In-Vivo Bytecode Instrumentation for Improving Privacy on Android Smartphones in Uncertain Environments

Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix, Yves Le Traon

HAL Id: hal-00700319

https://hal.archives-ouvertes.fr/hal-00700319v2

Submitted on 7 Oct 2013

In-Vivo Bytecode Instrumentation for Improving Privacy on Android Smartphones in Uncertain Environments

Alexandre Bartel · Kevin Allix · Martin Monperrus · Jacques Klein · Yves Le Traon

Abstract In this paper we claim that an efficient and readily applicable means to improve privacy of Android applications is: 1) to perform runtime monitoring by instrumenting the application bytecode and 2) 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: BetterPermissions, 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.

Keywords: Android Security, Bytecode Manipulation, Privacy, Fine-grained permission policy

1 Introduction

Android is one of the most widespread mobile operating system in the world accounting for more than 72% of the market share [1]. More than 500 000 Android applications available on dozens of application markets can be installed

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

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

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 her smartphone is similar to choosing an apple on an apple tree: she only sees the surface and has no evidence that there is no worm in it. Unfortunately there are many worms of different kinds waiting to infect smartphones such as malware leaking private data and adware calling premium-rate numbers.

In this paper we claim that an efficient and readily applicable means to improve privacy of Android applications is to perform runtime monitoring and interception of the application interactions with the Android stack by instrumenting the application bytecode directly on the smartphone (in-vivo). Before further introducing our contribution let us defend our key claim.

Why performing runtime monitoring and interception? We want to allow or disallow behaviors of an application at runtime. We use runtime monitoring as it consists of observing the behavior of an application during execution. It collects certain metrics or intercepts all exchanges at the interface between the application and the rest of the system. 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 [2].

Why performing bytecode instrumentation? There are at least two ways to perform runtime monitoring and interception: modification of the Android software stack or bytecode instrumentation. Modification of the software execution stack consists in 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. Also, this solution would require users to change their firmware which is a non-trivial task, further complicated by the so called *fragmentation problem* of the Android system as there is not a single Android system but many different Android systems each customized to run on a specific device (tablet, smartphone, . . .). If the operating system is modified, one would need to create a custom instrumented version for every possible Android version which is not easily doable in practice. Bytecode instrumentation however, 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-grained policy enforcement for improving privacy, we are able – thanks to bytecode instrumentation – to enforce a fine-grained permission model of already deployed applications on Android smart-

---

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

phones without any modification of the Android software stack.

Why performing in-vivo instrumentation directly on smartphones? Bytecode instrumentation could be done outside the device for instance using a remote service on the Internet. However, many countries forbid distributing binaries to third-party services (e.g. France). Also, terms of service of several markets (e.g. Google Play for Android) do not allow this. Instrumenting applications directly on the device keeps the application within the device.

To sum up, we believe that the most efficient and practical way for ensuring security and privacy on mobile devices is to instrument the application bytecode directly on the smartphone (in-vivo), the instrumentation being tailored for a given security or privacy concern. Our main contributions are that:

- We have built a toolchain to automatically repackage Android applications directly on an Android device;
- We have built a toolchain to automatically analyze Android applications directly on an Android device;
- The toolchain has been tested by implementing two prototypes which increase the end-user privacy. One removes advertisement and the other gives the user total control over the applications' runtime permissions.
- The feasibility of such a tool chain has been evaluated. Limitations and challenges have been pinpointed.

To the best of our knowledge, we were the first[2] to present a tool chain to automatically transform Android applications directly on a device.

The paper is organized as follows: Section 2 provides the reader with two scenarios motivating the need of bytecode instrumentation of Android applications. Section 3 describes a tool chain for instrumenting Android applications directly on Android devices (smartphones, tablets, ...). Section 4 presents the design and implementation of valuable bytecode instrumentations for the security and privacy of smartphones. Section 5 demonstrates the feasibility of running the whole tool chain in a reasonable amount of time. Section 6 discusses the related work and Section 7 concludes the paper.

2 Use Cases of In-Vivo Instrumentation

There are different scenarios in which it would be beneficial to manipulate and analyze Android applications' bytecode directly on smartphone devices (in vivo). In this Section we present two valuable use cases: *AdRemover* and *BetterPermissions*.

Both of them improve the privacy for the user. *AdRemover* hinders advertisement libraries to work and thus, at the same

time, prevents them from sending private information related to localization (GPS coordinates,...) or of the device itself such as the IMEI (International Mobile Equipment Identity). *BetterPermissions* gives users the power to enable or disable applications' permissions. In an extreme case where the user would like no application to have access to her contact list, she would remove the contact permission from all applications on the phone. The result is a better privacy for the user.

