NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running stock Android

Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna
University of California, Santa Barbara
{antoniob,yanick,chris,vigna}@cs.ucsb.edu

ABSTRACT

Malware poses a serious threat to the Android ecosystem. Moreover, even benign applications can sometimes constitute security and privacy risks to their users, as they might contain vulnerabilities, or they might perform unwanted actions. Previous research has shown that the current Android security model is not sufficient to protect against these threats, and several solutions have been proposed to enable the specification and enforcing of finer-grained security policies. Unfortunately, many existing solutions suffer from several limitations: they require modifications to the Android framework, root access to the device, to create a modified version of an existing app that cannot be installed without enabling unsafe options, or they cannot completely sandbox native code components.

In this work, we propose a novel approach that aims to sandbox arbitrary Android applications. Our solution, called NJAS, works by executing an Android application within the context of another one, and it achieves sandboxing by means of system call interposition. In this paper, we show that our solution overcomes major limitations that affect existing solutions. In fact, it does not require any modification to the framework, does not require root access to the device, and does not require the user to enable unsafe options. Moreover, the core sandboxing mechanism cannot be evaded by using native code components.

Keywords

Mobile Security; Android; Code Sandboxing; System Call Interposition

1. INTRODUCTION

Smartphones have thoroughly established themselves as the dominant computing platform for many users, with 1.5 million devices activated daily, and over a billion active devices in use, according to a recent Google report [1]. Smartphones are used in an ever-growing variety of use-cases including highly-sensitive tasks, from accessing private information to on-line banking. In fact, it is not a coincidence that the Google Play market currently offers more than 1 million applications [3] ("apps," from now on). Unfortunately, the huge popularity of this platform makes it more attractive for malware writers. Moreover, even benign apps can sometimes pose a threat to the user. In fact, previous research has shown that even reputable apps sometimes violate the users' privacy, for example by accessing and leaking the user's address book, even when not strictly required [12]. Research has also shown that benign apps often contain vulnerable components that expose the users to a variety of threats: common examples are component hijacking vulnerabilities [20], permission leaking [17], and remote code execution vulnerabilities [21].

To minimize the impact of these threats, Android features a complex permission system that can selectively limit the capabilities of Android apps. One of the main problems is that it follows an *"all-or-nothing"* paradigm: a user can either grant to an app all the permissions it asks for, or abort its installation. In other words, the user cannot selectively enable or disable specific permissions. That said, Google has experimented with adding this feature to Android. A hidden, developmental feature called "AppOps", included with Android 4.3 and 4.4, allowed this to be done on a limited basis. However, this feature was removed in subsequent Android versions. As the EFF pointed out [11], the absence of this feature constitutes a problem for the privacy of Android users. Another limitation of the current permission system is that it is too *coarse-grained*. For example, if an app has the INTERNET permission, it will have access to the entire web: some apps may require such broad capabilities (e.g., a web browser), but it is likely that the vast majority of them would need to contact only a small set of domain names, as studied in [14].

As a reaction to the aforementioned problems, the security community started to propose different approaches to provide a fine-grained permission system that is also more configurable and flexible than the current one. Unfortunately, many of the proposed approaches are either affected by several important limitations that impact their usability and large-scale adoption, or they cannot completely sandbox app's components written in native code.

An important thrust of research proposes to modify the Android framework itself. For example, MOSES [23] and AirBag [29] implement a virtualization-based approach, while MockDroid [7] develops a set of patches to the Android framework. FireDroid [24] implements an approach based on system call ("syscall," in short) interposition, showing

SPSM'15, October 12 2015, Denver, CO, USA
 ISBN 978-1-4503-3819-6/15/10 $15.00
DOI: http://dx.doi.org/10.1145/2808117.2808122

how this technique can be effective in sandboxing arbitrary Android applications. Unfortunately, all these approaches suffer from several limitations: they either require intrusive modifications to the Android framework, or administrative ("root") privileges. While the first limitation makes the wide adoption of such systems difficult, the requirement of having root access is not always reasonable. In fact, not all the users are capable of (or willing to) "root" their device, as the process of "rooting" the device may void its warranty, and it affects its overall security.

Other works propose to instrument and sandbox the execution of a given Android app by statically rewriting its bytecode [6, 9, 10, 30]. While these works have the advantage of providing as output a self-contained Android app, they have several limitations. First, such approaches currently do not offer complete protection against native code, thus offering to the adversary a simple way to evade them. In fact, even those approaches that can sandbox native code components (e.g., [30]) rely on library function call interposition and hence can be evaded, for example, by invoking syscalls through inline assembly code. Second, if the modified Android app is not published on the official Google Play market, the user would need to enable the "allow apps from unknown sources" option to install and execute it. This option is not present in all devices, and enabling it decreases the security of the entire system, as it allows the installation and execution of untrusted apps. As an alternative, one could modify a given app and upload it to the Google Play market: while this is technically possible, this would introduce legal issues, as the rewritten app would contain the original app's content (in a slightly modified form).

Independently to our work, a recent paper [5] proposes a sandboxing mechanism that does not require modifications of the Android system. However, this system requires the usage of an app granting all the possible Android permissions and the reimplementation of many of the security checks usually implemented by the Android operating system. While this approach is practical and, in principle, secure, it introduces a single point of failure (i.e., the application with all the permissions), it increases the attack surface, and it requires the porting of the security checks for each new Android version, thus opening another venue for the introduction of security vulnerabilities. For this reason, we believe that this approach may potentially weaken the Android security properties, as we will better explain in Section 7.

In this paper, we introduce NJAS (Not Just Another Sandbox), an approach that aims to sandbox arbitrary Android apps. At its core, NJAS sandboxes a given application by means of syscall interposition (using the `ptrace` mechanism), and it works by loading and executing the code of the original application within the context of another, monitoring, application. As we discuss throughout the paper, this proved to be technically challenging, because Android applications are written with the assumption that the code is executed in a very specific execution context.

NJAS comes with important advantages that overcome several limitations of many existing systems. In fact, our approach does not require any system modification, it transparently works across different Android versions, it does not require root access on the device, and it does not force the user to enable the "allow apps from unknown sources" option. Moreover, as our system performs sandboxing by using a kernel-based feature (i.e., `ptrace`), it offers complete protection even when native code is used. Finally, the usage of NJAS does not require any technical knowledge, and, for this reason, we believe it could benefit a wide range of users.

Our approach is flexible enough to enforce a variety of fine-grained security policies. In particular, our current prototype supports the enforcing of policies that monitor an application's network access, file system access, SMS-related capabilities, and access to user's private information, such as the contact list. To test the applicability of our approach, we used NJAS to enforce different security policies to several real-world applications downloaded from the Google Play market, with an acceptable performance overhead.

