Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy

Vitor Afonso*, Antonio Bianchi†, Yanick Fratantonio†, Adam Doupé‡,
Mario Polino§, Paulo de Geus¶, Christopher Kruegel†, and Giovanni Vigna†
*CAPES Foundation
Email: vitor@lasca.ic.unicamp.br
†UC Santa Barbara
Email: {antoniob, yanick, chris, vigna}@cs.ucsb.edu
‡Arizona State University
Email: doupe@asu.edu
§Politecnico di Milano
Email: mario.polino@polimi.it
¶University of Campinas
Email: paulo@lasca.ic.unicamp.br

***Abstract*—Current static analysis techniques for Android applications operate at the Java level—that is, they analyze either the Java source code or the Dalvik bytecode. However, Android allows developers to write code in C or C++ that is cross-compiled to multiple binary architectures. Furthermore, the Java-written components and the native code components (C or C++) can interact.**

Native code can access all of the Android APIs that the Java code can access, as well as alter the Dalvik Virtual Machine, thus rendering static analysis techniques for Java unsound or misleading. In addition, malicious apps frequently hide their malicious functionality in native code or use native code to launch kernel exploits.

It is because of these security concerns that previous research has proposed native code sandboxing, as well as mechanisms to enforce security policies in the sandbox. However, it is not clear whether the large-scale adoption of these mechanisms is practical: is it possible to define a meaningful security policy that can be imposed by a native code sandbox without breaking app functionality?

In this paper, we perform an extensive analysis of the native code usage in 1.2 million Android apps. We first used static analysis to identify a set of 446k apps potentially using native code, and we then analyzed this set using dynamic analysis. This analysis demonstrates that sandboxing native code with no permissions is not ideal, as apps' native code components perform activities that require Android permissions. However, our analysis provided very encouraging insights that make us believe that sandboxing native code can be feasible and useful in practice. In fact, it was possible to automatically generate a native code sandboxing policy, which is derived from our analysis, that limits many malicious behaviors while still allowing the correct execution of the behavior witnessed during dynamic analysis for 99.77% of the benign apps in our dataset. The usage of our system to generate policies would reduce the attack surface available to native code and, as a further benefit, it would also enable more reliable static analysis of Java code.

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23384

I. Introduction

Mobile operating systems allow third-party developers to create applications (hereafter referred to as *apps*) that extend the functionality of the mobile device. Apps span across all categories of use: banking, socializing, entertainment, news, health, sports, and travel.

Google's Android operating system currently enjoys the largest market share, currently at 84.7%, of all current smartphone operating systems [25]. The official app market for Android, the Google Play Store, has around 1.4 million available apps [2] (according to AppBrain, a third-party Google Play Store tracking site) with over 50 billion app downloads [38].

Android apps are typically written in Java, and then compiled to bytecode that runs on an Android-specific Java virtual machine, called the Dalvik Virtual Machine (DVM).[1] These apps can interact with the filesystem, the Android APIs (to access phone features such as GPS location, call history, microphone, or SMS messages), and even other apps.

The wealth of information stored on smartphones attracts miscreants who want to steal the user's information, send out

[1]In recent versions, the bytecode is instead compiled and executed by a new runtime, called ART. For simplicity, in the rest of the paper we will only refer to the DVM. However, everything we describe conceptually applies to ART as well.

premium SMS messages, or even have the user's device join a botnet [10].

Static analysis of Android applications has been proposed by various researchers to check the security properties of the apps that the user installs [5], [7], [17], [18], [22], [23], [28], [29], [39], [42]–[45].

All the proposed static analysis techniques for Android apps have operated at the Java level—that is, these techniques process either the Java source code or the Dalvik bytecode. However, Android apps can also contain components written in *native code* (C or C++) using the Android NDK [19]. Some of the reasons why developers might use this feature, as stated by the NDK documentation [19], are:

> For certain types of apps, [native code] can be helpful so you can reuse existing code libraries written in these languages, but most apps do not need the Android NDK.
>
> Typical good candidates for the NDK are CPU-intensive workloads such as game engines, signal processing, physics simulation, and so on.

Using the NDK, the C or C++ code will be compiled and packaged with the app. Android provides an interface (JNI) for Java code to call functions of native code and vice versa.

While attempting to allow native code in Android apps is noble, there are serious security implications of allowing apps to execute code outside the Java ecosystem.

The existence of native code severely complicates static analysis of Android apps. First, to our knowledge, no static analysis of Android apps attempts to statically analyze the native code included in the app. Thus, malware authors can include the malicious payload/behavior in a native code component to evade detection. Furthermore, the native code in an Android app has *more capabilities* than the Java code. This is because the native code has direct access to the memory of the running process, and, because of this access, can read and modify the Dalvik Virtual Machine and its data.[2] Effectively, this means that the native code can completely modify and change the behavior of the Java code—*rendering all static analysis of the Java code unsound.*

In light of these security problems with native code usage in Android applications, researchers have turned to sandboxing mechanisms, which limit the interaction between the native code and the Java code [8], [33], [35]. This follows the least-privilege principle: The native code does not need full access to the Java code and thus should be sandboxed.

A native code sandbox should be *security-relevant* and *usable* with benign, real-world apps. These requirements result in the following properties:

- **Least-Privilege**: The native code of the app should have access only to what is strictly required, thus reducing the chances the native component could extensively damage the system.
- **Compartmentalization**: The native code of the app should communicate with the Java part only using specific, limited channels, so that the native component cannot modify, interact with, or otherwise alter the Java runtime and code in unexpected ways.
- **Usability**: The restrictions enforced by the sandbox must not prevent a significant portion of benign apps from functioning.
- **Performance**: The sandbox implementation must not impose a substantial performance overhead on apps.

Even though previous research has focused on the *mechanism* of native code sandbox enforcement [33], [35], to this point no research has focused on *how to generate a security policy* that a sandbox can enforce so that the policy is be both practical (i.e., it would not break benign apps) and useful (i.e., it would limit malicious behaviors).

Sun and Tan [35], in their paper presenting the native code sandboxing mechanism NativeGuard, state:

> We decide to follow a heuristic approach and by default grant no permission to the [sandboxed native code] in NativeGuard. The approach is motivated by the observation that it is rare for legal native code to perform privileged operations, as it is a "bad practice" according to the NDK.

Sun and Tan are correct that the NDK considers native code performing privileged operations to be bad practice, however, *we need data to confirm this intuition.* We must know: what is the native code in real-world apps doing? How do real-world apps use native code? For instance, what if native code is used to perform exactly the same actions as Java code? In this case, it would not be possible to meaningfully constrain the permission of native code components, and enforcing the least-privilege principle would not grant any security benefits. We also need clarification as to how tightly coupled the communication is between the native code and the Java code. Enforcing compartmentalization might break or negatively affect tightly-coupled apps.

To answer these questions, we perform a large-scale analysis of real-world Android apps. Specifically, we look at how apps use native code, both statically and dynamically. We statically analyze 1,208,476 Android apps to see if they use native code, then we dynamically analyze the 446,562 that were determined to use native code. Our system is able to monitor the dynamic execution of an app, while recording activities performed by its native code components (e.g., invoked system calls, interactions between native and Java components). From this analysis, we shed light on *how real-world Android apps use native code.*

In addition, our dynamic analysis system can be used to generate a native code sandboxing policy that allows for normal execution of the native code behaviors observed during the dynamic analysis of a set threshold of apps, while reducing the attack surface and thus limiting many malicious behaviors (e.g., root exploits) of malicious apps.

The main contributions of this paper are the following:

[2]Even if the Dalvik Virtual Machine memory is initially mapped as read-only, a native code component can change the memory permission by using the `mprotect` syscall.

- We develop a tool to monitor the execution of native components in Android applications and we use this tool to perform the largest (in terms of number of apps and detail of information acquired) study of native code usage in Android.
- We systematically analyze the collected data, providing actionable insights into how benign apps use native code. Moreover, we release the full raw data and we make it available to the community [1].
- Our results show that completely eliminating permissions of native code is not ideal, as this policy would break, as a lower bound, 3,669 of the apps in our dataset. However, we propose that our dynamic analysis system can be used to derive a native code sandboxing policy that limits many malicious behaviors, while allowing the normal execution of the native code behaviors observed during the dynamic analysis of a set threshold of apps (99.77% in our experiment).

II. Background

To understand the analysis that we perform on Android applications and our proposed policy, it is necessary to review the Android security mechanisms, how native code is used in Android, the damage that malicious native code can cause, and the previously proposed native code sandboxing mechanisms.