2.1 Advertisement Removal

Nearly half of the Android applications embeds third-party code to handle in-app advertisement [4]. A significant proportion of ad-supported apps include at least two advertising libraries [5].

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.

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. It means that when a user grants permissions to an application, she actually grants permissions to components potentially written by different entities, including the ad libraries.

For example, a newspaper app may be allowed to send its location back to the app publisher so that she is presented with local news. However, from a privacy perspective the embedded advertisement library should not be allowed to send the location data to the ad companies. Currently, the user faces a dilemma: she either has to reduce her privacy level expectation, or refrain from using an otherwise valuable application.

A workaround of this limitation of the platform is to disable the use of the ad library in-vivo.

This may have positive side-effects, since advertisement libraries also have a significant impact on the battery usage. According to a recent study [6], third-party advertisement modules can be held responsible for up to 65%-75% of energy spent in free applications .

2.2 Fine-Grained Permission Policy

The Android framework relies on a permission-based model and follows an *"all or nothing"* policy. At installation time, users must either accept or reject all permissions requested by the application. An application is installed only if all the requested permissions are accepted. There is no way to accept only some permissions (such as accessing the localization data) and not others (such as connecting to the Internet).

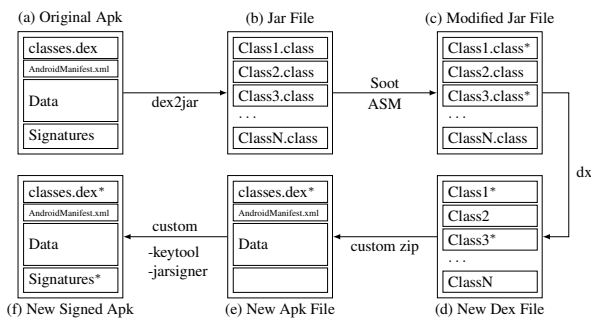[2] we published a technical report in May 2012 [3]

Fig. 1 Our Process to Instrument Android Applications

Users are doomed to completely trust the application developers who write the list of permission. Enck et al. [7] 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 a permission to read the contact list, the contact list could potentially be sent to a remote phone by sending it through SMS.

A fine-grained permission model consists in giving users the ability to specify their own set of permissions to applications, according to their own usage. All sets of permissions for all applications on the device constitutes the security policy. The underlying permission-based system would then enforce this user-defined policy.

Running such user-level security policy is impossible on a unmodified Android platform with unmodified application code. However, as we show later, it is indeed possible by manipulating the application bytecode at installation time, in-vivo.

3 Toolchain for In-vivo Bytecode Instrumentation

This section presents our proposal for performing bytecode instrumentation of Android applications in vivo, i.e. directly on smartphones.

3.1 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 `Android Manifest.xml` (application metadata), data (e.g. images), and the public key needed to check the provided signatures of all files.

Applications are made of four different components (software abstractions): Activity for the user interfaces, Service for background operations, Content Provider to save data

and Broadcast Receiver to receive messages from the system. The bytecode of an Android application interacts with the Android system through the Android framework (also referred to as the Android API). Some methods of the API are protected by permissions because they access system resources such as GPS coordinates or the contact list. Only an application with the right permission(s) can access a protected resource.

3.2 Requirements

Instrumenting and repackaging a fully-runnable Android application 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 applications to be signed.

Our toolchain has the following requirements:

1. The Android OS must be unmodified (for the sake of a broad applicability as presented in Section 1);
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 Section 1);
3. The hardware that is used to instrument bytecode must be representative of common smartphones on the market.

3.3 Toolchain

The bytecode instrumentation process features the following steps: 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.

Those three steps can be broken down into five elementary steps, as shown in 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 which can be analyzed with standard unmodified Java bytecode analysis toolkits. For this step, we use the tool `dex2jar`[3].

---

[3] available at `http://code.google.com/p/dex2jar/`

ii) Instrumenting the Bytecode In this step, we experiment with two different tools which manipulate bytecode. Recall that bytecode manipulation is the step from (b) to (c) at illustrated in Figure 1. Using different tools gives us the opportunity to measure the difference in terms of execution time and memory consumption between them and decide which one is more appropriate to manipulate bytecode in a memory constrained system.

ii.a) Soot. Classes are transformed to Jimple with the Soot analysis toolkit. Soot [8] is an open-source analysis toolkit for Java programs. It operates either on Java source code or bytecode. It allows developers to analyze and transform programs. For instance, an intra-procedural flow analysis could determine if a variable can be *null* at some point in the code. Soot can also perform different call-graph 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 on the Android system by converting its Java bytecode to Dalvik and creating a wrapper Android application. To our knowledge there is no previous work which represent Android bytecode as an abstraction on which on could perform static analysis directly on the smartphone.

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 [9] 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`[4] 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 all inserted in a new zip[5] file.

