The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android

Jie Huang
CISPA, Saarland University
Saarland Informatics Campus
huang@cs.uni-saarland.de

Oliver Schranz
CISPA, Saarland University
Saarland Informatics Campus
schranz@cs.uni-saarland.de

Sven Bugiel
CISPA, Saarland University
Saarland Informatics Campus
bugiel@cs.uni-saarland.de

Michael Backes
CISPA, Saarland University
Saarland Informatics Campus
backes@cs.uni-saarland.de

ABSTRACT

Third-party libraries are commonly used by app developers for alleviating the development efforts and for monetizing their apps. On Android, the host app and its third-party libraries reside in the same sandbox and share all privileges awarded to the host app by the user, putting the users' privacy at risk of intrusions by third-party libraries. In this paper, we introduce a new privilege separation approach for third-party libraries on stock Android. Our solution partitions Android applications at *compile*-time into isolated, privilege-separated compartments for the host app and the included third-party libraries. A particular benefit of our approach is that it leverages compiler-based instrumentation available on stock Android versions and thus abstains from modification of the SDK, the app bytecode, or the device firmware. A particular challenge for separating libraries from their host apps is the reconstruction of the communication channels and the preservation of visual fidelity between the now separated app and its libraries. We solve this challenge through new IPC-based protocols to synchronize layout and lifecycle management between different sandboxes. Finally, we demonstrate the efficiency and effectiveness of our solution by applying it to real world apps from the Google Play Store that contain advertisements.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**;

KEYWORDS

Android Runtime, App Compartmentalization, Third-party Libraries

1 INTRODUCTION

Third-party libraries are constituent parts of mobile apps and help app developers to quickly deploy common utility functionality or leverage services, such as analytics or app monetization. However, past experience has shown that those third-party libraries do not only provide convenience [19, 26, 34, 35], but also bare risks for the users' privacy. On Android, particularly the fact that third-party libraries and their host apps share the same sandbox has been identified as a means for nosy libraries to exploit their ambient authority to tap into device-local resources, such as location tracking, phone identifiers, or users' private data, which can be of high interest to external parties, like advertisement networks.

In light of those risks, the security community has recently proposed different approaches to tame overly curious or even maliciously acting libraries, where the focus clearly lies on privilege-separating the notorious advertisement libraries. The proposed solutions range from completely removing the library payload, dedicated advertisement system services [27, 32] and system modifications [31, 39] to application bytecode rewriting [25, 36, 40]. However, while those solutions greatly benefit the users' privacy, they do not entirely satisfy deployment restrictions from the end-users' perspective. Unfortunately, application or system modifications are unavoidable in the currently proposed solutions. Modifying applications breaks the same origin policy of Android application updates, since the original app has to be repackaged and resigned. As a consequence this repackaged app version can no longer update automatically. System modifications, on the other hand, are notoriously hard to distribute to end-user devices and distribution via after-market ROMs is generally considered as a too high technical hurdle for most layman end-users.

In this paper, we propose an alternative approach to privilege-separation of untrusted advertisement libraries in Android apps by using *compiler*-based instrumentation of apps. Since compilation is an integrated, standardized part of app installation, compile-time modifications do not require the target application to be repackaged and resigned, hence abstaining from breaking the application signature. Moreover, Android's *dex2oat* on-device compiler can be operated entirely at the application layer and does not require changes to the application framework or system image. As such, compiler-based instrumentation forms a beneficial trade-off in the deployment of a library separation solution. The foundation of our approach to compiler-based library separation is a systematic study of the ten most frequently used advertisement libraries to identify the integration patterns between advertisement library and their host apps. We discover that only a small number of such patterns

CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134064

exist and that they establish only a loose coupling between libraries and host apps (e.g., callbacks, field access, or method invocations). Based on those insights, we design and implement an extension for the Android on-device *dex2oat* compiler suite, which at compile-time identifies the code segments that integrate the advertisement library into the app. It then splits the app at those integration points into two distinct apps to be installed with a strong (process) security boundary in between and with being privileged separately. The challenge of this approach is to reintegrate the now compartmentalized library with its host app, e.g., manage the event-driven advertisement and application lifecycles or ensure visual fidelity by correctly displaying advertisements. We solve this challenge in our solution through a new IPC-based protocol for synchronizing lifecycle events between the host app's and library's sandboxes as well as for synchronizing the layout management between an overlayed advertisement and the app's user interface. More concretely, we make the following contributions:

Study of advertisement library integration techniques. In order to provide a solid foundation for our solution, we thoroughly analyzed the ten most prevalent advertisement libraries in the Google Play store that represent a large fraction of the market share of apps that include advertisements. Beyond motivating the design of our compartmentalization solution, we consider the results of our study to be useful for the academic audience to facilitate independent research on the topic.