A. Android Security Mechanisms

When apps are installed on an Android phone, they are assigned a new user (`UID`) and groups (`GIDs`) based on the permissions requested by the app in its manifest. Every app is executed in a separate process, which is a child of `Zygote`, a process started when the system is initialized. Moreover, inter-process communication is done using intents which all flow through an Android system-level process called Binder [11].

On Android, some operations and resources are protected by permissions. Apps must declare the permissions needed in the manifest, and at installation time the requested permissions are presented to the user, who decides to continue or cancel the installation. Permissions are enforced app-wise using Linux access-control mechanisms and by system services that check if the app is allowed to access certain resources or perform the requested operation [16].

B. Native Code

Native code in Android apps is deployed in the app as ELF files, either executable files or shared libraries. There are four ways in which the Java code of an Android app can execute native code: Exec methods, Load methods, Native methods, and Native activity.

Exec methods. Executable files can be called from Java by two methods, namely `Runtime.exec` and `ProcessBuilder.start`. Hereinafter we refer to these methods as *Exec methods.*
Load methods. Native code in shared libraries can be loaded by the framework when a NativeActivity is declared in the manifest, along with its library name, or by the app through the following Java methods, which are hereinafter referred to as *Load methods:* `System.load`, `System.loadLibrary`, `Runtime.load`, and `Runtime.loadLibrary`. Native code in shared libraries can be invoked at loading time, through calls to native methods and through callbacks in native activities. When a library is loaded, its `_init` and `JNI_OnLoad` functions are called.
Native methods. Native methods are implemented in shared libraries and declared in Java. When the Java method is called, the framework executes the corresponding function in the native component. This mapping is done by the Java Native Interface (JNI) [21]. JNI also allows native code to interact with the Java part to perform actions such as calling Java methods and modifying Java fields.
Native activity. Native code is invoked in native activities using activities' callback functions, (e.g., `onCreate` and `onResume`), if defined in a native library.

C. Malicious Native Code

Malicious apps can use native code to hide malicious actions from static analysis of the Java portion of the app. These actions can be calls to methods in Java libraries, such as sending SMS messages, or complex attacks that involve exploiting the kernel or privileged processes to compromise the entire OS. These root exploits are possible because native code is allowed to directly call system calls. Another possible way that attackers can directly call system calls to execute root exploits is by exploiting vulnerabilities in native code used by benign apps.

As previous research has shown [35], because native code shares the same memory address space as the Dalvik Virtual Machine, it can completely modify the behavior of the Java code, rendering static analysis of the Java code fundamentally unsound. For instance, malicious code can use functions exported by *libDVM.so* to identify where the bytecode implementing a specific Java method is placed in memory. At this point, the native code can dynamically replace the method at run time.

D. Native Code Sandboxing Mechanisms

Several approaches have been proposed to sandbox native code execution. For instance, NativeGuard [35] and Robusta [33] move the execution of native code to a separate process. Two complementary goals are obtained: (1) the native code cannot tamper with the execution of the Java code and (2) different security constraints can be applied to the execution of the native code.

Communication between the Java code and the native code is then ensured by modifying the JNI interface to make the two processes communicate through an OS-provided communication channel (e.g., network sockets).

While moving native code to a separate process is a natural mechanism to achieve the aforementioned goals (because it relies on OS-provided security mechanisms, such as process memory separation or process permissions), other solutions are possible. For instance, thread-level memory protection (as proposed in Wedge [8]). However, applying this solution in Android would require significant modifications to the underlying Linux kernel.

TABLE I. RESULTS OF THE STATIC ANALYSIS.

Apps	Type
267,158	Native method
42,086	Native activity
288,493	Exec methods
242,380	Load methods
221,515	ELF file
446,562	At least one of the above

III. ANALYSIS INFRASTRUCTURE

We designed and implemented a system that dynamically analyzes Android applications to study how native code is used and to automatically generate a native code sandboxing policy. Our analysis consists of an instrumented emulator, and it records all events and operations executed from within native code, such as invoked syscalls and native-to-Java communication. The dynamic instrumentation is completely generic, and it allows the usage of any manual or automatic instrumentation tool. The version of the Android system used was 4.3.

Since our goal was to obtain a comprehensive characterization of native code usage in real world applications, we used a corpus of 1,208,476 distinct—different package names and APK hashes—free Android apps that we have continuously downloaded from the Google Play store from May 2012–August 2014. The age of the apps varies throughout the timeframe, as we currently do not download new versions of apps.

A. Static Prefiltering

Performing dynamic analysis of all 1,208,476 apps by running each app would take a considerable amount of time; therefore, by using static analysis, we filtered the apps that had some indication of using native code. The characteristics we looked for in the apps are the following: having a native method, having a native activity, having a call to an Exec method, having a call to a Load method, or having an ELF file inside the APK.

We used the Androguard tool [12] as a basis for the static analysis. To identify native methods we searched for methods declared in the Dalvik bytecode with the modifier[3] "native." Native activities were identified by two means: (1) looking for a NativeActivity in the manifest and (2) looking for classes declared in the Dalvik bytecode that extend NativeActivity. Finally, calls to Exec and Load methods were identified by investigating method invocations in the bytecode.

Of the 1,208,476 apps statically analyzed, 446,562 apps (37.0%) used at least one of the previously mentioned ways of executing native code. Table I presents the number of apps that use each of these characteristics.

B. Dynamic Analysis System

Now that we have identified *which* Android apps use native code, we now want to understand *how* apps use native code. During the dynamic analysis we monitor several types of actions performed by the analyzed apps, including system calls, JNI calls, Binder transactions, calls to Exec methods, loading of third-party libraries, calls to native activities' native callbacks, and calls to native methods. The system calls were captured using the `strace` tool, and the other information we obtained through instrumentation.

To monitor JNI calls, calls to native methods, and library loading, we modified `libdvm`. However, we do not want to monitor all JNI calls, just JNI calls to the app's native code, rather than calls to native code in the standard libraries that Android includes. To avoid monitoring JNI calls in standard libraries and calls to native methods in standard libraries, we modified the "Method" structure to include a property indicating whether it belongs to a third-party library or not. When a third-party library is loaded, this property is set accordingly.

We modified `libbinder` to track and monitor Binder transactions. We record the class of the remote function being called and the number that identifies the function. To map the identifiers to function names, we parse the AIDL (Android Interface Definition Language) files and source files that define Binder interfaces. To find files that have such definitions, we search for uses of the macros `DECLARE_META_INTERFACE` and `IMPLEMENT_META_INTERFACE` and classes that extend "IInterface." Furthermore, to match identification numbers to names, we search in ".cpp" files for enumerations that use `IBinder::FIRST_CALL_TRANSACTION` and, in ".java" files, for variables defined using `IBinder.FIRST_CALL_TRANSACTION`. We use the names assigned `FIRST_CALL_TRANSACTION` as the functions with identifier 1, the ones assigned `FIRST_CALL_TRANSACTION + NUM` as the functions with identifier 1+NUM and, for the enumerations that only use `FIRST_CALL_TRANSACTION` to define the first element, we consider they are increasing the identifier one by one.

Calls to Exec methods are identified by instrumenting `libjavacore`. Finally, to monitor the use of native callbacks in native activities, we modified `libandroid_runtime`.

We determine which actions were performed by native code and which by Java code after the dynamic analysis. To make this determination, we observe when threads change execution context from Java to native and from native to Java. Thus, we process all system calls, keeping a list of threads that are executing native code. We add a thread to this list when one of the following happens: Exec method is executed—we add the child process, which is then used to call `execve`, a custom (third-party) shared library is loaded, a native method is executed, or a callback in the native component of a native activity is executed. When these actions are completed and the execution control changes back to Java, the thread is removed from the list.

We also remove a thread from the list when one of the JNI methods in Table II is executed. The `Call*<TYPE>` functions are used to call Java methods, and the `NewObject*` functions are used to create instances of classes, which results in the execution of Java constructors. When these methods return, the thread is placed back on the list. Additionally, we remove a thread from the list when the `clinit` method, which

[3]Modifier here is an attribute of a method, similar to `public`. An example Dalvik method signature would be: `.method public native example()`.

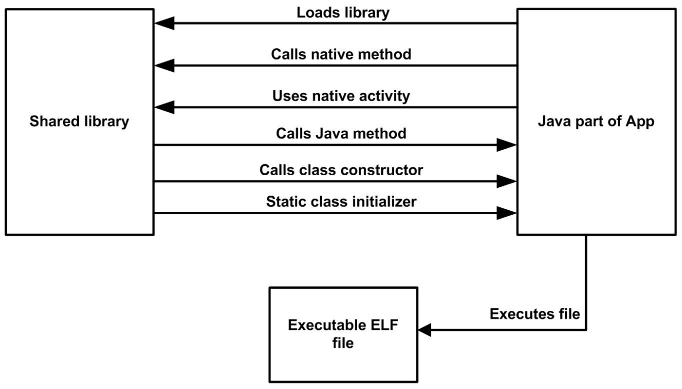

Fig. 1. Possible transitions between native code and Java.