---

[4] using `com.android.dx.command.Main` from the Android SDK
[5] using the `java.util.zip` library

v) Signing the Modified Application Android requires applications to be cryptographically signed. Hence, all files of the generated zip file are signed using a newly created couple of public/private keys (not represented on the figure), The new public key is added to the zip (not represented on the figure). We used the `keytool` and `jarsigner` Java programs to sign applications (Fig. 1.(e-f)).

Signing applications with new keys may cause compatibility problems between applications. For instance two or more applications signed with the same key can share the same process. In order for this feature to continue working a one-to-one mapping between old keys and new ones needs to be maintained in order to sign two transformed applications (originally signed with the same keys) with the same new generated keys. Maintaining this mapping and handling such compatibility between applications is out of scope of this paper.

We have devised a bytecode manipulation process on Android using standard tools. The following presents the design and implementation of two concrete bytecode instrumentation prototypes.

4 Use-case Design and Implementation

Any use-case leveraging the toolchain presented in Section 3 analyzes or modifies the bytecode of an application. Analyzing or modifying the bytecode is represented by step (b-c) in Figure 1. We now present how we have implemented and evaluated the two use-cases of Section 2. Thy both modify the bytecode of applications. AdRemover modifies the bytecode to remove advertisement. BetterPermissions modifies the bytecode to enable a fine-grained permission policy system for the user.

4.1 Implementation of AdRemover

We focus on two widely used Android advertisement modules: AdMob and AdSense. Advertisement is not part of the Android system but is present in the application's bytecode. Thus, applications do not share ad library code. However, they each have a copy of the library code. Disabling advertisement requires to instrument every application containing an ad library.

Advertisement requires I/O operations for fetching the ad data. An Android application developer using an ad library do not want her app to crash because of the ad library. This is the reason why developers of ad libraries take special care of exceptions when designing the ad library. They expect I/O operations to fail on a regular basis, depending on unpredictable contexts. For example, an exception can be thrown if the device has no network coverage anymore.

Building on this observation, we make the assumption that I/O code has been placed by ad developers inside a Try/-Catch block to recover for exceptions raised by I/O failures. Our tool leverages this assumption and inhibits every Try/-Catch section of the ad packages of the application. For every Try/Catch block it encounters, our tool extracts the type of the handled I/O exception, creates such an exception object, and inserts an instruction that throws this exception at the very beginning of the try block.

For this, 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. We wrote two implementations of AdRemover: One using Soot and one using ASM.

4.2 BetterPermissions: A Fine-grained Permission Policy Management

In this context a fine-grained policy is a file in which the user specifies which permissions are granted to applications. In the real world users are only familiar with permissions and applications, so it makes perfect sense to limit policies at the level of applications and not a lower level (such as Android component or Java methods). However, for explanatory purposes the policies in this Section contain a mapping between Java methods and permissions.

For a user-centric policy to exist, we need to instrument the bytecode of every application one wishes to control. Recall from Section 3.1 that Android applications communicate to the Android system through the Android API. The instrumentation detects all API calls protected by one or more permissions and redirected every of those calls to a *policy service*. The policy service is a Android service component part of independent Android application. Base on the user defined policy it authorizes or not the application to call the protected method.

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.

Our prototype tool enforces a user-defined policy at the user level (also called application level). It allows users who previously could not modify the system policy to enforce their own policy for a set of applications. Modifying code to insert security check is known as Inline Reference Monitoring (IRM) and has been first introduced by Erlingsson et al. and Evans et al. [10, 11, 12].

Instrumenting the Application To control or limit an application's permission it's bytecode has to be instrumented.

This is illustrated in Figure 2 where application $NewsReader$ is represented as a graph of method calls starting from node s. All method calls that require one or more permissions [13, 14] are wrapped with code which in order:

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 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 the policy approves this call, the original getLocation(p1) is executed, otherwise a fake method is invoked, returning a fake default value.

In total, 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 permissions for each application. Behind the scene, the policy generates a list of all Java methods which require the enabled permissions. Those methods are set as authorized. In Figure 3, only method getLocation() is allowed for application Instrumented NewsReader.

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 getLocation and its associated permission GPS) the running application calls policy Accepts() (step *A*) and the policy is checked by calling policyHas() (step *B*). Method policyAccepts() returns true if the policy allows the original method or false if it does not. 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 *true*). Otherwise, the stub method corresponding to the original method is executed. Here, the stub handling method getLocation is not executed. We implemented the policy service as an Android service and the instrumentation code as a plugin for the static analysis tool Soot.