In summary, this paper makes the following contributions:

- We propose an approach to sandbox generic Android applications that does not suffer from usability limitations that affect many existing solutions, while, at the same time, it can completely sandbox an application even when native code components are used.
- We address the general problem of executing an application within the context of another, non-privileged, application, on the Android platform. We describe the many technical challenges that needed to be addressed, and we systematically explore the advantages and limitations of the proposed solution.
- We implement this approach in a prototype tool called NJAS. As we show in the evaluation section, NJAS can effectively enforce configurable security policies related to different categories of sensitive operations. We tested our system by instrumenting several real-world applications downloaded from the Google Play market, and we show that the performance overhead is aligned to the one of similar approaches.

2. THE ANDROID ECOSYSTEM

In this section, we provide an overview of the Android ecosystem and an in-depth description of the components on top of which our approach is based.

2.1 Android Application

An Android app is shipped in a single file, called an Android application package (APK, in short). An APK is a zip archive that includes the apps's codebase (stored in a file called `classes.dex`) and app's configuration files (usually called *Resources*), which store configuration data like string values, icons, and images. Moreover, each APK needs to include the *Android manifest* configuration file, which contains critical information, such as the list of app's components and the required permissions. Optionally, an APK can contain shared object modules (`*.so` files) that include native code components, which can be directly executed by the device's processor.

The main components of an app are usually written in Java. The Java source code is then compiled to Dalvik bytecode and assembled in the previously-mentioned `classes.dex` file. This bytecode is executed by the Dalvik virtual machine. Additionally, developers can choose to implement the more CPU-intensive parts of the application in *native* libraries, written in a low-level language, such as C or C++.

Every app is uniquely identified by its *package name* (specified in the manifest file) and signed with a private key from its developers. In Android, an app is composed of different components. Four different types of components exist: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. An *Activity* defines a graphical user interface and how the user can interact with it. Differently, a *Service* is a component that runs in background to perform long-running operations. Finally, a *Broadcast Receiver* is a component that responds to specific system-wide messages, while a *Content Provider* manages app data shared with other components (within the same app or in external ones).

2.2 Security Model

Android is an operating system based on Linux. Differently from standard Linux distributions, in Android each application runs as a different Linux user: a unique user id (UID) is created and assigned to each app at installation time. This ensures that interactions among different apps can only happen through well-defined IPC mechanisms.

Moreover, the Android framework is characterized by a complex permission system that monitors the execution of every security-sensitive functionality. Each app needs to declare the set of required permissions in its manifest file. The user will then need to review and accept these permissions at installation time.

Permissions are enforced by means of two different mechanisms. The first one relies on the Linux group abstraction: for example, if the INTERNET permission is required by a given app, the Linux user associated with such app will be added to the *aid_inet* group. Only applications run by users in this group can connect to the network.

The second mechanism relies on run-time security checks. These checks are not implemented in a centralized location, but they are spread among several different components, depending on the particular permission. As an explanatory example, let us consider an app that attempts to send a text message. When the app invokes the `sendTextMessage` Android API to send a text message, a request is sent to the `SmsManager` service, the Android built-in system service that is responsible for actually sending text messages. At this point, `SmsManager` will perform a query to the Android's `PackageManager` service to check whether the app that initiated the request has the associated permission. These kinds of checks are performed at run-time every time an app requests to perform a sensitive operation. It is worth noting that the `SmsManager` service, as well as every other system service, runs as a separate process, owned by the privileged *system* user. The communication between processes is performed through one of the well-defined IPC mechanisms, the most common being the Binder kernel module (we will provide a detailed description of this IPC mechanism in Section 4.3.1).

2.3 Life Cycle of an Android Application

In this section we provide an overview of the different phases of an Android application's life cycle.

2.3.1 Installation

When the user requires the installation of an app, its APK is downloaded to the device, and the list of required permissions is extracted. Such list is then displayed to the user, who has now two options: she can either accept all the permissions or abort the installation of the app. Such *"all-or-nothing"* policy constitutes one of the main limitations of the current Android's permission system, and it is one of the main motivations behind the development of approaches that give the user the possibility to use more flexible security policies.

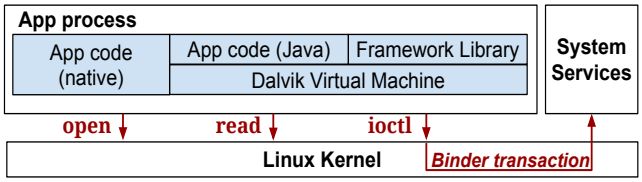

Figure 1: Interaction among an app, the Linux kernel, and remote system services. The arrows exemplify syscalls called by the app's process.

If the user accepts, the app is installed and a new Linux user is created and associated to it. From now on, the Android framework becomes *aware* of the new app: in fact, the many Android system components, as well as other apps, will now be able to access all the information about the newly installed app by querying the `PackageManager` service.

2.3.2 Launching

The most common way for a user to start a specific Android application is to click on its associated icon. At this point, the application that is responsible to handle events related to the menu, the *Launcher* app, asks the system to show the app's main activity. If this app is not already running, the `zygote` process starts to execute a series of operations that will eventually lead to the app's execution.

The `zygote` process is a privileged process that is the analogous for Android of the Linux's `init` process: all Android applications' processes are generated starting from a `fork` operation by `zygote`. After the `fork` operation, the newly generated process first retrieves the UID associated to the *target* application, and it then drops its privileges accordingly. At this point, this process will receive (from the `ActivityManager` service) the information related to which app needs to be executed.

After retrieving the needed information, the newly generated process loads the code of the *target* application (by using a `ClassLoader` object[1]), and it then creates a `Context` object which contains information about the context the application should be executed in.

2.3.3 Run-time

At run-time, an Android app is constituted by the code written by its developers (stored in its APK file) and the Android framework libraries, which expose a variety of useful APIs (Figure 1). Independently of where the code is implemented, it runs within the scope of the same process. The code of each application can be correctly run only within a very specific execution context. In fact, such code is designed to heavily interact with the surrounding Android framework, mainly through the Binder IPC mechanism. Moreover, an application interacts with the surrounding framework even when it needs to perform intra-

[1]A `ClassLoader` is a special object that allows loading additional code from external resources.

application operations. For example, an app can switch from an activity to another, by invoking the `startActivity` API, which, in turn, will communicate with the `ActivityManager` service through the Binder IPC mechanism.