TABLE II. JNI METHODS THAT CAUSE A TRANSITION FROM NATIVE TO JAVA. <TYPE> CAN BE THE FOLLOWING: OBJECT; BOOLEAN; BYTE; CHAR; SHORT; INT; LONG; FLOAT; DOUBLE; VOID.

Call<TYPE>Method
CallNonVirtual<TYPE>Method
Call<TYPE>MethodA
CallNonVirtual<TYPE>MethodA
Call<TYPE>MethodV
CallNonVirtual<TYPE>MethodV
CallStatic<TYPE>Method
CallStatic<TYPE>MethodA
CallStatic<TYPE>MethodV
NewObject
NewObjectV
NewObjectA

is the static initialization block of a class, is executed. Figure 1 presents all mentioned transitions.

To understand how isolating the native code from the Java code would impact the performance of the apps, we also monitor the amount of data exchanged between native and Java code. We measured the amount of data passed in parameters of calls from native code to Java methods and vice versa, as well as the size of the returned value. We also capture the size of data used to set fields in Java objects. The results of this analysis are presented in Section IV-B.

IV. Evaluation and Insights

We ran both the static pre-filter and dynamic analysis across numerous physical machines and private-cloud virtual machines. In total, we used 100 cores and 444 GB of memory. Moreover, the analysis was run in parallel.

The dynamic analysis was performed using an instrumented Android emulator (as described in the previous section), and to keep the analysis time feasible we limited the analysis to two minutes for each app. To dynamically exercise each application, we followed an approach similar to what is used in Andrubis [40]: we used the Google Monkey [20] to stimulate the app with random events, and we then automatically generated a series of targeted *events* (by means of sending properly-crafted intents) to stimulate all activities, services, and broadcast receivers defined in the application.

Ideally, it would have been possible to use more sophisticated dynamic instrumentation systems. However, the large scale of our analysis motivated our choice to use a simpler approach, as it would have required a prohibitive amount of resources to run on hundreds of thousand of apps. While our dynamic instrumentation system is acceptable for the purposes of understanding the lower bound on what behaviors native code performs, the incompleteness inherent in dynamic analysis can affect the native code policies generated by our system. However, if Google or another large company were to adopt the idea of using a dynamic analysis system to automatically generate a native code security policy, they could use substantial resources to run the applications for longer periods of time, use sophisticated dynamic analysis approaches [32], or even introduce the instrumentation into the Android operating system and sample the behaviors from real-world devices.

During dynamic analysis, 33.6% (149,949) of the apps identified by static analysis as potentially having native code actually executed the native code. Table III presents the number of apps that executed each type of native code. These numbers constitute a lower bound of the apps that could actually execute native code.

In order to understand, for our study, why the native code was not reached during dynamic analysis, we manually analyzed, statically and dynamically, 20 random apps that were statically determined to have native code. For 40% (8) of them, we established through analysis of the decompiled code that the native code was unreachable from Java code (also known as deadcode). The remaining applications were too complex to be manually inspected, and we were not able to ascertain whether the native code components were not reached due to deadcode. For this reason, we dynamically analyzed and manually interacted with them and we did not find any path that led to the execution of the native code. Thus, we believe that also in this case the native code component was not reached due to deadcode, even if we were not able to be completely certain, due to the incomplete nature of manual analysis.

We further investigated *why* there was deadcode in these apps. In each case, the native code was deadcode in third-party libraries. In fact, in our experience, it often happens that an app includes a third-party library, to then actively use only a (sometimes very limited) subset of its functionality, thus leading to deadcode. Hence, we expect this to be the case for many apps where our analysis did not reach native code. As an additional experiment, we also manually and extensively dynamically exercised another 20 random apps. We observed no cases of significant changes in the results compared to the Google Monkey automated analysis (neither additional native code components were reached nor more syscalls were called).

To further understand the coverage of our dynamic analysis system we performed two additional experiments, one measuring the Java method coverage and one measuring the native code coverage. Section VII discusses these experiments in depth.

TABLE III. THE NUMBER OF APPS THAT EXECUTED EACH TYPE OF NATIVE CODE.

Apps	Type
72,768	Native method
19,164	Native activity
132,843	Load library
27,701	Call executable file (27,599 standard, 148 custom and 46 both)
149,949	At least one of the above

TABLE IV. OVERVIEW OF ACTIONS PERFORMED BY CUSTOM SHARED LIBRARIES IN NATIVE CODE.

Writing log messages
Performing memory management system calls, such as `mmap` and `mprotect`
Reading files in the application directory
Calling JNI functions
Performing general multiprocess and multithread related system calls, such as `fork`, `clone`, `setpriority`, and `futex`
Reading common files, such as system libraries, font files, and "/dev/random"
Performing other operations on files or file descriptors, such as `lseek`, `dup`, and `readlink`
Performing operations to read information about the system, such as `uname`, `getrlimit`, and reading special files (e.g., "/proc/cpuinfo" and "/sys/devices/system/cpu/possible")
Performing system calls to read information about the process or the user, such as `getuid32`, `getppid`, and `gettid`
Performing system calls related to signal handling
Performing `cacheflush` or `set_tls` system calls or performing `nanosleep` system call
Reading files under "/proc/self/" or "/proc/<PID>/", where PID is the process' pid
Creating directories

A. *Native Code Behavior—An Overview*

We present in this section an overview of the actions performed by native code on Android. We split the actions into those performed by shared libraries (including those performed during library loading, native methods, and native activities) and those that are the result of invoking custom, executable, and binaries through Exec methods. We also present the actions performed using standard binaries (i.e., not created by the app), but in this case based on their names and parameters, instead of looking at the system calls.

94.2% (125,192) of the apps that used custom shared libraries executed only a set of common actions in native code, and Table IV contains the common actions.

TABLE V. TOP FIVE MOST COMMON ACTIONS PERFORMED BY APPS IN NATIVE CODE, THROUGH SHARED LIBRARIES (SL) AND CUSTOM BINARIES (CB). FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

SL	CB	Description
3,261	72	`ioctl` system call
1,929	39	Write file in the app's directory
1,814	35	Operations on sockets
1,594	5	Create network socket
1,242	144	Terminate process or thread group

TABLE VI. TOP FIVE MOST COMMON ACTIONS PERFORMED BY APPS THAT CALLED STANDARD BINARIES IN THE SYSTEM. FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

Apps	Description
19,749	Read system information
3,384	Write file in the app's directory or in the sdcard
3,362	Read logcat
1,041	List running processes
861	Read system property

The top five most common actions performed by apps in native methods, native activities, and custom binaries called through Exec are presented in Table V. Table VI presents the top five most common actions performed by the apps that used Exec to call standard (system) binaries.

By analyzing the system calls and the Java methods called from native code, we identified 3,669 apps that perform an action requiring Android permissions from native code. Table VII presents the top five most popular permissions used, how many apps use them, and how we detected its use. We used PScout [6] to compute the permissions required by each Java method. Comparing the permissions used in native code with the permissions requested by the app, we found that only 81 apps use, in native code, *all the permissions requested by the app*.

In addition to this being the first concrete look into how many apps use native code and what that native code does, we can draw two important conclusions: (1) if the native code is separated in a different process, it is necessary to give some permissions to the native code and (2) the permissions of the native code can be more strict (less permissive) than the permissions of the Java code.

It is interesting to note how conclusion (1) shows that the drastic measure adopted in NativeGuard [35], which does not grant any permissions to the native code, would break 3,669 of apps. This observation reinforces even more our belief that security policies should be generated following a data-driven approach. For instance, a reasonable tradeoff would be to allow to the native code only the `INTERNET`, `WRITE_EXTERNAL_STORAGE`, and `READ_EXTERNAL_STORAGE` permissions (the three most commonly used in native code), thus blocking only 152 applications.

B. *Java—Native Code Interactions*

To better understand the performance implications of separating the native code from the Java code of the apps, we measured the number of interactions per millisecond between Java and native code, i.e., the number of calls to JNI functions, calls to native methods, and Binder transactions.