4.3 Evaluation

We now check whether our use-case implementations work. For both of them, we run the instrumentation against a real-
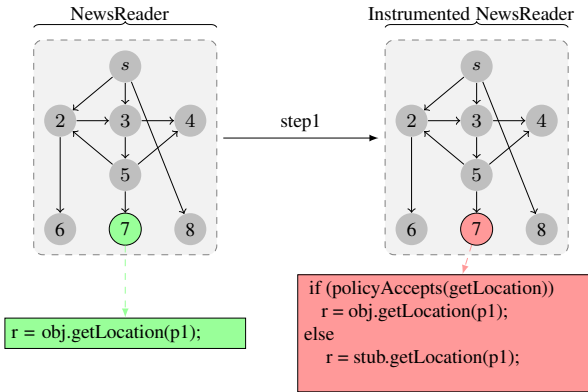
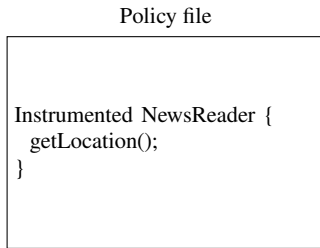**Fig. 2** Step 1: Wrapping and Redirection of Android API Calls For Fine-Grained Permission Management

Fig. 3 Step 2: The Policy File Defines that InstrumentedNewsReader is Allowed to Use API Method getLocation
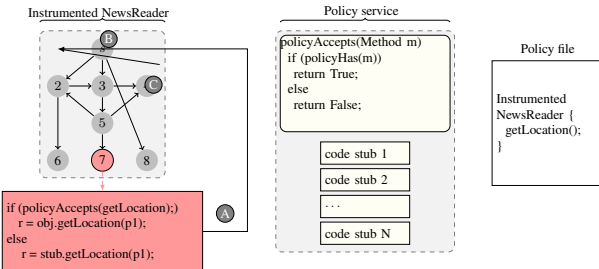

Fig. 4 Step 3: the Policy Monitor Enforces the Fine-Grained Permissions by Returning Default Values for Unauthorized API Calls

world application and runs the resulting modified application.

AdRemover We test that our tool is functional by selecting a random application on the Android Market. We make sure that the test application uses one of the two advertisement modules currently handled by AdRemover.

First we run the unmodified test application on an Android devices, and make sure that it is a working application, and that it actually displays advertisements.

We then send this application to our toolchain (with the Soot implementation) running on a PC. The modified application is still functional, and no more advertisements are displayed. We monitor the network connection during the

test and found out that it the application does not send any ad request anymore.

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

BetterPermissions For evaluating the fine-grain policy, we select another random application and instrument it to wrap every permission sensitive API call related to the GPS. The application is instrumented and then repackaged into a new signed application. We run the instrumented application on an Android device, and test it with different policies. The user-defined policy is enforced as expected.

To sum up, the two bytecode transformations result in applications that correctly runs. Those first results are important as the two use cases illustrate what can be achieved using the bytecode instrumentation toolchain. What also matters for us is to know whether the toolchain under consideration can be run in vivo on a large dataset of Android applications given the memory and CPU limitations of current smartphones. The next section answers to this questions by measuring execution time and memory consumption of in vivo instrumentation.

5 Performance of In-Vivo Instrumentation

In this section we present the results of applying the instrumentation process presented in Section 3 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.

5.1 Measures

We measure the execution time of the five steps of the instrumentation process on a set of 130 Android applications. This set is described in Section 5.3. We run the instrumentation process on three different Android smartphones whose configurations are presented in Section 5.2.

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: $signature$). The time to run each step and the number of applications that successfully go through each step are measured as well.

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

Name	Processor	Memory	Android	Heap Size
smartphone1	ARM 800MHz, 1 core	512MiB	2.2	24MiB
smartphone2	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

on the AdRemover case study. The AdRemover transformation leverages the ASM tree API to perform the try-/catch block manipulation described in 4.1. *Soot* is evaluated by measuring the time required to generate Java classes for both AdRemover and BetterPermissions case studies (AdRemover is implemented with ASM and Soot).

5.2 Experimental Material

We conduct the experiment on three Android-based smartphone devices. Their configuration is detailed in Table 1. The main differences are the processor clock speed (0.8, 1.2 and 1.4 GHz), the total amount of main memory (512, 768 and 1024 MiB), 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 by a single process it also controls the maximum number of objects that can be allocated simultaneously.

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 3.2.

5.3 Dataset

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. This dataset is not artificial as it only consists of real world applications.

To give a better overview on these applications, Figure 5 shows the key application metrics as boxplots. They indicate that most (75%) of Android applications have less than 614 KiB of Dalvik bytecode, less than 602 classes, an average method degree smaller than 3. Haft of the applications have more than 30 calls to a method of the Android API which require a permission.