Compiler-based Application Compartmentalization. We introduce *CompARTist*, a compiler-based application compartmentalization system that enforces privilege separation and fault isolation of advertisement libraries on Android. Our approach offers a deployment alternative to existing solutions, since it does not require modifications of the firmware and does not break Android's signature-based same origin model. The primary challenge for our solution was the reintegration of the library compartment with the host through compile-time code instrumentations.

Outline. The remainder of this paper is organized as follows: In Section 2, we provide background on the advertising ecosystem on Android and present in Section 3 the findings of our study of advertisement library integration techniques. Section 4 categorizes and discusses prior related work. In Section 5, we introduce the overall architecture of *CompARTist*. Furthermore, we discuss robustness, performance, and limitations as well as future directions of our approach in Section 6 and conclude the paper in Section 7.

2 BACKGROUND

We briefly provide background information on the Android advertisement ecosystem and on advertisement libraries in context of Android's sandboxing design.

2.1 Advertising Ecosystem

There are typically three participating parties in mobile advertising on Android: publishers, advertisers, and advertising networks. Developers take the role of publishers who spare some part of their apps' user interface to show banners, interstitials, or other advertisements to their users. Advertisers provide the actual advertisements to be shown to customers. The advertising network is the broker that controls the integration and delivery of advertisements from advertisers to publishers. Figure 1 depicts a typical scenario where the advertiser entrusts the network (here Google Admob) with the delivery of advertisements. Conversely, app developers receive payments for displaying advertisements or generating clicks through their users. To ease the task of integrating advertisements into applications, the advertisement network usually provides the app developers with dedicated SDKs, i.e., advertisement libraries.

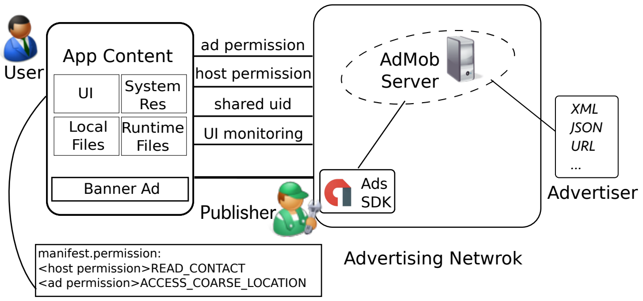


Figure 1: Advertising Ecosystem

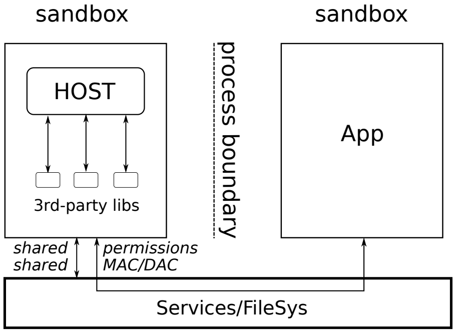


Figure 2: Default Android sandboxing

2.2 Android App Sandboxing

Figure 2 gives an overview over Android's default application sandboxing. Android's user-based permission model mandates that access to certain resources, e.g., location information or user contacts, requires the declaration of specific permissions in the application manifest. File system access, on the contrary, is enforced through the UID-based sandboxing system of the underlying Linux kernel where apps are assigned distinct UIDs and cannot access each other's files.