The mean of interactions per millisecond is 0.00142, whereas the variance is 0.00003 and the maximum value is 0.22. NativeGuard's [35] performance evaluation with the Zlib benchmark shows a 34.36% runtime overhead for 9.81 interactions per millisecond and 26.64% for 3.96 interactions per millisecond. Therefore, our experiment shows that isolating

TABLE VII. THE FIVE MOST COMMON (BY NUMBER OF APPS) ACTIONS IN NATIVE CODE THAT REQUIRE ANDROID PERMISSION. FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

Apps	Permission	Description
1,818	INTERNET	Open network socket or call method `java.net.URL.openConnection`
1,211	WRITE_EXTERNAL_STORAGE	Write files to the sdcard
1,211	READ_EXTERNAL_STORAGE	Read files from the sdcard
132	READ_PHONE_STATE	Call methods `getSubscriberId`, `getDeviceSoftwareVersion`, `getSimSerialNumber` or `getDeviceId` from class `android.telephony.TelephonyManager` or Binder transaction to call `com.android.internal.telephony.IPhoneSubInfo.getDeviceId`
79	ACCESS_NETWORK_STATE	Call method `android.net.ConnectivityManager.getNetworkInfo`

TABLE VIII. TOP FIVE MOST COMMON TYPES OF COMMAND PASSED WITH THE "-C" ARGUMENT TO `SU`, SEPARATED BETWEEN THE APPS THAT MENTION THEY NEED ROOT PRIVILEGES IN THEIR DESCRIPTION OR NAME AND THE ONES THAT DO NOT MENTION IT. FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

Does not Mention Root	Does Mention Root	Description
12	10	Custom executable (e.g., su -c sh /data/data/com.test.etd062.ct/files/occt.sh)
1	13	Reboot
2	12	Read system information
1	8	Change permission of file in app's directory
1	7	Remove file in app's directory

native code in a different process should not have a substantial performance impact on average.

Additionally, we measure the number of bytes exchanged between the Java code and native code per second. The mean of bytes exchanged per second is 1,956.55 (1.91 KB/s) and the maximum value is 6,561,053.27 (6.26 MB/s). Only 11 apps exchanged more than 1 MB/s. We believe the amount of data exchanged between Java and native code would not incur a significant overhead, although it could vary greatly depending on the specific app.

C. Usage of the `su` Binary

Unlike common Linux distributions, in Android, users do not have access to a super user account and, therefore, are prevented from performing certain actions, such as uninstalling pre-installed apps. Thus, to have greater control over the system, many users perform a process known as "rooting," to be able to perform actions as the "root" user. Usually, during this process, a suid executable file called `su` is installed, as well as a manager app that restricts which apps can use this binary to perform actions as root. Because this process is so common among users, there are many apps that provide functionality that can only be performed by the root user, such as changing the fonts of the system or changing the DNS configuration.

Our analysis identified 1,137 apps that try to run `su`. Surprisingly, 28.23% (321) of these apps *do not mention in their description or in their name that they need root privileges*.

Some of these apps use the "-c" argument of `su` to specify a command to be executed as root. Table VIII presents the top five most common types of actions that these apps tried to execute using `su`, along with the number of apps that attempt to execute that command, and if the app mentioned that it requires root or not. This table gives insights into what the app is trying to accomplish as root. The table shows that the most common action used with the "-c" argument of `su` is calling a custom executable. Because apps cannot use `su` in the emulator, these actions did not work properly during dynamic analysis, so we cannot obtain more information on their behavior.

D. JNI Calls Statistics

Understanding the JNI functions called by native code can reveal how the native components of apps interact with the app and the Android framework. Table IX presents the types of JNI functions that were used by the apps and how many apps used them. The most relevant actions for security considerations in this table are: (1) calling Java methods and (2) modifying fields of objects. Calling methods in Java libraries from native code can be used to avoid detection by static analysis. Moreover, modifying fields of Java objects can change the execution of the Java code in ways that static analysis cannot foresee.

Calling Java methods, both from the Android framework and from the app can be performed by some of the methods presented in Table II, more precisely the ones whose name starts with "Call." As Table IX shows, we identified 35,231 apps that have native code which calls Java methods. More specifically, 24,386 apps used these functions to call Java methods from the app and 25,618 apps used them to call Java methods from the framework. Table X presents what groups of methods from the framework were called, along with the amount of apps that called methods in each group.

E. Binder Transactions

1.64% (2,457) of the apps that reached native code during dynamic analysis performed Binder transactions. Table XI presents the top five most commonly invoked classes of the remote methods. The most common class remotely invoked by this process is `IServiceManager`, which can be used to list services, add a service, and get an object to a Binder interface. All apps that used this class obtained an object to a Binder interface and two apps also used it to list services. This data shows that using Binder transactions from native code is not common. From a security perspective this is good as the use of Binder transactions represent a way in which native code can perform critical actions while staying undetected by static analysis.

TABLE IX. GROUPS OF JNI CALLS USED FROM NATIVE CODE.

Apps	Description
94,543	Get class or method identifier and class reference
71,470	Get or destroy JavaVM, and Get JNIEnv
53,219	Manipulation of String objects
49,321	Register native method
45,773	Manipulate object reference
41,892	Thread manipulation
35,231	Call Java method
19,372	Manipulate arrays
18,601	Manipulate exceptions
14,330	Create object instance
6,918	Modify field of an object
2,203	Manipulate direct buffers
47	Memory allocation
37	Enter or exit monitor

TABLE X. TOP 10 GROUPS OF JAVA METHODS FROM THE ANDROID FRAMEWORK CALLED FROM NATIVE CODE.

Apps	Description
7,423	Get path to the Android package associated with the context of the caller
6,896	Get class name
5,499	Manipulate data structures
4,082	Methods related to cryptography
3,817	Manipulate native types
3,769	Read system information
3,018	Audio related methods
2,070	Read app information
1,192	String manipulation and encoding
575	Input/output related methods
483	Reflection

F. *Usage of External Libraries*

Understanding the libraries used by the apps in native code can help us comprehend their purpose. Table XII presents the top 10 most used system libraries and Table XIII presents the top 10 must used custom libraries by apps in native code. It demonstrates that apart from the bitmap manipulation library, which was used by 16.6% (24,942) of the apps that reached native code, no standard library was used by a great number of apps. On the other hand, several custom libraries were used by more than 7.5% of the apps that executed native code.

V. SECURITY POLICY GENERATION

One step to limit the possible damage that native code can do is to isolate it from the Java code using the native code sandboxing mechanisms discussed in Section II-D. These mechanisms prevent native code from modifying Java code, which allows static analysis of the Java part to produce more reliable results. However, this is not enough, considering that the app can still perform dangerous actions—that is, by interacting with the Android framework/libraries and by using system calls to execute root exploits.

TABLE XI. TOP FIVE MOST COMMON CLASSES OF THE METHODS INVOKED THROUGH BINDER TRANSACTIONS. FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

Apps	Class
2,427	android.os.IServiceManager
740	android.media.IAudioFlinger
725	android.media.IAudioPolicyService
327	android.gui.IGraphicBufferProducer
303	android.gui.SensorServer

TABLE XII. TOP 10 MOST USED STANDARD LIBRARIES.

Apps	Name	Description
24,942	libjnigraphics.so	Manipulate Java bitmap objects
2,646	libOpenSLES.so	Audio input and output
2,645	libwilhelm.so	Multimedia output and audio input
349	libpixelflinger.so	Graphics rendering
347	libGLES_android.so	Graphics rendering
183	libGLESv1_enc.so	Encoder for GLES 1.1 commands
183	gralloc.goldfish.so	Memory allocation for graphics
182	libOpenglSystemCommon.so	Common functions used by OpenGL
182	libGLESv2_enc.so	Encoder for GLES 2.0 commands
181	lib_renderControl_enc.so	Encoder for rendering control commands

TABLE XIII. TOP 10 MOST USED CUSTOM LIBRARIES.

Apps	Name	Description
19,158	libopenal.so	Rendering audio
17,343	libCore.so	Used by Adobe AIR
16,450	libmain.so	Common name
13,556	libstlport_shared.so	C++ standard libraries
11,486	libcorona.so	Part of the Corona SDK, a development platform for mobile apps
11,480	libalmixer.so	Audio API of the Corona SDK
11,458	libmpg123.so	Audio library
11,090	libmono.so	Mono library, used to run .NET on Android
10,857	liblua.so	Lua interpreter
10,408	libjnlua5.1.so	Lua interpreter

Our goal here is to *reduce the attack surface* available to native code, by restricting the system calls and Java methods that native code can access. In particular, we propose to use our dynamic analysis system to generate *security policies*. A security policy represents the *normal* behavior, which can be seen as a sort of whitelist that represents the syscalls and Java methods that are *normally* executed from within native code components of benign applications. These policies also implicitly identify which syscalls and Java methods should be considered as *unusual* or *suspicious* (as they do not belong to the *common* syscalls), such as the ones used to mount root exploits.