5.4 Dalvik to Java Bytecode Conversion

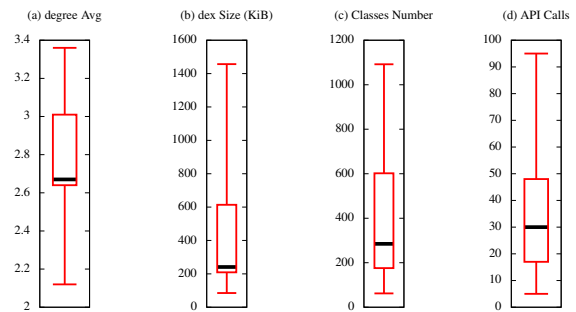The conversion time from the Dalvik executable code to Java bytecode using *dex2jar* is shown in Fig. 6.

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

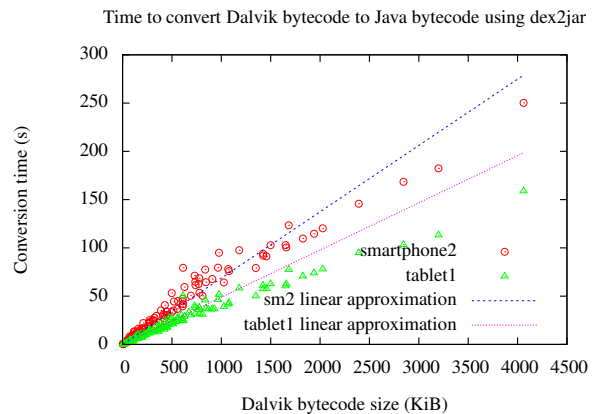**Fig. 5** Descriptive Statistics of the 130 Applications of our Dataset

Fig. 6 Performance of Dalvik to In-Vivo Java Bytecode Conversion.

Observation 1 The time to convert dex files to jar does not exceed 60 seconds on *smartphone2* and *tablet1* for 75% of the applications. The conversion time does not exceed 250 seconds on our dataset of Android applications.

Observation 2 The application with the biggest Dalvik bytecode file (4000 KiB) is successfully converted both on *smartphone2* and on *tablet1*.

Observation 3 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 4000 KiB. 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. The linear relation between the conversion time and the size of the Dalvik bytecode enables us to theoretically predict the necessary amount of time to convert any size of Dalvik bytecode (if we extrapolate for size bigger than 4000 KiB). For instance, the time to process the Android application with 10 MiB of Dalvik bytecode would be 700 seconds for *smartphone2* and 500 seconds for *tablet1*.

Conclusion 1 Converting Dalvik bytecode to Java bytecode in-vivo is feasible within minutes.
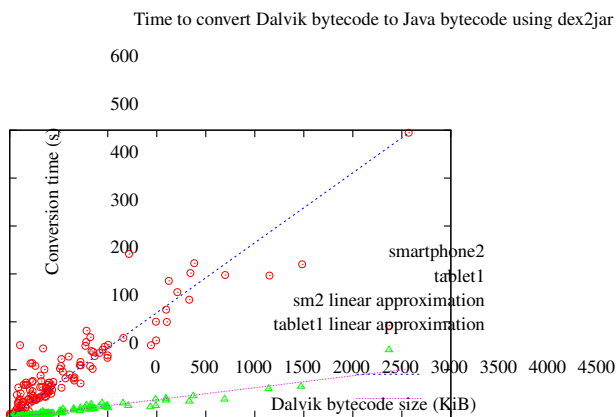
Fig. 7 Transformation Time of In-Vivo Java Bytecode Manipulation with ASM

Limitations: smartphone1 is unable to process any dex file. Also, when using *smartphone2* and *tablet1*, 26 and 11 dex files, respectively, cause the conversion Android application dex2jar to crash. This crash is either an `OutOfMemory` or a `StackOverflow` exception.

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

However, 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 24 MiB, in Android 2.3.4 32 MiB and in Android 3.0 48 MiB. This continued growth would allow our tool chain to convert Android applications which have bigger Dalvik bytecode size.

Also, some applications may be obfuscated to prevent Dex2jar to convert Dalvik bytecode to Java classes. We did not encounter any obfuscation during the experiment. Our toolchain relies on independent components. Thus, if Dex2jar cannot handle some obfuscation techniques it could easily be replaced by an equivalent component which handles them.

5.5 Performance of Bytecode Manipulation

This section presents our performance measures of in-vivo bytecode manipulation using two different instrumentation libraries: ASM and Soot.

5.5.1 Manipulation With ASM

Transformation time of Java bytecode using ASM is represented in Figure 7. In this experiment the AdRemover transformation described in 2.1 is implemented using ASM.

Observation 4 All 104 applications successfully transformed with dex2jar on *smartphone2* are successfully processed by ASM in-vivo. It processes every jar (up to 4MiB in size) in less than 600 seconds.

