference in the computation time only reflects the difference of CPU speed.

4.8 Signing the Generated `apk` File

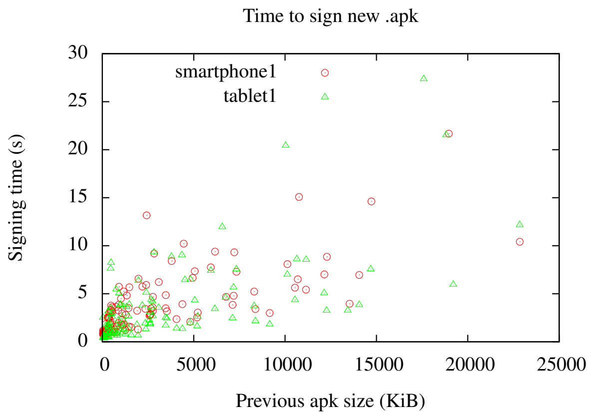

Figure 11: Signing the Generated Apk File

Signing time of applications is represented in Fig. 11.

Observation 13 Almost all 130 Android applications were successfully processed. There is no clear relation between the size of the `apk` file and the signature time of the `apk` file. Some applications generate an exception because their size is too big and can thus not be processed (14 on *smartphone2* 10 on *tablet1*).

Observation 14 For 95% of the applications a maximum of 12 seconds is required to sign the application regardless of the device and the size of the `apk` file.

Conclusion 6 It is feasible to sign `apk` files on smartphones. Similarly to the `apk` file creation step, the computation time is negligible. The difference observed between *smartphone1* and *smartphone2* reflects the difference in their CPU clock frequencies.

4.9 Conclusion

4.9.1 Feasibility

Those experiments show that it is feasible to manipulate bytecode directly on Android devices. The most expensive steps of the process are the conversion of Dalvik to Java bytecode and vice versa, and the Soot bytecode manipulation step. However, depending on the manipulation to be made on the bytecode, ASM could be sufficient. Table 3 is a summary of all the experiments for *smartphone2* and highlights the feasibility of the whole approach. Values for the sum of all steps for the Soot and ASM version are computed from the application that successfully passed through all steps. For an ASM-based instrumentation it takes a median time of 120 seconds to process an application. We think that users would agree with waiting 2 minutes before starting using an application, if they are provided more guarantees with this instrumentation process enabling e.g. malware detection. For the biggest application, it takes 952 seconds, i.e. less than 16 minutes which also seems reasonable to us, given that the vast majority of Android application are much shorter and therefore need less time to be processed.

4.9.2 Blocking Factors and Optimization Opportunities

According to our analysis, the main blocking factors which can be overcome are the following.

The maximum heap size required to analyze and transform applications is an issue for many transformation steps. We think that this issue will be easily solved by 1) the next generation of more powerful hardware and 2) the upcoming versions of the Android OS and virtual machines which will likely have a significantly higher maximum heap size (e.g. Android 4 heap size is set to 48MiB).

Actually, Dalvik to Java conversion and Java to Dalvik conversion are two very time expensive steps. They are required to use unmodified versions of Soot and ASM. However, those steps could be skipped if we had either 1) an ASM-like library for manipulating Dalvik bytecode or 2) a bi-directional transformation directly from Dalvik bytecode to Jimple bytecode which are both register based. We are now studying the required effort to design and implement such tools.

Last but not least, we have used tools in a completely unmodified version (in particular dex2jar, Soot). Most of these tools were not meant to run on platforms with limited resources and were never optimized as such. In other terms, we believe that there are probably many optimization opportunities in terms of CPU and memory consumption.

To sum up, our results show that we can reasonably imagine to manipulate the bytecode on 100% of our dataset applications within at most minutes.

4.9.3 Threats to Validity

Let us now discuss the threats to validity of our experimental results.

Implementation Bug: Our results hold as far as there is no serious bug in the implementation of any of the five programs involved in the five steps, as well as in the glue and measurement code we wrote.

Dataset Generalizability: Our dataset may not be representative of the Android applications used in the real-world.

Linear Extrapolation: The linear relations we establish for the Dalvik to Java and the Java to Dalvik conversions holds for bytecode size less or equal to 300KiB. It may not hold for bytecode whose size is bigger. In the presence of non-linear singularities, it may not be possible to analyze large applications.