One aspect to be considered is what action is taken when an unusual syscall is executed. Similar to the design choice adopted by SELinux, we envision two modes: permissive and enforcing. In permissive mode, the system would log and report the usage of unusual behavior, while in enforcing mode the system would block the execution of such unusual behavior

TABLE XV. SYMBOLS USED TO REPLACE THE ARGUMENTS OF SYSTEM CALLS.

<USER-PATH>	A file path in the apps' directory or in the sdcard
<SYS-PATH>	A file path different than the ones represented by <USER-PATH>
<URANDOM-DEV>	"/dev/random" or "/dev/urandom"
<ASHMEM-DEV>	"/dev/ashmem"
<LOG-DEV>	"/dev/log/system", "/dev/log/main", "/dev/log/events" or "/dev/log/radio"
<NEG INT>	A negative number
<STD IN/OUT/ERR>	A file descriptor equal 0, 1, or 2
<NON STD FD>	A file descriptor different than 0, 1, or 2
<POS INT>	An integer greater than 0

and stop the application. Depending on the context, it might make sense to use permissive or the more aggressive enforcing mode. As an alternative, one could selectively pick permissive or enforcing mode depending on whether the unusual syscall is well-known to be used by root exploits. The policy generation process for syscalls is described in Section V-A, while the one for Java methods is described in Section V-B. We discuss the possibilities and the implications of this choice in Section VI.

It is worth noting that while this will not guarantee perfect protection from attacks, by applying the security principle of *least privilege* to the native code, we gain the dual security benefits of (1) increasing the precision of Java static analysis and (2) reducing the impact of malicious native code.

A. System Calls

Based on the system calls performed by the apps in native methods, in native activities, during libraries loading, and by programs executed by Exec methods, our system can automatically generate a security policy of allowed system calls. To compile this list, we first normalize the parameters of the system calls and later iterate over them, selecting the ones performed by most apps, until the list of selected system calls is comprehensive enough to allow at least a (variable threshold) percentage of the apps that executed native code to run properly. In Android, inter-process communication is done through Binder. Native code can directly use Binder transactions to call methods implemented by system services. At the system call level, these calls are performed by the using the `ioctl` system call. To consider these actions in our automatically generated whitelist, we substitute `ioctl` calls to Binder with the Binder transactions performed by the apps.

To understand the possible policies that could be generated, we performed this process using a threshold (the percentage of apps that use native code whose dynamically-executed behavior would function properly when enforcing this policy) of 99%. Table XIV presents the actions obtained by this procedure. The system call arguments that were normalized were replaced by symbols in the form <*> and * (meaning anything). Some of the arguments that are file descriptors were changed to a file path representation of it. All arguments that were not normalized represent a numeric value or a constant value that was converted by `strace` to a string representation. For the system calls that do not have the arguments next to it in the policies, the policy accepts calls with any arguments. Table XV presents more details about the symbols used.

To better understand which types of apps would be blocked by our example policy (when in enforcing mode), we studied them and manually analyzed a subset of them. The findings of this analysis are presented in Section VI.

The policies restrict the possible actions of native code, thus following the principle of *least privilege* and making it harder for malicious apps to function. Previously, malicious code could easily hide in native code to evade static analysis. With our example policies enforced by a sandboxing mechanism, the native code does not (depending on the exact threshold) have the ability to perform any malicious actions in native code, and therefore attackers will have to move the malicious behavior to the Java code, where it can be found by existing Java static analysis tools. Furthermore, the policies do not prevent the correct execution of the dynamically-executed behavior of many benign apps. Using the rules generated with the 99% threshold, only 1,483 apps (0.12% of the total apps in our dataset) would be affected. Of course, as the dynamic analysis performed by our system is incomplete (in that it can not execute all possible app code), this number is a lower bound. This can be alleviated by an organization wishing to use our system in one of two ways: (1) increase the completeness of the dynamic analysis or (2) deploying the sandboxing enforcement mechanism in reporting mode. Both choices will reveal more app behaviors.

Another benefit of enforcing a native code sandboxing policy is that it would prevent the correct execution of several root exploits. For this work, we considered the 13 root exploits reported in Table XVI. These exploits require native code to be successful. Our example security policy would hinder the execution of 10 of them. This follows because the policies attempt to reduce the attack surface of the OS for native code, while at the same time maintaining backward compatibility. Table XVI presents which of the considered exploits are successfully blocked, along with which entry of the policy they violate.

The root exploits that are prevented by our example security policy are blocked due to rules related to four system calls, namely `socket`, `perf_event_open`, `symlink`, and `ioctl`. More precisely, two exploits need to create sockets with `PF_NETLINK` domain and `NETLINK_KOBJECT_UEVENT` (15) protocol, however, the rules only allow `PF_NETLINK` sockets with protocol 0. One of the exploits needs the `perf_event_open` system call, which is not allowed by the policy. Two exploits need to create symbolic links that target system files or directories, but the policy only allows symbolic links to target "USER-PATH," which means files or directories in the app's directory or in the SD Card. Finally, five exploits use `ioctl` to communicate with a device. One of the rules allows `ioctl` calls to any device, namely `ioctl(<NON STD FD>,SNDCTL_TMR_TIMEBASE or TCGETS,*)`. However, this rule specifies the valid request value (the second parameter), whereas the exploits use different values, therefore they would be blocked.

The table also reports the details about the three exploits that would not be currently blocked. In one case (CVE-2011-1149), the exploit would still work because our example policy allows the invocation of the `mprotect` syscall, since it is used by benign applications. In the two remaining cases (RATC and Zimperlinch), the exploits rely on repeatedly invoking the `fork` syscall to exhaust the number of available processes.

TABLE XIV. Allowed system calls automatically generated using a threshold of 99% apps unaffected by the policy.