Observation 5 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 4000 KiB. 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. Given our transformation and our dataset, ASM does not have specific memory or CPU requirements that are incompatible with smartphone resources.

5.5.2 Manipulation With Soot

We now consider the Soot implementation of the AdRemover transformation. Out of the 130 Android applications, only 3/130 are correctly processed on *smartphone2* and 19/130 are correctly processed on *tablet1*.

Observation 6 Only the smallest applications (in terms of Dalvik bytecode) can be converted. For instance, it takes less than 30 seconds to convert any jar which size is less or equal to 20 KiB on *smartphone2*. However, larger, yet small applications (in the 25% quartile), take up to 18 minutes for being instrumented with Soot.

Conclusion 3 Using Soot in-vivo is feasible only for the smallest applications. We assume that the heap size is the main blocking factor of using Soot in-vivo. 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 5 Mib to 50 Mib by steps of 5 Mib). Results are presented Fig. 8. Soot was able to process 67 applications with a heap size of 50 Mib. Those results clearly indicate that maximum half of the Android applications could be processed with a heap size of 50 MiB. 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.

5.6 Java Bytecode to Dalvik Conversion

Once an application has been instrumented at the Java 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.
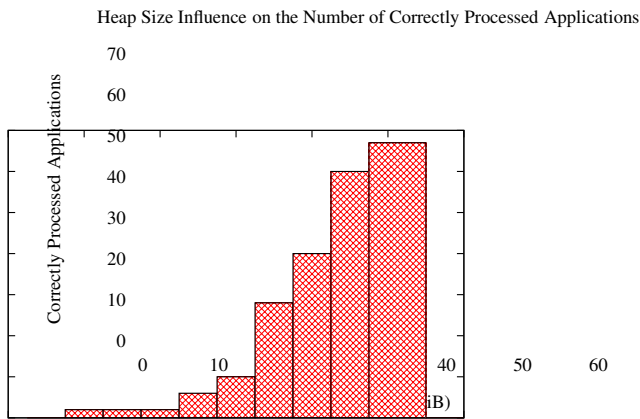
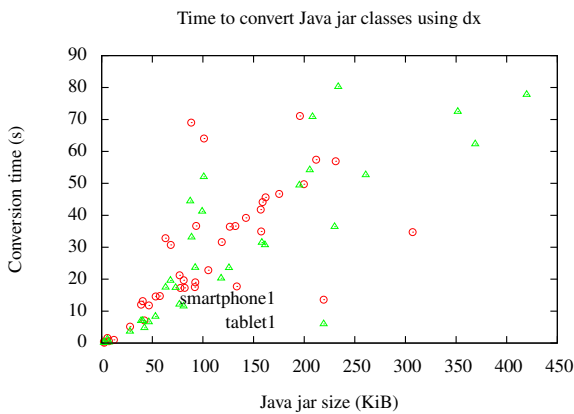Fig. 8 Influence of the Heap Size on Jimple Transformation

Fig. 9 Conversion Time of In-Vivo Java Bytecode to Dalvik Translation.

Observation 7 Java bytecode of 33/130 on *smartphone2* and and 39/130 applications on *tablet1*, respectively, have been successfully converted to Dalvik bytecode.

Observation 8 Conversion time for jar files ranging from 20 to 400 KiB does not exceed 80 seconds.

Conclusion 4 The Dx tool is a bottleneck of the tool chain. It can only correctly process 25 to 30% of the applications. The reason is that it puts every Java class in memory and suffers from the memory limitation of in-vivo processing, similarly to Soot. This tool is used off the shelf and could be optimized to run on devices where resources are limited, by processing class after class to limit memory consumption. .

5.7 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. We only
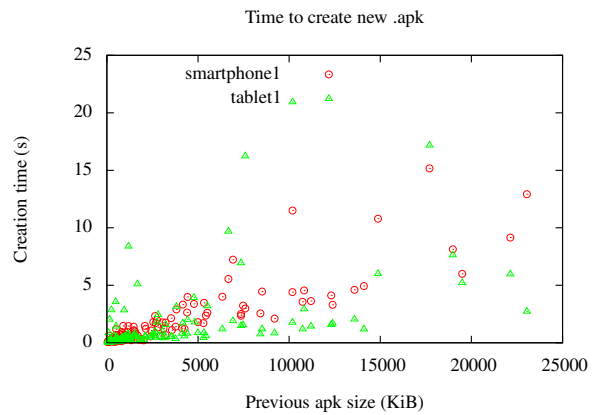

Fig. 10 Creation Time of a New `apk` File In-Vivo

have 39/130 applications that have been correctly processed in the previous steps. At every step, some applications failed. For the remaining 91/130 applications where the final instrumented Dalvik bytecode could not be computed, we take as input the original Dalvik 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 9 121/130 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`. Only 9/130 applications generate an exception because their size is too big and can thus not be processed by the zip utility.