3. NJAS – OVERVIEW

In this section, we first discuss a high-level overview of our approach, and how it provides a usable and flexible solution for sandboxing Android applications. Then, we describe our design goals and the several technical challenges we needed to address. We postpone the description of the low-level technical details to Section 4.

3.1 Approach

From a high-level point of view, our approach aims to sandbox a generic Android application by running it in an instrumented environment that monitors its execution by means of syscall interposition. At its core, our system relies on the ability to load an application within the context of another one, i.e., our sandbox. The main technical challenge is related to the creation of a *compatible* execution environment. In fact, if not properly handled, the code of the sandboxed application cannot work correctly, as it was designed to run in a different context.

Starting from the app we want to sandbox (which we will refer to as `orig`), the first step is the generation of a *compatible* `stub` application. The `stub` app is generated in a completely automated fashion and its only role is to act as a "container" application. Such application only contains few pieces of information taken from the manifest of the original app (such as the list of permissions, the activity names, and similar others), and it does not contain any code or resources from the original app. In our current implementation, the size of a `stub` is about 315 KB, sensibly lower than the size of normal Android applications (for instance, the current size of the Gmail application is about 6 MB)

When opened, `stub` loads `orig`'s code, monitoring its execution through syscall interposition. Our system requires the installation of a single *compatible* `stub` for each `orig` application. Therefore, the same `stub` application can be used to enforce several different user-defined security policies.

It is worth noting that the process of generating an application starting from another one has few things in common with the process of repackaging. However, these two approaches are profoundly different. In fact, while our solution only aims to generate a *compatible* stub, the approaches based on repackaging techniques generate a slightly modified version of the original app, sharing the same codebase and resource files. For this reason, it is not possible to publish such app on the official Google Play market, as this process would lead to legal issues. Alternatively, a user of such systems would need to retrieve and install the repackaged app from a third-party source: this forces the user to enable the "allow apps from unknown sources" option that, in turn, lessens the security of the entire device.

On the contrary, publishing `stub` applications on the market does not pose any problem. In fact, these applications do not contain code or resources taken from the original ones. To verify this possibility, we submitted to the Google Play market the `stub` apps associated to several real-world popular applications (the same apps we used for our evaluation, described in Section 5). As expected, all these apps were promptly approved on the official market.

3.2 Usability Discussion

As one of our main goals is to develop a system that is intuitive and easy to use, we envision the usage of an application, which we will refer to as `manager`, that guides the user through the necessary steps to use our system. Specifically, `manager` takes care of downloading, installing, configuring, and opening the `stub` associated to a given `orig`. We now describe all the different steps.

First, `manager` will ask the user to install `orig` on the device (if not already installed). This can be easily done by using the default market application of the device (typically, the Google Play market), or any additional third-party market app. After `orig` has been successfully installed, `manager` will prompt the user to download and install from the market the `stub` associated to `orig` (if not already present on the device). Note that `manager` cannot transparently install these apps because Android requires an explicit consent from the user before every app's installation. The `manager` app can identify the `stub` associated to `orig` in a number of ways. As an example, `manager` could identify the correct `stub` by applying simple conventions based on the package name. Upon `stub`'s installation, `manager` will give the user the possibility to specify a fine-grained security policy that will be enforced (refer to Section 4.4 for more details). As a final step, the user can then execute the sandboxed version of `orig` by clicking on its associated entry.

If `stub` is not already present on the market, `manager` would contact a dedicated server that will take care of automatically generating a *compatible* `stub` for the selected app and publishing it on the market. Note that these steps are executed in an on-demand fashion, only for apps users want to use with our system. To ease its usability, `manager` can optionally appear as a complete replacement of the standard Android "home" screen (as many apps that are currently present on the Google Play market do). This application would show to the user a list of all the installed apps (as a traditional "home" app would do), and the icon of each application for which a `stub` is available would appear with a *lock* symbol superimposed to it.

Finally, it is worth mentioning that our system addresses one of the usability limitations of existing approaches based on bytecode rewriting techniques. In particular, it is known that these techniques break the application's signature, and this, in turn, causes problems to the automatic update mechanism available on the Android platform. In fact, when a new version of a given app is available on the market, the app is automatically updated only if the signature used to sign the new version matches the signature of the locally installed version. For this reason, the users of such systems need to manually regenerate and reinstall newer versions of the repackaged apps.

On the contrary, NJAS does not break such automatic update mechanism. In fact, as an unmodified version of `orig` is installed on the device, the system will be able to automatically update it when a newer version is published on the market. In addition, a corresponding new *compatible* version of `stub` can be published as well (if required), and the system will be able to update it to the current version, once again automatically. Specifically, when signing `stub` apps, we maintain a one-to-one mapping between the signatures used to sign original applications and their corresponding `stub` apps. In other words, `stub` applications associated to two versions of `orig` signed with the same key, would be

signed with the same corresponding key. (Of course, such key will be different from the one used to sign `orig`, but this does not constitute any usability problems.)

3.3 Challenges and Design Choices

In the remainder of this section, we will discuss the main challenges we needed to address and our design choices.

How to sandbox an app. As we already mentioned, our approach is based on executing a given app within the context of another one. The first operation that `stub` performs is to fork itself, and to create a child process, which we call `monitor`. Using the `ptrace` mechanism, `monitor` is able to sandbox the execution of the original application (which runs within the context of `stub`) by means of syscall interposition. Note that this gives us control at the lowest-possible level and, since `ptrace` is a kernel-level feature, it is able to properly sandbox both the Java and the native components of an application.

Note that this is technically possible, even if `monitor` is an unprivileged process. In fact, since `stub` and `monitor` are in a parent-child relation, they share the same UID, and the `ptrace` interface can be therefore used. It is worth noting that while in Android this is possible, some specific Linux distributions (e.g., Ubuntu) disallow this possibility.

How to load code from the original app. As we previously explained in Section 3.1, the `stub` app contains only little high-level information about the `orig` app: that is, it does not contain `orig`'s codebase nor its resources. The `stub` app is able to load and execute `orig`'s codebase by directly reading `orig`'s APK file: this is possible as the APK files of apps are world-readable by default. However, note that this is possible only for free apps, as the APK files of paid apps are stored in a non-readable location. In Section 6 we will discuss this limitation, which NJAS shares with all systems based on bytecode rewriting.

The reader might be surprised that an app can actually load and execute code of another app, as it could seem that this technique could be used to bypass the Android security mechanism. However, this is not possible. In fact, the code loaded from `orig` will be executed within the context of `stub`, therefore, it will have the same privileges that `stub` has. For instance, let us assume that `orig` has the Internet permission: clearly, if `stub` itself does not have the same permission, it will not be able to connect to the network just by executing `orig`'s code. Interestingly, the possibility of loading code from external applications is a well-documented functionality provided by the Android framework and it is used by popular apps [21].