accept(*,*,*)	access(<SYS-PATH>, F_OK)	access(<SYS-PATH>,R_OK)
access(<SYS-PATH>, W_OK)	access(<SYS-PATH>,X_OK)	access(<USER-PATH>, F_OK)
access(<USER-PATH>,R_OK)	access(<USER-PATH>, R_OK|W_OK|X_OK)	bind
BINDER(android.os.IServiceManager. CHECK_SERVICE_TRANSACTION)	brk	cacheflush(*,*,0,*,*)
cacheflush(*,*,0,0,*)	chdir	chmod(<USER-PATH>,*)
clone(child_stack=*,flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM)		
connect(*,{sa_family=AF_UNIX, path=@"jdwp-control"},*)	connect(*,{sa_family=AF_INET,*,*},*)	connect(*,{sa_family=AF_UNIX, path=@"android:debuggerd"},*)
connect(*,{sa_family=AF_UNIX, path=<SYS-PATH>},*)	dup	dup2
epoll_create(*)	epoll_ctl(*,*,*,*)	epoll_wait
execve	exit(<NEG INT>)	exit(0)
exit_group(<POS INT>)	exit_group(0)	fcntl64(<NON STD FD>,F_DUPFD,*)
fcntl64(<NON STD FD>,F_GETFD)	fcntl64(*,F_GETFL)	fcntl64(<NON STD FD>,F_SETFD,*)
fcntl64(<NON STD FD>,F_SETFL,*)	fcntl64(<NON STD FD>,F_SETLK,*)	fdatasync(*)
fork	fstat64	fsync(*)
ftruncate(*,*)	futex	getcwd
getegid32	geteuid32	getgid32
getpeername	getpgid(0)	getpid
getppid	getpriority(PRIO_PROCESS,*)	getrlimit(RLIMIT_DATA,*)
getrlimit(RLIMIT_NOFILE,*)	getrlimit(RLIMIT_STACK,*)	getrusage(RUSAGE_CHILDREN,*)
getrusage(RUSAGE_SELF,*)	getsockname	getsockopt(*,SOL_SOCKET,SO_ERROR,*,*)
getsockopt(*,SOL_SOCKET, SO_PEERCRED,*,*)	getsockopt(*,SOL_SOCKET,SO_RCVBUF,*,*)	gettid
getuid32	ioctl(<ASHMEM-DEV>,*,*)	ioctl(*,FIONBIO,*)
ioctl(<LOG-DEV>,*,*)	ioctl(*,SIOCGIFADDR,*)	ioctl(*,SIOCGIFBRDADDR,*)
ioctl(*,SIOCGIFCONF,*)	ioctl(*,SIOCGIFFLAGS,*)	ioctl(*,SIOCGIFHWADDR,*)
ioctl(*,SIOCGIFINDEX,*)	ioctl(*,SIOCGIFNETMASK,*)	ioctl(<STD IN/OUT/ERR>, SNDCTL_TMR _TIMEBASE or TCGETS, *)
ioctl(*,SNDCTL_TMR_TIMEBASE or TCGETS,*)	ioctl(<URANDOM-DEV>, SNDCTL_TMR_TIMEBASE or TCGETS,*)	listen
lseek(*,*,SEEK_CUR)	lseek(*,*,SEEK_END)	lseek(*,*,SEEK_SET)
lstat64	madvise(*,*,MADV_DONTNEED)	madvise(*,*,MADV_NORMAL)
madvise(*,*,MADV_RANDOM)	mkdir(<SYS-PATH>,*)	mkdir(<USER-PATH>,*)
mmap2	mprotect	mremap(*,*,*,MREMAP_MAYMOVE)
munmap	nanosleep	open(<SYS-PATH>,*,*)
open(<SYS-PATH>,*)	open(<USER-PATH>,*,*)	open(<USER-PATH>,*)
pipe	poll	prctl(PR_GET_NAME,*,0,0,0)
prctl(PR_SET_NAME,*,*,*,*)	prctl(PR_SET_NAME,*,*,*,0)	prctl(PR_SET_NAME,*,0,0,0)
ptrace(PTRACE_TRACEME,*,0,0)	readlink(<USER-PATH>,*,*)	recvfrom
recvmsg	rename(<USER-PATH>,<USER-PATH>)	rmdir(<USER-PATH>)
rt_sigprocmask(SIG_BLOCK,*,*,*)	rt_sigprocmask(SIG_SETMASK,*,*,*)	rt_sigreturn(*)
rt_sigtimedwait([QUITUSR1], NULL, NULL, 8)	sched_getparam	sched_getscheduler
sched_yield	select	sendmsg
sendto	setitimer(ITIMER_REAL,*,*)	setpriority(PRIO_PROCESS,*,<POS INT>)
setpriority(PRIO_PROCESS,*,0)	setrlimit(RLIMIT_NOFILE,*)	setsockopt(*,SOL_IP,*,*,*)
setsockopt(*,SOL_SOCKET,*,*,*)	set_tls(*,*,*,*,*)	set_tls(*,*,0,*,*)
sigaction	sigprocmask(SIG_BLOCK,*,*)	sigprocmask(SIG_SETMASK,*,*)
sigprocmask(SIG_UNBLOCK,*,*)	sigreturn	sigsuspend([])
socket(PF_INET,SOCK_DGRAM, IPPROTO_ICMP)	socket(PF_INET,SOCK_DGRAM, IPPROTO_IP)	socket(PF_INET,SOCK_DGRAM, IPPROTO_UDP)
socket(PF_INET,SOCK_STREAM, IPPROTO_IP)	socket(PF_INET,SOCK_STREAM, IPPROTO_TCP)	socket(PF_NETLINK,SOCK_RAW, 0)
socket(PF_UNIX, SOCK_STREAM, 0)	stat64	statfs64(<SYS-PATH>,*)
statfs64(<USER-PATH>,*)	symlink(<USER-PATH>,<USER-PATH>)	tgkill(*,*,SIGTRAP)
umask	uname	unlink(<USER-PATH>)
utimes	vfork	wait4

TABLE XVI. This table shows the list of considered root exploits, on which syscall-level behavior they rely, and which exploits are successfully blocked by our policy.

Name / CVE	Description	Blocked
Exploid (CVE-2009-1185)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
GingerBreak (CVE-2011-1823)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
CVE-2013-2094	Uses `perf_event_open` system call	Yes
Vold/ASEC [34]	Creates symbolic link to a system directory	Yes
RATC (CVE-2010-EASY)	Relies on invoking many times the `fork` syscall	No
CVE-2013-6124	Creates symbolic links to system files	Yes
CVE-2011-1350	`ioctl` call used violates our rules	Yes
Zimperlinch	Relies on invoking many times the `fork` syscall	No
CVE-2011-1352	`ioctl` call used violates our rules	Yes
CVE-2011-1149	It relies on the `mprotect` syscall	No
CVE-2012-4220	`ioctl` call used violates our rules	Yes
CVE-2012-4221	`ioctl` call used violates our rules	Yes
CVE-2012-4222	`ioctl` call used violates our rules	Yes

The `fork` syscall is allowed by our policy as some benign applications do use it. However, note that this kind of exploit could be blocked by a security policy that would take into account the frequency of invocations of a given syscall: In fact, no benign application would ever invoke the `fork` syscall so frequently. We believe that considering this additional aspect of native code behavior is a very interesting direction for future work.

Although our example security policy does not block all exploits, we believe the adoption of native sandboxing to be useful. In fact, it does sensibly reduce the attack surface available to native code components, and it is able to successfully block a number of root exploits. Similarly, we believe that useful policies can be generated by our dynamic analysis system that will be able to block future exploits.

B. Java Methods

Even with the system call restrictions, native code can still perform dangerous actions by invoking Java methods. This can be accomplished by using certain JNI functions, as discussed in Section III-B. Static analysis of the Java component of apps cannot identify these calls, therefore, the possibility of apps calling methods in Java libraries poses a threat to the system and can be abused by malicious apps.

We performed the same process presented in Section V-A to automatically generate policies that restrict the use of methods in Java libraries. Table XVII presents these policies, using different values as the minimum percentage of allowed apps that reached native code during dynamic analysis. We used 97%, 98%, and 99% as the values for the minimum. The methods authorized for each threshold include the ones associated with lower thresholds.

Using the list of apps associated with a minimum of allowed apps of 99% (the most permissive of our thresholds), we would block 1,414 apps (0.12%). The method `java.lang.ClassLoader.loadClass`, which is allowed when using 99% as a threshold, causes the invocation of the static initialization block (`<clinit>`) of a class. Therefore, it could be used to execute the static initialization block of classes in Java libraries. However, as far as we know, these blocks do not contain important operations that need to be contained.

TABLE XVII. List of allowed methods (Java methods called from native code) automatically generated for allowing a minimum of 97%, 98% and 99% of apps that reached native code.

Allowed apps (%)	Method
97	java.lang.Integer.doubleValue
97	android.content.ContextWrapper.getPackageName
97	java.lang.String.getBytes
98	java.lang.Double.doubleValue
98	android.content.ContextWrapper.getClassLoader
98	android.content.ContextWrapper.getFilesDir
98	java.io.File.getPath
98	android.content.ContextWrapper.getExternalFilesDir
98	android.view.WindowManagerImpl.getDefaultDisplay
98	java.lang.String.toLowerCase
98	android.app.Activity.getWindowManager
98	java.util.ArrayList.add
98	android.view.Display.getMetrics
98	android.app.Activity.getWindow
98	android.view.View.getWindowVisibleDisplayFrame
98	java.util.Calendar.getInstance
98	android.view.View.getDrawingRect
99	java.util.Calendar.get
99	android.os.Bundle.getByteArray
99	android.content.ContextWrapper.getPackageManager
99	android.content.res.AssetManager$AssetInputStream.read
99	java.lang.Long.doubleValue
99	java.lang.ClassLoader.loadClass
99	android.app.ApplicationPackageManager.getPackageInfo
99	android.content.res.AssetManager$AssetInputStream.close
99	java.lang.Float.doubleValue
99	java.lang.Class.getClassLoader

VI. Impact of Security Policies

Considering both our policies—Java methods and system calls—, and the 99% threshold, we would block 0.23% (2,730) of all the apps in our dataset. To understand what the impact of implementing (and enforcing with the strictest enforcement mechanism) these policies would be on users, we analyzed the popularity (lower number of installations) of the apps whose behavior seen during the dynamic analysis would be blocked. Figure 2 presents the cumulative distribution of the popularity of the apps that would be blocked. As the figure shows, among the applications for which our policy would block at least one behavior that has been executed at runtime, 1.87% (51) of them have more than 1 million installations.