Observation 10 For 95% of the applications it takes less than five seconds regardless of the device and of the size of the original `apk` file.

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.

There is no linear relation with the Dalvik size as it is the case in Fig. 6 and 9. This is probably due to the fact that when generating apk files, others factors than the bytecode size come into play, such as handling the media files (images, sound, etc.), which sometimes dominate the Dalvik bytecode size.

5.8 Signing the Generated `apk` File

Signing time of applications is represented in Figure 11.

Observation 11 120/130 Android applications were successfully signed on *tablet1*. There is no clear relation between the size of the `apk` file and the signature time of the `apk` file. 14/130 applications generate an exception because
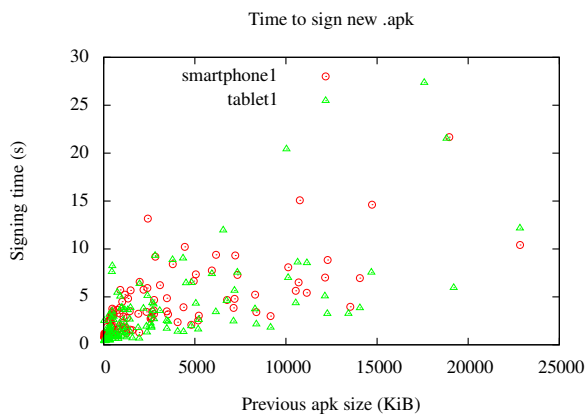
Fig. 11 Performance of In-Vivo Signature of the Instrumented Apk File

their size is too big and can thus not be processed (14 on *smartphone2* and 10 on *tablet1*).

Observation 12 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.

5.9 Conclusion

We now recapitulate the results of our experiments of in-vivo modification of Android applications.

5.9.1 Feasibility

Table 3 summarizes all the experiments for *smartphone2* and highlights the feasibility of the whole approach.

Total execution times for all steps of the toolchain are computed for the Soot and ASM version. For an ASM-based instrumentation it takes a median time of 120 seconds, that is 2 minutes, 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 better privacy. During those 2 minutes the phone is still usable since only once core is used (most smartphones feature multi-core CPUs) and only the maximum amount of heap memory allowed by the virtual machine can be used (and not all memory).

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.

5.9.2 How to Improve Performance of In-Vivo Instrumentation?

According to our analysis, the main blocking factor is the memory. The maximum heap size required to analyze and transform applications is an issue for many transformation steps. We think that this issue can easily be 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 48 MiB).

Dalvik to Java conversion and Java to Dalvik conversion are two very time expensive steps. They use unmodified versions of Dex2jar and dx. There are two ways to overcome those resource-hungry tools.

First, those tools were never optimized to run on platforms with limited resources. We believe that there are many optimization opportunities in terms of CPU and memory consumption.

Second, one could replace those tools by better alternatives. For instance, an ASM-like library for manipulating Dalvik bytecode would allow to skip Dalvik-to-Java and Java-to-Dalvik conversion. Such tools are emerging such as ASMdex[7]. Another solution would consist of performing bi-directional transformations directly from Dalvik bytecode to Jimple bytecode which are both register based. We are indeed working on a Dalvik to Jimple translation prototype called Dexpler [15].

To sum up, our results show that we can reasonably imagine to manipulate the bytecode on 100% of our dataset applications within at most 5 minutes.

5.9.3 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 300 KiB. 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.

---

[7] See `http://asm.ow2.org/asmdex-index.html`

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 Metrics of Our In-Vivo Instrumentation Process for Smartphone2. There are problematic steps but the overall process is feasible.

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 Metrics of Our In-Vivo Instrumentation Process for (for Tablet1)

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 and consume much more memory. However, for the use cases presented in Section 2, the instrumentation only consists in monitoring and proxying Java methods.

6 Related Work

Monitoring Applications Monitoring smartphone applications at runtime is an idea which recently emerged, due to the explosion of "mobile" malware and the increasing sophistication of mobile OS.

Bose et al. [16] 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.

Enck et al. [17] 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. [18] extends the Java virtual machine for mobile devices (Java ME) for adding runtime monitoring capabilities. On the con-trary, our feasibility study indicates that it is possible to achieve runtime monitoring in an unmodified Android system.

Recently, Burgera et al. [19] 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 level, 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. [20] 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. [5] 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. [4] made the case for an 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 Erlingsson et al. and Evans et al. [10, 11, 12] were the first to manipulate bytecode to weave a security policy directly in a Java programs. Their Inline Reference Monitor (IRM) technique allows (1) to completely separate the program development from the policy definition and (2) to have a policy mechanism independent of the Java Virtual Machine on which the program is running. We also weave the security policy directly in Android applications, obtaining robust Android applications whose security policy is independent of the Android system on which they are running.