How to guarantee the enforcement of a permission set that is stricter than the original one. We chose to create the `stub` app in a way that it requires the same permission set of `orig`. In this way, it is guaranteed that the sandboxed code is subject to at least the same permissions than it would have been subject to when run by the `orig` app. This acts as a starting point that our sandbox mechanism uses to enforce fine-grained user-defined policies (discussed in more detail in Section 4.4).

This approach has the disadvantage of requiring the generation of a different, *compatible* `stub` for each `orig` application. However, it is difficult to implement a safe sandboxing mechanism without requiring such a step. Specifically, one alternative to our approach would be to first create a generic sandbox application that enjoys all the permissions, and then enforce a subset of them by means of syscall interposition. While this approach would be feasible from a theoretical point of view, it is a task that would require a significant engineering effort, and it would inevitably make the attack surface bigger. In fact, it would require the reimplementation of all the security checks performed by the Android framework, which, as we mentioned earlier, are not implemented in a centralized location. Alternatively, this generic sandbox could drop its Android permissions before executing the code of the app to be sandboxed. However, this is technically impossible because Android does not provide any mechanism to change an app's permission set at run-time.

How to properly run the original app within the context of another one. Running an Android app within the context of another one proved to be the main challenge we needed to address. In fact, the codebase of each application assumes to be executed within a very specific context that varies among different apps. Thus, failing to provide such context will inevitably lead the app to crash or behave incorrectly. We will now provide a high-level description about which part of the context our system patches. In particular, we discuss two representative examples.

First of all, the parameters of some syscalls need to be patched. For example, consider the case where `orig`'s code tries to access one of `orig`'s private directories: such code is executed by the `stub` app and, since `orig` and `stub` are actually different applications (whose processes are run by different users), the process would crash with a "permission denied" exception, if the syscall arguments would not be properly patched. The technical details on how this is implemented are provided in Section 4.2. Similarly, NJAS needs to patch the content of several Binder transactions, as we will detail in Section 4.3.

4. IMPLEMENTATION

In this section we document the underlying technical details of NJAS. We first give an overview of the setup phase, then we provide details related to how we reconstruct an execution context so that it is possible to properly execute the code loaded from `orig`. In particular, we discuss how it is needed to patch syscalls and Binder transactions. Next, we explain how we implemented the enforcing of user-defined security policies. Finally, we discuss the security properties of our system.

4.1 Setup

The first operation performed by `stub` consists in invoking the `prctl` syscall by passing the `SET_DUMPABLE` flag as argument. This guarantees that `stub`'s threads will be *debuggable* using the `ptrace` mechanism. Then, `stub` forks itself and creates a new process, which we call `monitor`. This process is responsible for monitoring the operations performed by `stub`. More precisely, `monitor` uses `ptrace` to *attach* to all `stub`'s threads and monitor them.

After the `monitor` process is created, `stub` loads `orig`'s code and starts its execution. To do so, `stub` creates a proper `ClassLoader` object. In Java, a `ClassLoader` is a special object used to load other classes. When it is created, a list of file paths is specified, which indicates where the classes' implementation should be searched. In our case, we obtain a reference to a `ClassLoader` object that loads

classes from orig's APK file by using the createPackageContext API (specifying the CONTEXT_INCLUDE_CODE and CONTEXT_IGNORE_SECURITY flags).

By using such object, stub can successfully load the main activity of orig and then spawn it by invoking the startActivity API. Note that, in general, when a class C needs to load another class D, the virtual machine will attempt to load D by using the same ClassLoader object used to load C. For this reason, all classes loaded by orig's main activity will be automatically loaded using the correct ClassLoader. In other words, orig's main activity will transparently load all the other classes implemented within orig's APK file.

4.2 File-related Syscall Patching

The Android OS assigns to each installed app a private folder, which is used, for example, to store the user's preferences or temporary files. The key problem lies in the fact that the code loaded from the orig app will try to access its own private folder (i.e., <orig_dir>). However, such folder is not accessible by the stub process, as its associated Linux user does not have permission to access orig's private folder. For this reason, the monitor process modifies the arguments of every file-related syscall, so that the file path is changed from <orig_dir> to a (private) directory that the stub app is able to access, <stub_dir>. For example, when the app tries to write to the <orig_dir>/database.db file, NJAS transparently changes the file path to <stub_dir>/database.db.

4.3 Patching Binder Communications

Android apps are designed to continuously communicate with the surrounding environment. Most of these communications are implemented by means of remote procedure call (RPC) invocations through the Binder IPC mechanism. As we mentioned in Section 3.3, some of these communications and interactions need to be properly patched, for two reasons: to guarantee that the orig app code behaves correctly, and to actually implement the sandbox mechanism.

To this aim, the monitor process modifies the content of Binder transactions by means of syscall interposition. The technical details of this process are discussed in the remaining of this section: we will first describe several low-level details on how the Binder mechanism works, and then will discuss which operations are performed by monitor.

4.3.1 Binder Internals

From a technical point of view, all Binder communications are implemented by means of the ioctl syscall. More specifically, each Binder invocation is implemented by calling the ioctl syscall on the /dev/binder device. Different "commands" on the Binder are encoded by specifying different arguments to the underlying ioctl system call. Even if several Binder commands can be specified, within the context of this paper we are only interested in the BINDER_WRITE_READ Binder command, as this is the main mechanism through which all RPC invocations are performed. Such mechanism is used in two scenarios: when an app invokes a remote function implemented in a system service (e.g., to send a text message) and when a system service invokes a function defined in the app (e.g., to communicate user input).

The BINDER_WRITE_READ command can, in turn, encode different types of operations. As a concrete example, consider the drawing in Figure 2, which shows the low-level details of a Binder transaction that encodes an RPC invocation of the sendText remote function exported by the SmsManager service. This request is automatically generated when the SmsManager.sendTextMessage API, which an app can invoke to send a text message, is called. The write_buffer argument points to a buffer that encodes all the information related to the requested operations. In particular, the *Request type* field encodes the type of the request (BC_TRANSACTION, in this example), the *InterfaceToken* field specifies the remote service (com.android.internal.telephony.ISms), and the *code* field specifies which of its exported functions must be invoked (sendText). Moreover, a number of additional function arguments can be specified (such as, in this example, the destination number). All function arguments (which could be both Java primitive types and generic Java objects) are serialized and deserialized before being written to and read from a Binder transaction.