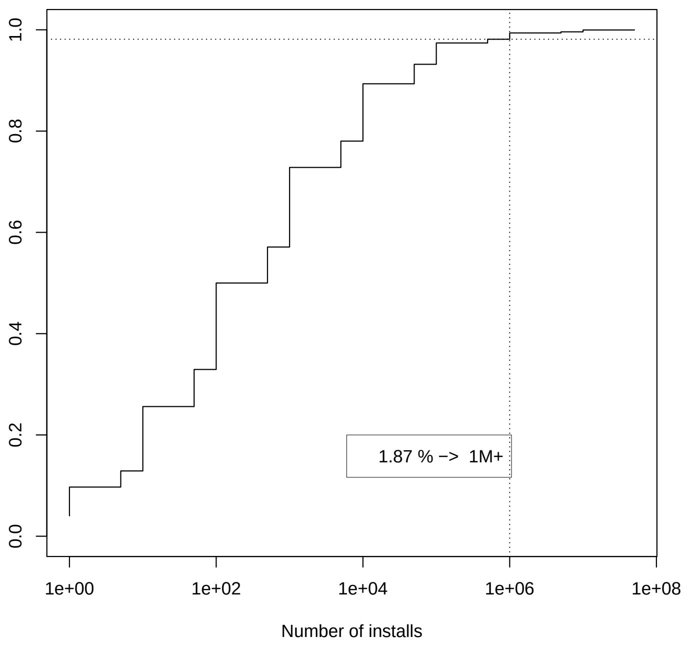

Fig. 2. Popularity of apps that would be blocked by enforcing our policy. X-axis is in logarithmic scale, and the Y-axis is the percentage of apps that would be blocked.

Because manual analysis is very time-consuming, we did not perform it on all blocked apps. However, we did a general investigation of the blocked apps and manually analyzed the ones that showed traces of suspicious behavior. We identified three types of suspicious activities among these apps, and we discuss them here.

Ptrace. Overall, 280 apps used `ptrace`. 276 of these only call `ptrace` to trace itself without checking the result. We assume that the developers do this as a defensive measure to prevent the analysis of the app, because an app cannot be traced by another process if there is already a process tracing it. Therefore, for these 276 apps we believe that the app's functionality would remain intact with our policy. Four apps, on the other hand, create a child process, which try to attach `ptrace` to the parent, checking the result of the call and changing behavior if the call failed.

Modifying Java code. We identified 7 apps that modify the Java section of the app from native code. All these apps perform this action from the library `libAPKProtect.so` [3]. This library is provided by an obfuscation service, thus making it harder for reverse engineering tools to decompile the app. This functionality can also be used by malicious apps and illustrates the importance of isolating native code.

Fork and `inotify`. We identified 57 apps that create a child process in native code and use `inotify` to monitor the apps' directory, in order to identify when they are uninstalled. In fact, the spawned child process uses `inotify` to detect when the app is uninstalled and, when this happens, it opens a survey in the browser. This behavior is not a malicious action; however, executing code after being uninstalled is suspicious, as the user does not expect the app to be running after being uninstalled.

VII. Dynamic Coverage

Dynamic analysis is inherently incomplete, and in this section we attempt to measure the code coverage of the dynamic analysis that we used, using function coverage of the Java code and function coverage of the native code. Both code coverage methods have large overhead, so we were only able to analyze a subset of the apps.

A. Java Method Code Coverage

To measure the code coverage based on the Java methods executed, we instrumented the DVM. The instrumented code records the execution of every method of the app under analysis. Since this instrumentation introduces more overhead and slows the emulator, we did the experiment with 25,000 apps randomly selected and used a kernel driver, instead of `strace`, to record the system calls executed. The code coverage obtained was 8.31%

B. Native Code Coverage

While code coverage of the Java methods allows us to gain insight into the high level code coverage of our dynamic analysis system, it does not shed light on the core issue we are interested in: *how much of an app's native code is the dynamic analysis able to execute*? To answer this question, we modified both the Android emulator and the Android framework to support measuring function coverage of the native code.

One technical challenge here is that the native code coverage must understand not only which native libraries are loaded by an app, but also which part of the native library is actually executed. Thus we need to: (1) trace the executed native functions and (2) statically determine the total number of native functions. This will allow us to calculate the function coverage of the native code.

To the best of our knowledge, there is no previously released tool to trace the execution of the native code of an app. Android Open Source Project implements a tracing mechanism since version 4.4. This tracing mechanism is implemented using a kernel device called qemutrace that is part of the goldfish kernel. The kernel send information to assist the emulator to trace correctly the execution, e.g., the PID of the running process each time there is a context switch, a message that notifies that a fork or an execve is executed, etc. The whole tracing system significantly slows down the performance of the emulator. However, this tracing system is too general: we are interested only in the execution of the native code of a specific app. We need to trace only functions of loaded libraries of the app under analysis.

For this reason, we created two ways to limit the tracing to the interesting part only. First, we only want to trace processes with a specific UID because each app in Android is executed with it own UID. In addition, we are interested only in portions of the executable memory where the native libraries have been loaded.

To inform the emulator about the UID of the currently executing process we leverage the existing qemutrace device. We added the UID into the message sent for each context switch. To send the information about the map of the memory to the emulator we cannot use the qemutrace device, since it can only pass 32 bit integers as messages. Moreover, we also need a mechanism to extract the libraries from the emulated system. To solve both problems we instrumented the Android framework. We found that the function `java.lang.Runtime.doLoad` is able to intercept all the loading operations. Our hook inside the `doLoad` function blocks the loading (and the app) while syncing all the gathered data to the external emulator. The mapping of the memory and the PID are read from `/proc/self/`. The path of the loaded library is one of the parameters of the `doLoad` function. Hence, when `doLoad` returns, the emulator knows the address space reserved for the new library, and the content of the native library.

After the dynamic execution, we compute the code coverage using all the data gathered during the execution. We use IDA Pro to find all functions boundaries of libraries. Then, we use the map of the memory to translate the virtual addresses traced by the emulator. Next, we flag all the functions whose boundaries include at least one address of the trace. The code coverage is then calculated.

Our tracing system slows down the execution of the apps by around 10 times. Therefore, we only ran it on a small subset of the apps, more specifically, we analyzed 177. The code coverage of most libraries is less that 1%. Some small libraries, on the other hand, were covered by 100%. Furthermore, the average coverage was 7%. More details about executed libraries and coverage can be seen in Figure 3.

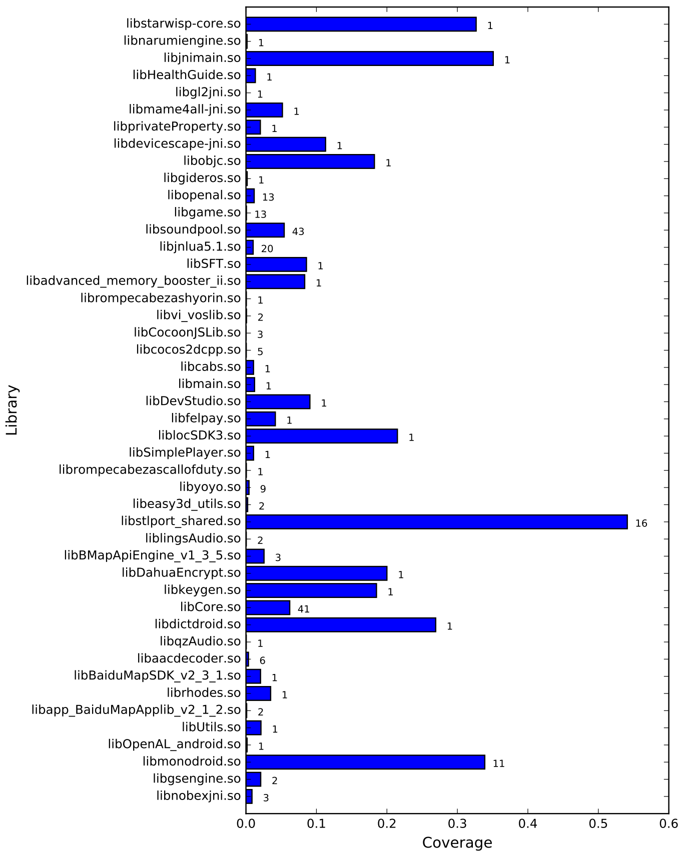

Fig. 3. Per library coverage of executed functions. Horizontal axis contains libraries name, vertical, instead contains the function coverage. For each bar we also show the number of libraries that has been found in all executed applications

VIII. Threats to Validity

Our study is affected by a few limitations, which we discuss in this section. An intrinsic limitation of the automatically-generated security policies is that we base their automatic generation on data and insights obtained by means of dynamic analysis, which is well-known to be incomplete and affected by code coverage issues. In fact, dynamic analysis does not ensure that all native code is exercised in the apps that actually use it, and for those apps that used native code, dynamic analysis may not have exercised all code paths in the native code. Consequently, the policies that our tool generated might not be complete, they might block more applications when adopted at large-scale, and the performance overhead of isolating native code could be higher. However, using a more-sophisticated instrumentation tool could possibly improve the amount of native code behavior that our system observes, or deploying the automatically generated policies in a native sandbox with reporting mode would help to observe the behaviors that the policies would block.