However, third-party libraries can undermine those security mechanisms by exploiting their ambient authority. As depicted in Figure 2, all app components share the same UID and are considered the same security principal. This kind of coarse authorization gives untrusted third-party libraries the opportunity to exploit all permissions assigned to their host application, as well as access all its files. While the introduction of a dynamic runtime permission system in Android 6 allows users to revoke a predefined set of permissions from applications, this is still enforced on the app level. Even the introduction of SELinux [33] in Android 4.3 only reduces the granularity to the process level, while component-level granularity would be required to separate library privileges. As a result, on default Android, a privacy-invading third-party library [19, 22, 26, 34, 35] can easily access and leak private resources or a vulnerable library version unnecessarily extends its host app's attack surface [28].

3 LIBRARY INTEGRATION TECHNIQUES

Statistical results from the freely available library detection tool LibScout [9, 20] indicate a low fragmentation of advertising libraries among the top apps on Google Play. As shown in the first column in Table 1, between the first and the tenth most popular advertisement library the integration rate drops down significantly from 25.94% (Google Play Services Ads) to 3.11% (Amazon Ads). In particular, this means that analyzing the ten most popular advertisement libraries allows us to cover a large fraction of all applications shipping advertisement code. Since the focus of our study is on how a host app can integrate a library, we checked the possible integration patterns by analyzing the libraries' official API documentations. For those libraries that did not provide a full list of public APIs, we use Oracle's Java class file disassembler `javap` to extract the public fields and methods from the library's codebase. Table 1 summarizes the results of our study on possible integration techniques of advertisement libraries into host apps.

Method Invocation and *Field Access* are the two most common integration techniques among all libraries. Typically, method invocation and field access are used to exchange data between the host and the library, e.g., to request loading of an advertisement or to retrieve advertisement information.

We observed two possible techniques for deriving subclasses from library code in order to integrate the library into the app: *Class Inheritance* and *Interface Implementation*. Libraries use those techniques to allow host apps to register callback components to react to certain events, such as displaying or closing an advertisement. In many cases, the callback methods are triggered with library-specific objects as parameter values. This intertwines the library and host tighter than, e.g., method invocations and field accesses, making the library's separation more challenging (as discussed in Section 5).

Furthermore, a small fraction of advertising libraries also propagates information to their hosts by throwing customized *Exceptions* that the host needs to catch and react to.

Layout Arrangement is an integration technique that allows banner advertisements to occupy part of the host app's user interface. To integrate this kind of non-full-screen views, app developers need to make changes to their apps' UI hierarchy. There are two ways to integrate a banner view element: It can either be added in the corresponding XML resource file for interface definition or it can be instantiated and added as a new view element at runtime.

We found that all analyzed advertisement libraries require at least one permission from their host app, `INTERNET` being the most prevalent one. Further, dedicated advertisement components, e.g. `Activity`, `BroadcastReceiver`, or `ContentProvider` need to be registered for the advertisement library as well. All this requires the host app developer to make changes to the host app's *manifest file*.

Based on our findings, we conclude that most advertisement libraries share a common set of well-defined integration techniques, which makes them amenable targets for efficiently separating them at those integration points from their host apps.

4 RELATED WORK

In this section, we discuss prior works for compartmentalization of libraries as well as related works for blocking of advertisements and general application-layer approaches to enhance Android's app sandboxing.

4.1 Library Compartmentalization

We first discuss closest related works for compartmentalizing app components, in particular libraries. We categorize those existing solutions based on their deployment strategy and compare them for their respective advantages and drawbacks. Table 2 summarizes the results of this discussion.

4.1.1 System-centric Solutions. System-centric solutions usually ship a compartmentalization approach as part of the firmware (F3: ✗). This generally provides the advantage of establishing dedicated system services/processes for advertising code (F1: ✓), running monitoring code by-design with elevated privileges (F5: ✓), and avoiding changes to the apps' bytecode (F2: ✓). For instance, AdDroid [27] and AFrame [39] both introduce new system services that expose APIs for integrating advertisement libraries into applications. Trivially, a system-centric solution can always keep the signature-based same origin model of apps intact by customizing the signature verification process (e.g., whitelisting own changes). While this allows for a robust privilege separation by running advertisement code in a separate process, it also requires developers to adapt their apps to the system (F4: ✗). In contrast, AdSplit [32] takes the developer out of the loop by automatically retrofitting applications to use their system (F4: ✓). FlexDroid [31] takes an even more involved approach by modifying the operating system to introduce so called inter-process stack inspection to allow per-component permission enforcement and uses fault isolation techniques within app processes to secure the stack-inspection code (F1: ✗). Additionally, it requires developers to include custom per-component permission policies in their apps' manifests (F4: ✗).