Closest to our work are two Dalvik bytecode manipulation systems: I-Arm Droid [21] and Mr. Hide [22]. The main difference is that our approach runs in-vivo whereas theirs does not.

In-vivo bytecode manipulation is also achieved by App-Guard [23, 24]. However, the approach is based on *dexlib* a bytecode manipulation library which does not offer an abstract representation like Jimple with Soot. Thus, more advanced reasoning on the bytecode (on graphs for instance) is difficult with their approach.

Redirecting methods of interest to a monitor is the basic of IRM. Von Styp-Rekowsky et al. present a novel approach by modifying the equivalent of Dalvik function pointers at runtime [25]. Such an approach reduces the overhead and could easily be adopted by our fine-grained permission system.

Xu et al. present Aurasium [26], another permission management system. It does operate at the level of C libraries and redirect low level functions of interest to the monitor. Operating at this low level makes it difficult to inject fake values and to differentiate between normal and Java-level security relevant operations.

Reddy et al. [27] 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 resource 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. [28] 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.

7 Conclusion

The toolchain we propose and evaluate in this paper is a milestone that respond to the recent claim of Stravou et al. [2] 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 hard-coded 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.

References

[1]   Gartner. *Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent.* http://www.gartner.com. 2012.

[2]   A. Stavrou et al. "Building Security into Off-the-Shelf Smartphones". In: *Computer* 45-2 (2012).

[3]   A. Bartel et al. *Improving privacy on android smartphones through in-vivo bytecode instrumentation.* Technical Report. http://hal.archives-ouvertes.fr/docs/00/70/03/19/PDF/hal.bartel.pdf. 2012.

[4]   P. Pearce et al. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *Proceedings of AsiaCCS.* Seoul, Korea, May 2012.

[5]   S. Shekhar, M. Dietz, and D. S. Wallach. "AdSplit: Separating smartphone advertising from applications". In: *CoRR* abs/1202.4030 (2012).

[6]   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. Bern, Switzerland: ACM, 2012, pp. 29–42.

[7]   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. Chicago, Illinois, USA: ACM, 2009, pp. 235–245.

[8]   P. Lam et al. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011).* 2011.

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

[10] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* Tech. rep. Cornell University, 2003.

[11] U. Erlingsson and F. B. Schneider. "IRM Enforcement of Java Stack Inspection". In: *IEEE Symposium on Security and Privacy.* IEEE Computer Society. 2000, pp. 246–255.

[12] D. Evans and A. Twyman. "Flexible Policy-Directed Code Safety". In: *Proceedings of the IEEE Symposium on Security and Privacy.* 1999, pp. 32–45.

[13] A. Felt et al. *Android permissions demystified.* Tech. rep. UCB/EECS-2011-48. University of California, Berkeley, 2011.

[14] A. Bartel et al. "Automatically securing permission-based software by reducing the attack surface: An application to Android". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM. 2012, pp. 274–277.

[15] A. Bartel et al. "Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis.* ACM. 2012, pp. 27–38. 2237315.

[16] A. Bose et al. "Behavioral detection of malware on mobile handsets". In: *MobiSys.* 2008, pp. 225–238.

[17] W. Enck et al. "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.

[18] G. Costa et al. "Runtime monitoring for next generation Java ME platform". Anglais. In: *Computers & Security / Computers and Security* 29.1 (2010), pp. 74–87.

[19] 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. Chicago, Illinois, USA: ACM, 2011, pp. 15–26.

[20] B. Davis et al. "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications". In: *IEEE Mobile Security Technologies (MoST).* San Francisco, CA, May 24, 2012.

[21] B. Davis et al. "I-arm-droid: A rewriting framework for in-app reference monitors for android applications". In: *IEEE Mobile Security Technologies (MoST), San Francisco, CA* (2012).

[22] J. Jeon et al. "Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android". In: (2011).

[23] M. Backes et al. "Appguard-real-time policy enforcement for third-party applications". In: (2012).

[24] M. Backes et al. "AppGuard–Enforcing User Requirements on Android Apps". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2013, pp. 543–548.

[25] P. von Styp-Rekowsky et al. "Idea: callee-site rewriting of sealed system libraries". In: *Engineering Secure Software and Systems.* Springer, 2013, pp. 33–41.

[26] R. Xu, H. Saïdi, and R. Anderson. "Aurasium: Practical policy enforcement for android applications". In: *Proceedings of the 21st USENIX Security Symposium.* 2012.

[27] N. Reddy et al. *Application-centric security policies on unmodified Android.* Tech. rep. Technical Report 110017, University of California, Los Angeles, 2011.

[28] 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.