Bytecode Manipulation Time: Our results on the bytecode manipulation time were obtained with relatively simple transformations. It may be the case that complex transformations are not of the same order of magnitude. However, for the use cases presented in Section 3, the instrumentation that is needed is mostly monitoring and proxying. We see no reason why this may take a much longer time.

5. RELATED WORK

Monitoring applications.

The idea of monitoring smartphone applications at runtime recently emerged, due to the explosion of "mobile" malware and the increasing sophistication of mobile OS.

Bose et al. [2] aimed at detecting malware based on their behavior at runtime. For this, they added hooks in the Symbian OS emulator to track OS and API calls. In other words, malware detection is only achieved in the emulator, *in vitro*. On the contrary, we aim malware detection in live user environments, *in vivo* and showed in this paper that is it is feasible in the mid-term.

Step Name	Min. Time (s)	Avg. Time (s)	Median Time (s)	Max. Time (s)	App.	Feasibility
Conversion .dex to .jar (a-b)	0.22	43.76	28.9	250.2	104/130 (80%)	⋆⋆⋆
Analyzing .jar with Soot(b-c)	25.8	76	26	187.7	3/130 (2.3%)	
Analyzing .jar with ASM(b-c)	1.55	90.45	65.1	594.67	129/130 (99.2%)	⋆⋆⋆⋆
Conversion class to dex (c-d)	0.09	28.07	22.8	71	39/130 (30%)	⋆⋆
Creating new .apk (d-e)	0.06	1.89	0.87	15.1	119/130 (91.5%)	⋆⋆⋆⋆
Signing new .apk (e-f)	0.71	3.85	3.0	21.67	116/130 (89.2%)	⋆⋆⋆⋆
All Steps with Soot (a-b-c-d-e-f)	26.88	153.57	81.57	545.67	3/130 (2.3%)	⋆
All Steps with ASM (a-b-c-d-e-f)	2.63	168.02	120.67	952.64	39/130 (30%)	⋆⋆⋆

Table 2: Summary (for Smartphone2)

Step Name	Min. Time (s)	Avg. Time (s)	Median Time (s)	Max. Time (s)	App.	Feasibility
Conversion .dex to .jar (a-b)	0.19	25.6	17.85	158.9	119/130 (91.5%)	⋆⋆⋆
Analyzing .jar with Soot(b-c)	24.2	76	352	1054	19/130 (14.6%)	⋆
Analyzing .jar with ASM(b-c)	1.55	11.3	7.06	65.5	119/130 (91.5%)	⋆⋆⋆⋆
Conversion class to dex (c-d)	0.09	29.5	20.2	80.2	33/130 (25.3%)	⋆⋆
Creating new .apk (d-e)	0.03	1.6	0.5	20.9	121/130 (93.1%)	⋆⋆⋆⋆
Signing new .apk (e-f)	0.4	3.4	1.91	27.3	120/130 (92.3%)	⋆⋆⋆⋆
All Steps with Soot (a-b-c-d-e-f)	24.91	136.1	392.46	1341.3	19/130 (14.6%)	⋆
All Steps with ASM (a-b-c-d-e-f)	2.26	71.4	47.52	352.8	33/130 (25.3%)	⋆⋆⋆

Table 3: Summary (for Tablet1)

Enck et al. [8] presented a runtime monitoring framework called TaintDroid, which allows taint tracking and analysis to track privacy leaks in Android. Their prototype is based on a modified version of the Dalvik virtual machine which runs Android applications. Similarly, Costa et al. [6] extends the Java virtual machine for mobile devices (Java ME) for adding runtime monitoring capabilities. On the contrary, our feasibility study indicates that it is possible to achieve runtime monitoring in an unmodified Android system.

Recently, Burgera et al. [4] presented an approach to detect malware based on collected operating system calls. Runtime monitoring can be done at different granularity levels. While the approach described by Burgera et al. is at the OS call grain, we aim at providing runtime monitoring at the API call level, i.e. much more fine-grained and closer to the application domain of mobile applications.

Davis et al. [7] presented an Android Application rewriting framework prototype, and discussed its use for monitoring an application, and for implementing fine-grained Access Control.

Finally, Shabtai et al. detects malware based on the collection and analysis of various system metrics, such as CPU usage, number of packets sent through the Wi-Fi, etc. This is an indirect way of detecting malware behavior. Again, by monitoring API calls, we observe the application behavior directly. The empirical results presented in this paper shows that this is actually possible.

Advertisement Permissions separation.