As we mentioned, the Binder mechanism is also used as a way for the system to invoke a function implemented in application space. In this case, an application's thread invokes the ioctl syscall, and the execution is temporarily blocked waiting for data, which specifies the *local* function to be invoked and its arguments. When such data becomes available, the syscall will unblock its execution, and the application will be able to read from a given buffer the data sent from a different process.

4.3.2 Handling Component-Component Interactions

An operation that is performed by even the simplest Android app is to start an activity. From a Java-level point of view, this can be achieved by invoking the startActivity API which takes as argument the details of the activity that needs to be started. From a syscall-level point of view, the execution of such API will generate two Binder transactions. First, the startActivity API will generate a call to the startActivity *remote* function exported by the ActivityManager service. Then, this service will call the scheduleLaunchActivity *local* (i.e., method executed by the application process) function that specifies how the new activity should be created (by defining, for example, its appearance). To correctly execute orig within the context of stub, monitor needs to patch the arguments of both Binder transactions, as described next.

The startActivity API takes as argument the activity to be opened, in the form of <package_name, activity_name>, where package_name is the package name identifying an installed app and activity_name is the name of a specific activity. The problem lies in the fact that when orig's code invokes the startActivity API (that is executed within the context of stub), it specifies orig's package name as the target package_name. As this code is executed by the stub process, a SecurityException would be thrown by the ActivityManager service: in fact, stub is not allowed to open an activity of orig, since it is defined in a different app. To avoid this, monitor patches the arguments of such API by changing the specified package name from orig's to stub's. Note that the system would generate an exception if a *compatible* activity is not found in stub. For this reason, we build the stub app in a way that it defines (in its manifest file) a *dummy* activity for each activity defined in orig.

The arguments of the scheduleLaunchActivity function (invoked by the ActivityManager service) need to be mod-

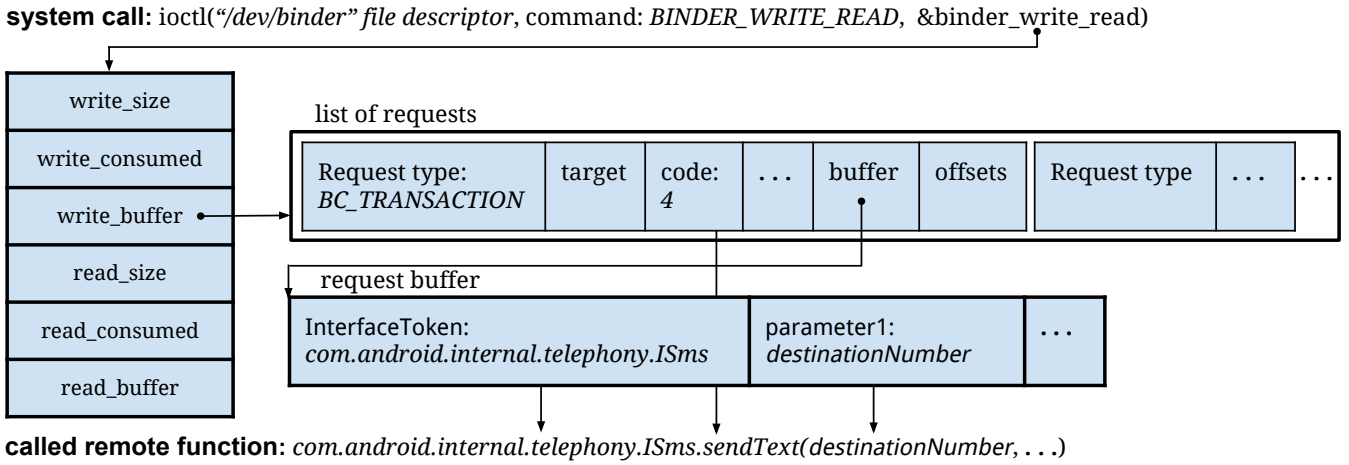

Figure 2: Example of a Binder transaction used to send a text message. Concrete values are written in *italic*. Arrows starting with a dot represent memory pointers. Note how the same `ioctl` syscall can specify more than one *request*.

ified as well. In fact, since the `monitor` process patched the first Binder transaction (as described in the previous paragraph), the framework will now try to spawn the *dummy* activity defined in the `stub`. This, in turn, would lead to a fatal exception, as `stub` does not contain any code associated to such activity. For this reason, `monitor` needs to patch the second Binder transaction as well. In particular, `monitor` patches the `info` argument, an instance of the `ActivityInfo` class, which contains all the information on how an activity should be created. This argument is modified at run-time with an `ActivityInfo` object *compatible* with the original target `orig`'s activity. To obtain an appropriate `ActivityInfo` object, `monitor` uses the `getPackageInfo` API. In this way, `orig`'s code is able to transparently start its activities, even when run within the execution context of another application, `stub`.

Similarly, NJAS patches the interactions with the services defined within an application. In particular, `monitor` needs to patch the Binder transactions related to the `startService` and the `scheduleCreateService` methods that are the analogous for services of the `startActivity` and `scheduleLaunchActivity` methods. As the low-level mechanism is equivalent to the one discussed for handling activities, we omit the technical discussion.

4.3.3 Handling Introspective Calls

Android apps often invoke a number of APIs to obtain information about themselves. These API calls are almost never performed by the app's codebase, but they are used by the underlying Android framework APIs. An example of such APIs is the `getActivityInfo` API. This API returns an `ActivityInfo` object, which contains a number of important information. As this API is invoked within the context of `stub`, its returned value needs to be properly patched, so that the execution of `orig`'s code can continue without crashing. The `monitor` process performs this patching procedure at run-time.

4.3.4 Additional Challenges

In the previous sections we have described how NJAS modifies the arguments of several Binder transactions. However, during our discussion we omitted some important low-level challenges that we needed to address, which we now discuss.

The first challenge arises when patching the content of the *request* buffer used by Binder transactions of `BR_TRANSACTION` type. Specifically, for some requests, the `offsets` field points to a data structure which contains a list of *absolute* pointers to the buffer itself. The problem is that when the buffer is patched by the `monitor` process, these absolute offsets need to be properly updated, so that they keep pointing to the right location. As a solution, we automatically update these offsets when necessary.

The second challenge is related to patching the request buffer used by Binder transactions of `BR_REPLY` type. The peculiarity of this buffer is that it is placed by the Binder kernel module in a memory region that is mapped (through the `mmap` syscall) to the `/dev/binder` device file. In this case, the problem is that this file is mapped as read-only, and the `monitor` process is not allowed to modify it. Note how `ptrace` can usually modify non-writable memory pages, but not when they are `mmap`-ed in this way. To address this issue, we copy the content of the buffer in a writable memory region, and we then modify the `buffer` field, so that it points to the beginning of such a copy.