Nonetheless, we believe this work to be a significant first step in a very important direction. In fact, to the best of our knowledge, this work is the first, largest, and most comprehensive study on how real-world applications use native code. Our results demonstrate that it is infeasible to adopt a completely restrictive sandboxing policy. In addition, we propose a system to automatically generate a native code sandboxing policy following a data-driven approach. This system could be used by large organizations that are interested in automatically generating a native code sandboxing policy. Furthermore, the completeness issues could possibly be addressed by increasing the fidelity of the dynamic analysis, either through more sophisticated analysis techniques or increased resources, or by obtaining the actual behavior of native code in the wild, by instrumenting real-world Android devices.

Another limitation is that our approach restricts access to permissions from native code, but it still allows the native code to invoke (some) Java methods. This aspect would make, in principle, Java-only analysis more precise, but still not completely sound, as a malicious application could introduce *hidden* execution paths by invoking a native method, which, in turn, could invoke a Java method. However, we note that our automatically-generated policy only allows native code to invoke a very narrow subset of Java methods defined in the Android framework (Table XVII), through which it is virtually impossible to perform any security-sensitive operation. Thus, our policy, although not perfect, would drastically reduce the possibility of introducing malicious behaviors.

Lastly, we consider all the apps we obtained from Google Play as benign, but we cannot be completely certain that there are no malicious apps among them. The effects of having malicious apps in our dataset vary depending on how the malware works. In the worst case it could cause our policies to allow some malicious actions.

IX. Related Work

In this section we relate our work to the vast amount of research published in the field of Android security.

Large Measurement Studies. Several works have analyzed large datasets of Android apps, but with goals that differ from ours. Viennot et al. [37] did a large measurement study on 1,100,000 applications crawled from the Google Play app store. In particular, they collected meta-data and statistics taken from the Google Play store itself. As part of their study, they measured the frequency with which Android applications make use of native code components. Another important measurement study has been performed by Lindorfer et al. [27]. In their work, they analyzed 1,000,000 apps, of which 40% are malware. To perform the analysis, the authors used Andrubis, a publicly-available analysis system for Android apps that combines static and dynamic analysis. When focusing on native code, our work significantly extends their study.

Application Analysis Systems. Several systems have been proposed to perform behavioral analysis of Android applications based on dynamic analysis [13], [14], [30], [31], [36], [41]. Moreover, several other works have been proposed to identify malicious Android apps [4], [9], [23]. Our analysis complements all these research efforts by performing a large scale study, based on dynamic analysis, specifically focused on native code usage.

Protection Systems. Fedler et al. [15] proposes a protection system from root exploits by preventing apps from giving execution permission for custom executable files and by introducing a permission related to the use of the `System` class. PREC [24] is a framework intended to protect Android systems from root exploits. PREC uses two steps, learning and enforcement. During the learning phase, the analysis generates a model of the normal behavior for a given app. Then, during the enforcement phase, the system makes sure that the app does not deviate from the *normal* behavior. Our work has the advantage that the generated policies can be applied to all apps, whereas PREC generates per-app models. Hence, our results are more general. Moreover, our analysis also monitors, in addition to system calls, JNI function calls, Binder transactions and calls from Java to native methods.

Native Code Isolation. Another way to protect the system is by isolating native code. The challenge of isolating native code components used by managed languages has been extensively studied. For instance, Klinkoff et al. [26] focus on the isolation of *.NET* applications, whereas Robusta [33] focuses on the isolation of native code used by Java applications. Recently, NativeGuard [35] proposed a similar mechanism to isolate native code in the context of Android. Our work is complementary to these sandboxing mechanisms and fills the knowledge gap necessary to define security policies on the execution of native code in Android that are both usable in real-world applications and effective in blocking malicious behavior of native components.

X. Conclusion

While allowing developers to mix Java code and native code enables developers to fully harness the computing power of mobile devices, we believe that, in the current state, this feature does more harm than good and that native code sandboxing is the correct approach to properly limit its potentially malicious side-effects. However, a native code sandboxing mechanism without a proper policy will never be feasible. We hope that, in addition to shedding light on the previously unknown native code usage of Android apps, this paper demonstrates an approach to automatically generate an effective and practical native code sandboxing policy.

Acknowledgment

This material is based upon work supported by CAPES Foundation under Award No. BEX 12269/13-1, by NSF under Award No. CNS-1408632, by DHS under Award No. 2009-ST-061-CI0001, and by Secure Business Austria. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of CAPES Foundation, NSF, DHS, or Secure Business Austria.

This material is also based on research sponsored by DARPA under agreement number FA8750-12-2-0101. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Full version of Tables 5, 6, 7, 8, and 11." [Online] Available: https://github.com/ucsb-seclab/android_going_native.

[2] AppBrain, "Number of Available Android Applications," [Online] Available: http://www.appbrain.com/stats/number-of-android-apps.

[3] A. Apvrille and R. Nigam, "Obfuscation in Android Malware, and How to Fight Back," in *Virus Bulletin*, 2014.

[4] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2014.

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the Android Permission Specification," in *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*, 2012.

[7] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities Within Android Applications," in *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software (MALWARE)*, 2011.

[8] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-Privilege Compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[9] 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)*, 2011.

[10] V. Chebyshev and R. Unuchek, "Mobile Malware Evolution: 2013," [Online] Available: http://securelist.com/analysis/kaspersky-security-bulletin/58335/mobile-malware-evolution-2013/, Feb. 2014.

[11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys)*, 2011.

[12] A. Desnos, "Androguard: Reverse Engineering, Malware and Goodware Analysis of Android Applications... and More (Ninja!)," [Online] Available: https://code.google.com/p/androguard/.

[13] Droidbox, "Android Application Sandbox," [Online] Available: https://code.google.com/p/droidbox/.

[14] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: an Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[15] R. Fedler, M. Kulicke, and J. Schütte, "Native Code Execution Control for Attack Mitigation on Android," in *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices (SPSM)*, 2013.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*, 2011.

[17] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated Security Certification of Android Applications," *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[18] C. Gibler, J. Crussel, J. Erickson, and H. Chen, "AndroidLeaks: Detecting Privacy Leaks in Android Applications," Tech. rep., UC Davis, Tech. Rep., 2011.

[19] Google, "Android NDK," [Online] Available: https://developer.android.com/tools/sdk/ndk/index.html.

[20] ——, "UI/Application Exerciser Monkey — Android Developers," [Online] Available: http://developer.android.com/tools/help/monkey.html.

[21] R. Gordon, *Essential JNI: Java Native Interface*. Prentice-Hall, Inc., 1998.

[22] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[23] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and Accurate Zero-Day Android Malware Detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys)*, 2012.

[24] T.-H. Ho, D. Dean, X. Gu, and W. Enck, "PREC: Practical Root Exploit Containment for Android Devices," in *Proceedings of the 4th ACM conference on Data and application security and privacy (CODASPY)*, 2014.

[25] IDC Corporate, "IDC: Smartphone OS Market Share 2014, 2013, 2012, and 2011," [Online] Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[26] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna, "Extending .NET Security to Unmanaged Code," *International Journal of Information Security*, 2007.

[27] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[28] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[29] C. Mann and A. Starostin, "A Framework for Static Detection of Privacy Leaks in Android Applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, 2012.

[30] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile Protection for Smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 347–356.

[31] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On Tracking Information Flows through JNI in Android Applications," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[32] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *Proceedings of the third ACM conference on Data and application security and privacy (CODASPY)*, 2013.

[33] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the Native Beast of the JVM," in *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.

[34] A. D. . Space, "Local Root Vulnerability in Android 4.4.2," [Online] Available: http://blog.cassidiancybersecurity.com/post/2014/06/Android-4.4.3,-or-fixing-an-old-local-root.

[35] M. Sun and G. Tan, "NativeGuard: Protecting Android Applications from Third-Party Native Libraries," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks (WiSec)*, 2014.

[36] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[37] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proceedings of the 2014 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

[38] C. Warren, "Google Play Hits 1 Million Apps," [Online] Available: http://mashable.com/2013/07/24/google-play-1-million/, Jul. 2013.

[39] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[40] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer, "ANDRUBIS: Android Malware Under The Magnifying Glass," Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001, 2014.

[41] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[42] Z. Yang and M. Yang, "Leakminer: Detect Information Leakage on Android with Static Taint Analysis," in *Proceedings of the 2012 Third World Congress on Software Engineering (WCSE)*, 2012.

[43] Z. Zhao and F. C. C. Osono, ""TrustDroid$^{\text{TM}}$": Preventing the Use of SmartPhones for Information Leaking in Corporate Networks Through the Use of Static Analysis Taint Tracking," in *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*, 2012.

[44] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[45] Y. Zhou and X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.