Shekhar et al. [18] proposed a new Android advertisement system that would allow to have an application and its advertisement module to run in different processes, and hence have a different permission set. This new system has to be manually inserted into the application during the development phase, since no automated application modification is provided.

Pearce et al. [16] made the case for a Advertisement framework that would be integrated inside the Android platform. Every developer would be able to use the custom-built API that would be available on Android devices. This approach requires a modification of the Android framework, and that a given user has a device with a Android version embeding this advertisement system.

Permission Policy.

Reddy et al. [17] claim that security of the Android platform would be improved by creating "application-centric permissions" i.e. permissions expressing what an application can do rather than current Android permissions that express what ressource an application can use. They wrote a library that allows the 'application-centric permissions" to be managed. In addition, they started developing a tool called "redexer" whose aim is to automatically rewrite existing applications in order for them to use these new permissions.

Nauman et al. [14] extended the Android policy-based security model so that it can enforce constraints at runtime. The tool they created, called Apex, allows a user to express limits imposed to an application's use of any permission: For example, it becomes possible with Apex to grant the `SEND_SMS` permission to any given application while ensuring that this application will not be able to send more than a user-defined amount a text message each day. The user also has the possibility to change her mind, and to totally prevent the application from sending short messages; This is an important improvement over the stock Android OS because it allows users to specify a much finer-grained policy, instead of having to choose between either granting an application every permission it may request at installation time or not installing this application. However, this approach requires modifications deep inside the Android framework, and hence would need to be backed by Google and integrated into future versions of Android if it was to be widely used.

6. CONCLUSION

The toolchain we propose and evaluate in this paper is a milestone that respond to the recent claim of Stravou et al. [19] about the urgent need for bytecode analysis to perform in-vivo security checks on mobile phones. We have 1) proposed a tool chain allowing the manipulation, instrumentation and analysis of Android bytecode and 2) shown that it is possible to run the tool chain in a reasonable amount of time directly on unmodified smartphones with unmodified Android software stack. Concretely, our experiment shows that with ASM, 39 (30%) applications of our dataset can be instrumented in less than 952 seconds (with a median time of 120s). Moreover, we discuss specific limitations that we observed, such as the hardcoded heap size of Android systems.

We believe that those various limitations could be quickly overcome, at least for two main reasons. First, we used off-the-shelf Java tools that are not optimized to run on environments where resources (memory/CPU) are limited, and there may be possibilities of significant optimization. Second, the hardware and OS evolution of smartphones will make it possible to process ever bigger Android applications (for instance, on Android 4, the default size of the heap is twice as large as in the previous version).

We are currently working on other use cases. In particular, we are implementing a behavioral malware detection approach that is set up and run on the smartphone. This approach involves instrumenting the bytecode to redirect API method calls to stubs responsible for detecting malicious behavior.

7. REFERENCES

[1] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Automatically securing permission-based software by reducing the attack surface: An application to android. Technical Report 978-2-87971-107-2, University of Luxembourg, 2011.

[2] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *MobiSys*, pages 225–238, 2008.

[3] E. Bruneton. ASM 3.0, a Java bytecode engineering library. `http://download.forge.objectweb.org/asm/asm-guide.pdf`, 2007.

[4] 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*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[5] Canalys. Android takes almost 50% share of worldwide smart phone market, 2011. `http://www.canalys.com/static/press_release/2011/canalys-press-release-010811-android-takes-almost-50-share-worldwide-smart-phone-market_0.pdf`.

[6] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter, R. Runtime monitoring for next generation java me platform. *Computers & Security / Computers and Security*, 29(1):74–87, 2010.

[7] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In *IEEE Mobile Security Technologies (MoST)*, San Francisco, CA, 2012.

[8] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[9] 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*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[10] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. Technical Report UCB/EECS-2011-48, University of California, Berkeley, 2011.

[11] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.

[12] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM.

[13] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6:151–180, August 1998.

[14] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[15] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[16] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of AsiaCCS*, May 2012.

[17] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster. Application-centric security policies on unmodified android. Technical report, Technical Report 110017, University of California, Los Angeles, 2011.

[18] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. *CoRR*, abs/1202.4030, 2012.

[19] A. Stavrou, J. Voas, T. Karygiannis, and S. Quirolgico. Building security into off-the-shelf smartphones. *Computer*, 45-2, 2012.

[20] R. Vallée-Rai, L. Hendren, V. Sundaresan, E. G. Patrick Lam, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.