4.4 Security Policies

Our approach can be used to enforce a variety of different security policies that restrict different capabilities and permissions of an Android device. Our current prototype can effectively enforce security policies related to four different categories of sensitive operations. In particular, NJAS can enforce fine-grained security policies related to connecting to the network, accessing the file system, sending text messages, and accessing the user's contact list. Our prototype can be easily extended to support additional categories of permissions, the only challenge being the required engineering effort. In fact, note that the currently supported permissions already cover technically-different categories of operations. In the remainder of this section, we describe the details on how NJAS handles the different cases.

To block unwanted Internet connections, NJAS intercepts the `connect` and `sendto` syscalls and parses the `addr` ar-

gument, which specifies the target IP address. By using this information, the enforcing mechanism decides whether to block a connection attempt, by, for instance, consulting a user-defined white-list of allowed IP addresses. Similarly, NJAS can restrict file system access. This functionality could be useful as many apps store potentially-sensitive files (e.g., user's photos) in publicly accessible locations on the device's file system, such as the SD card. For this reason, we give the user the possibility to limit the capabilities of untrusted applications to a specified set of file paths.

Another area where NJAS can enforce a fine-grained user-defined policy is related to the capability of sending text messages. In particular, the user might want to block sending text messages to a specific set of phone numbers. NJAS can enforce such finer-grained policy by intercepting all Binder requests that are used to invoke one of the following remote functions exported by the `SmsManager` service: `sendText`, `sendData`, and `sendMultipartText`.

Finally, NJAS can block an application from accessing the user's contact list by preventing its access to the `ContactManager` content provider. To this aim, NJAS blocks all the invocations to the `getContentProvider` remote API (exported by the `ActivityManager` service), when it is used to get a reference to the `ContactManager` content provider.

Note that our enforcing mechanism is implemented in a way that the execution of the app will continue its execution, even if it attempts to perform an operation that violates a security policy. This is achieved by hot-patching the argument values of each syscall with non-valid ones, instead of abruptly aborting the syscall invocation.

4.5 Security Discussion

In this section we discuss how a malicious application could evade our analysis system, and the countermeasures we implemented.

First, note that, by using syscall interposition, NJAS is able to *fully* control the behavior of an app. In fact, the usage of the `ptrace` mechanism guarantees that the `monitor` process will be able to intercept *every* syscall invoked by `orig`, independently from *how* they are invoked. In particular, an application can (directly or indirectly) invoke a syscall in three different ways: by invoking a Java-level API, by calling a function implemented in a native library (e.g., `libc`), or by directly invoking a syscall through inline assembly code (e.g., by executing the `svc` ARM assembly instruction). In all these three cases, the `monitor` process will be notified of the generated syscall, and it will have a chance to block it and/or modify its arguments. Note that, in contrast to NJAS, all the existing approaches based on bytecode rewriting techniques cannot properly sandbox an application in all these cases (especially when dealing with the last scenario).

A malicious app could try to actively evade our sandbox. However, since `ptrace` is a kernel-level functionality, there are only very few ways to interfere with this mechanism. In particular, an app could invoke the `ptrace` and `kill` syscalls to interfere with the `monitor` process. To prevent this, `monitor` forbids the execution of these two syscalls by the monitored code. This approach does not interfere with the execution of benign apps. In fact, the Dalvik VM does not use these syscalls, and apps that include a native code component are supposed to use it only to execute CPU-intensive tasks.

As a last consideration, NJAS can neither prevent the user to manually open `orig` nor it can prevent `orig` from starting at boot. This could be problematic as the `monitor` process would not have a chance to *monitor* the application, as it would be executed outside the context of `stub`. However, in practice, this does not constitute a problem: First, a user can simply use the `manager` app to load `orig` in the sandboxed environment; Second, an app that is installed but never opened will not start at boot automatically; Third, a user can easily prevent an app from starting at boot by closing it using the "Force stop" Android option. In fact, after an app has been closed in this way, it will not automatically start at boot, unless it is manually opened again.

5. EVALUATION

In this section, we describe how we evaluated NJAS. First, we discuss how our system is able to correctly sandbox an application we wrote in order to stress-test our system, as well as several real-world apps. Second, we measure the performance overhead introduced by our system, and we compare our results against FireDroid [24], the current state-of-the-art sandboxing technique based on syscall interposition. For our tests, we used a Samsung Galaxy Tab 10.1 (1 GHz dual-core Nvidia Tegra 2 with ARM Cortex-A9 CPU, 1 GB RAM), running Android 4.2.2.

5.1 Effectiveness

To test the effectiveness of NJAS, we first developed a simple application performing the sensitive operations that our current prototype is able to monitor and, if necessary, block. In particular, this application attempts to perform the following operations: initiate several network connections, access sensitive files on the SD card (such as the user's photos), send text messages to premium numbers, and access the user's contact list. The application also declares in its manifest all the permissions that are needed to perform these operations.

We then defined a set of security policies that would exercise the different components of our sandbox. For example, we created a series of security policies to enforce the following behaviors: block connections to network endpoints other than Google, block accesses to the *photos* folder on the SD card, block text messages directed to premium numbers, and block access to the user's contact list. We then used NJAS to enforce such security policies, and we verified that the application was not able to carry out any unwanted behavior.

As a second part of our evaluation, we used our prototype to sandbox the execution of 20 real-world applications. In particular, we selected 7 real-world popular apps from the *top-free* lists of 7 different *app categories* on the Google Play market. Table 1 shows the list of applications we selected for each category. We selected these apps as they are popular representatives of different typologies of applications (the Google Play market does not provide precise statistics, but each of these apps has currently more than 10 million installations). In addition, we selected other 13 apps from a set of application randomly downloaded from the Google Play market.

To test our system, we executed each one of the selected 20 apps for about two minutes, and we manually stimulated their functionality. For each of these apps, we did not notice any difference between running them with or without

Application Name	Version	Category
Angry Birds	3.0.0	Game
Instagram	5.0.5	Social
Bank of America	4.3.255	Finance
Super-Bright LED Flashlight	1.0.3	Tools
Q Droid	5.4.3	Productivity
Dictionary - Merriam-Webster	2.1	Books & Reference
Barcode Scanner	4.5.1	Shopping

Table 1: Details of popular applications we used to test the applicability of Njas in a real-world scenario.

our sandbox. In other words, the application did not crash and its behavior appeared to be the same. Our tests also showed that the sandboxing mechanism is effective. For example, when executing the Angry Birds app by enforcing a policy that blocks all network connections, we were able to normally play the game (since this app's gameplay does not need any Internet connectivity), the only difference being the missing presence of the advertisements: this is expected, as the used advertisement framework relies on network connectivity.

5.2 Performance

We now discuss the performance overhead caused by instrumenting Android apps with our system. We tested the performance impact of Njas by using a popular Android benchmark app, called Quadrant (version 1.11). We specifically chose this application for three reasons. First, using an app specifically designed to benchmark an Android device (as opposed to general-purpose applications) allows us to get an objective and deterministic evaluation of the overhead introduced by Njas. Second, this benchmark app tests the performance of a device under many different aspects (e.g., CPU time, memory speed, I/O operations speed, graphic rendering time). Finally, this application is used as part of the evaluation of FireDroid. This simplifies a direct comparison with the state-of-the-art research work that is the most similar to ours (in terms of sandboxing technique). In fact, FireDroid achieves sandboxing through `ptrace`-based syscall interposition, with the main difference that it requires root access to the device.

Table 2 shows the overhead of our approach with respect to the baseline and FireDroid. The table reports the numbers that the Quadrant benchmark application produced when executed. On the one hand, Njas has a very small overhead for CPU, memory, and GPU operations. On the other hand, it has a high performance impact on I/O operations. This is an expected result since the I/O performance overhead is related to the rate at which an app generates syscalls, which is, in turn, directly linked to number of context switches added by the `ptrace`-based syscall filtering which Njas uses.

Results show that the performance overhead introduced by Njas, while not being ideal, is acceptable. First, Table 2 shows that our results are in line with those of FireDroid. Moreover, we note that the one of the goals of our work is not to develop a more efficient implementation of the syscall interposition mechanism, but to show how a complete sandboxing mechanism can be implemented without requiring root access to the device. Second, as we mentioned in the previous section, we did not notice any performance hit when we executed the real-world applications, even when testing heavily interactive games, such as Angry Birds. Nonetheless, we acknowledge there is space for improvement in this direction, which we discuss in the next section.

6. LIMITATIONS

In this section, we discuss the limitations of our approach and of our current prototype. We also suggest future work that can address these limitations.

The first limitation is related to the Intent mechanism. From a practical point of view, an Intent is a Java object that encodes the willingness to perform an action. There are two types of Intents, explicit and implicit. An Intent is explicit when the target component is explicitly indicated in the Intent object itself. Njas cannot intercept these messages, but this does not constitute a big problem: in fact, the usage of explicit Intents is normally restricted to intra-app communication (as specified in the official Android documentation[2]). Differently, an Intent is implicit when it only specifies the type of action that needs to be performed, without explicitly specifying the target. Moreover, apps can register themselves as *available* to perform such actions. The current implementation of Njas does not support the usage of implicit intents. However, a possible extension could be to create `stub` in a way that it would systematically receive all the implicit intents that the original app could possibly handle.

Another limitation is related to the fact that Njas requires read access to the APK file of the app to sandbox. For this reason, our system cannot currently sandbox paid applications, since they are stored in a location that cannot be normally read by unprivileged applications. Note that all the approaches that rely on application-rewriting techniques have the same requirement, as they need access to the original application's codebase. For this reason, all such techniques cannot currently handle paid applications as well.

As a future work, the possibility of performing "time of check to time of use" (TOCTOU) attacks against our current `ptrace`-based implementation should be studied and mitigated as explained in [16]. In fact, it may be possible for an app to bypass some of the checks we implemented, by using a malicious thread that exploits the time gap between the moment in which our monitoring process checks the parameters of a syscall and the moment in which these values are actually read by the kernel.

In addition, an attacker could specifically detect the fact than an app is run by Njas and consequently modify or halt its execution by, for instance, detecting that the app's threads are monitored using the `ptrace` interface. How to avoid detection may be studied in future work, however, it is known that avoiding detection of monitoring systems is typically an arms race.

[2] `http://developer.android.com/guide/components/intents-filters.html#Types`

Test	Baseline	NJAS	NJAS overhead	FireDroid overhead
CPU	4333	4055	6.87%	5.2%
Memory	2210	2200	0.47%	3.9%
I/O	3892	1833	112.32%	97.5%
2D	96	96	0.00%	0.2%
3D	1960	1783	9.90%	3.3%

Table 2: Overhead introduced by Njas, compared against FireDroid. The numbers are obtained by using the Quadrant benchmark app (version 1.11). Note that the *Baseline* represents the performance of the system without any instrumentation, and that higher numbers correspond to better performances. The results have been averaged on 5 runs of each test.

As we discussed in Section 5, NJAS suffers from some performance overhead. While this is in line with state-of-the-art approaches that are based on syscall interposition, we believe that there is space for improvements. For example, in the near future it might be possible to use the `seccomp/BPF` functionality (officially introduced in Linux kernel 3.5) to perform syscall interposition on non-rooted, stock Android devices. Such functionality is not available yet (in fact, the latest version of Android runs Linux 3.4.0), but, once it becomes available, it could be used to develop a system with better performance. For example, it would be possible to integrate the techniques implemented in MBOX [19] to sandbox Linux processes more efficiently.

The implementation of our prototype does not currently support the enforcing of fine-grained policies for all the categories of sensitive operations that an app can perform. We note that it would be conceptually simple to extend our prototype to support additional permissions, the only challenge being the required engineering effort. We also note that the main goal of our work is to show how an approach like ours can be used to enforce fine-grained user-defined security policies on devices running non-rooted stock Android, and not to provide a full implementation that covers all the possibilities. Moreover, previous research [16, 24] has shown that supporting more permissions is indeed technically possible. An interesting direction for future work would be to integrate these works with our system.

Finally, while the tests performed in Section 5.1 clearly show that it is possible to use NJAS to instrument the execution of complex real-world applications, we acknowledge that the current prototype implementation of our system is not compatible with all the apps. This is due to the limitations already discussed in this section and the missing handling of specific syscalls or Binder transactions. However, our prototype clearly shows that extending our support to a wider range of syscalls and Binder transactions is a viable option, although it would require some engineering effort.

7. RELATED WORK

Several papers analyze the current Android permission system [13, 27, 4]. In particular, these works highlight the main problems that affect the current permission system (e.g., being too coarse-grained or not customizable by the user). These studies motivate the necessity of better sandboxing mechanisms, such as the one we propose.

Other works focus on analyzing Android applications to discovery *confused deputy* vulnerabilities, through which a trustworthy app could be lured into misusing its permissions [17], leaking private information [20, 31], or executing code on behalf of a malicious application [21]. These works show how confused deputy vulnerabilities constitute a real-world threat and motivate even more our work. In fact, the ability to sandbox an Android application could be used to reduce its permissions to adhere to the principle of least privilege: this sensibly mitigates the impact of a malicious app exploiting a confused deputy vulnerability.

Furthermore, several other works propose modifications to the Android framework to better sandbox Android apps. In particular, Cells [2], MOSES [23], and AirBag [29] implement a virtualization-based system, while others propose lighter modifications specifically designed to prevent the leakage of user information [32, 7]. Additionally, already-existing mandatory access control mechanisms, such as TOMOYO Linux [8] and SE Linux [25], have been ported to Android. However, the adoption of these systems on a large scale is complex, since they require the installation of custom Android versions on a device, or their integration within the official Android distribution. FireDroid [24] implemented a sandboxing mechanism based on syscall interposition using `ptrace` that requires minimal system modifications. This work shows that syscall interposition can be effective in enforcing user-defined policies in Android. However, FireDroid still requires having root access on the device on which it is used, an option that is not available in stock Android.

Concurrently and independently to our work, Boxify [5] proposes a sandbox mechanism for Android apps sharing the same goal of NJAS: providing sandboxed execution of Android apps in an unmodified system. Even though Boxify and NJAS achieve the same goal, they have significantly different design choices that result in different security properties. Specifically, Boxify relies on executing apps in an "isolated process" (a special type of process in Android, running with no permissions) and monitoring it by using a *Broker* process holding the union set of all permissions required by all the apps possibly hosted by Boxify (all the Android permissions, in a typical usage scenario). This makes the Boxify process a potential target for an attacker that could try to escape the Boxify sandbox to gain the wide range of privileges the Boxify process holds. Moreover, the implementation of Boxify needs to reimplement parts of the system services (e.g., the `PackageManager` service) and thus replicate the numerous security checks normally performed by these system services. For this reason, the task of securely implementing and updating (when newer Android versions are released) the Boxify codebase can be difficult. Finally, Boxify relies on GOT patching (instead of `ptrace`) to intercept the execution of the sandboxed app. This technique is easily escapable by a sandboxed process (which can perform syscalls without using any library). To mitigate this threat,

Boxify relies on the compartmentalization imposed by the Android operating on "isolated process" (e.g, isolated processes cannot perform any operation requiring an Android permission). However, this still exposes the entire kernel layer to a malicious sandboxed app, without giving any possibility to the Boxify monitoring process to intercept the potentially malicious syscalls executed by it. Nevertheless, we envision that NJAS can be used in conjunction with Boxify to provide an additional layer of security.

Many different works propose to automatically repackage Android apps using bytecode rewriting techniques, instead of modifying the Android OS. Specifically, AppGuard [6], I-ARM-Droid [10], RetroSkeleton[9], and Dr. Android [18] monitor potentially dangerous Java method invocations. Differently, Aurasium [30] works at the native code level by patching the GOT of native libraries. Differently from NJAS, none of these systems is able to completely monitor the behavior of an app. For instance, a malicious app could evade all of them by using native code that directly executes syscalls through inline assembly code. On the contrary, NJAS is able to monitor an app in its entirety, independently from how it operates. Additionally, approaches relying on app repackaging techniques have important usability problems, as we have already discussed.

Finally, the general problem of sandboxing a process by performing syscall interposition has been extensively studied by several previous works. In particular, Garfinkel et al. [15, 16] discuss several common challenges that affect approaches based on syscall interposition, and provide a series of recommendations on how to address them. We took these recommendations into account while designing and implementing NJAS. Different implementations of syscall-based sandboxing have been proposed in the literature, such as [28, 22, 26]. All these solutions require, however, modifications to the operating system on which they are used. More recently, MBOX [19] has shown how syscall interposition can be used to effectively sandbox a Linux process, without having root privileges. As previously discussed, MBOX performs syscall interposition by using the `seccomp/BPF` mechanism. We could not use this mechanism in NJAS, as it is not available yet on the Android platform. However, the possibility of integrating such mechanism with our approach represents one of the most promising and interesting future work direction.

8. CONCLUSIONS

In this paper, we proposed a novel approach for sandboxing Android applications. Our system comes with several usability and security advantages over previous work. Specifically, it does not require modifications to the Android framework, root access to the device, or to create a modified version of an existing app, which could not be installed without enabling unsafe options. At the same time, our approach is able to completely sandbox an application, even when it contains native code components. We implemented this approach in a tool, called NJAS, and we showed how it can be used to effectively sandbox different popular real-world applications, with an acceptable performance overhead. Finally, we discussed the limitations of the current implementation and we propose several directions for future work.

Acknowledgments

This material is based upon work supported by DHS under Award No. 2009-ST-061-CI0001, by NSF under Award No. CNS-1408632, 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 DHS, NSF, 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

9. REFERENCES

[1] AndroidCentral. Larry Page: 1.5 million Android devices activated every day. `http://www.androidcentral.com/larry-page-15-million-android-devices-activated-every-day`.

[2] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a Virtual Mobile Smartphone Architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[3] AppBrain. Number of available Android applications. `http://www.appbrain.com/stats/number-of-android-apps`.

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

[5] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-Fledged App Sandboxing for Stock Android. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2015.

[6] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. Styp-Rekowsky. Appguard - Real-Time Policy Enforcement for Third-Party Applications. 2012.

[7] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the ACM Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[8] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[9] B. Davis and H. Chen. RetroSkeleton: Retrofitting Android Apps. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[10] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. *Mobile Security Technologies*, 2012.

[11] EFF. Google Removes Vital Privacy Feature From Android, Claiming Its Release Was Accidental.

https://www.eff.org/deeplinks/2013/12/google-removes-vital-privacy-features-android-shortly-after-adding-them.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[14] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[15] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2003.

[16] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2004.

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

[18] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[19] T. Kim and N. Zeldovich. Practical and Effective Sandboxing for Non-Root Users. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2013.

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

[21] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2014.

[22] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2003.

[23] G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: supporting operation modes on smartphones. In *Proceedings of the Symposium on Access Control Models and Technologies (SACMAT)*, 2012.

[24] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark. FireDroid: Hardening Security in Almost-Stock Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[25] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2013.

[26] W. Sun, Z. Liang, V. Venkatakrishnan, and R. Sekar. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2005.

[27] T. Vidas, N. Christin, and L. F. Cranor. Curbing Android Permission Creep. In *IEEE Web 2.0 Security and Privacy Workshop (W2SP)*, 2011.

[28] D. A. Wagner. *Janus: an approach for Confinement of Untrusted Applications*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1999.

[29] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2014.

[30] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2012.

[31] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2013.

[32] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)*, 2011.