4.1.2 Application Layer Solutions. An alternative line of work applies application rewriting and inlined reference monitoring (IRM [21]) techniques, which abstain from modifying the firmware (F3: ✓). Instead they modify the apps' bytecode, which results in repackaging and resigning of the modified code and, thus, in turn breaking Android's signature-based same origin model (F2: ✗; F4: ✓). Since those techniques modify the code prior to installation, they do not require higher privileges to operate (F5: ✓). Such rewriting and IRM techniques have previously been used in different privacy-enhancing solutions [13, 17, 18, 24, 29, 37] and the PEDAL [25] and NativeGuard [36] approaches target specifically the privilege separation of libraries. NativeGuard in particular focuses on moving *native* code libraries to a dedicated process and reconnecting them to the host via inter-process communication (F1: ✓). PEDAL, in contrast, runs host and library in the same process (F1: ✗), but restricts the library through hooking into APIs that access sensitive

Table 1: Techniques used to integrate advertising libraries with host application.

Ad Lib	Share [9]	Method Invocation	Field Access	Inherit Class	Implement Interface	Custom Exception	Layout Arrang.	Android Manifest
Google Play Services Ads[†]	25.94%	✓	✓	✓	✗	✗	✓	✓
Flurry	17.85%	✓	✓	✓	✓	✗	✓	✓
Facebook Audience	12.11%	✓	✓	✓	✓	✗	✓	✓
Google Admob	9.30%	✓	✓	✗	✓	✗	✓	✓
InMobi	6.45%	✓	✓	✗	✓	✗	✓	✓
MoPub	6.13%	✓	✓	✓	✓	✓	✓	✓
Millennial Media	5.41%	✓	✓	✗	✓	✓	✓	✓
Tapjoy	4.29%	✓	✓	✗	✓	✗	✗	✓
AdColony	3.91%	✓	✓	✓	✓	✗	✓	✓
Amazon Ads	3.11%	✓	✓	✗	✓	✗	✓	✓

✓: technique used by library; ✗: technique not used by library

[†] The successor of AdMob and comprised of several advertising networks; we only focus on the basic package that includes Banner and Interstitial ads

Table 2: Comparison of existing (advertisement) library privilege separation approaches.

	System-centric				Application layer		
Features	FlexDroid [31]	AdDroid [27]	AdSplit [32]	AFrame [39]	NativeGuard [36]	PEDAL [25]	*CompARTist*
F1: Robust Privilege Separation	✗	✓	✓	✓	✓	✗	✓
F2: Preserves Same-Origin Model	✓	✓	✓	✓	✗	✗	✓
F3: No Firmware Modification	✗	✗	✗	✗	✓	✓	✓
F4: Developer Agnostic	✗	✗	✓	✗	✓	✓	✓
F5: No privilege escalation/App virt.	✓	✓	✓	✓	✓	✓	✗

✓: Solution provides feature; ✗: Solution does not provide feature

resources. Lastly, although not designed for privilege-separation of third-party libraries but instead of `WebView` components by using app rewriting techniques, the very recent WIREFrame [16] shares some design ideas with our *CompARTist* (see later Section 5), e.g., in that it establishes an IPC-based channel between host app and remote `WebView` for remote procedure calls, lifecycle management, or restoring visual fidelity. In *CompARTist* we, in contrast, show how such a channel can be established through a compiler-based rewriting.

4.2 Advertisement Blocking

The growing popularity of mobile advertisements also gave rise to a range of approaches that, in contrast to compartmentalization and monitoring, follows a more extreme path and blocks advertisements altogether. The downside of this approach is that it inhibits the free distribution model by reducing the developers' revenue from displaying ads. Tools such as AdAway [2], AdGuard [5] and AdblockPlus [4] utilize network-based filtering by either altering the device's `hosts` file or employing VPN-based content blocking. In addition, AdblockBrowser [3] provides a fully-featured browser with a deeply-integrated advertisement blocking functionality. In contrast, APKLancet [38] is capable of pruning a range of third-party libraries, in particular, advertisement libraries, by removing the libs' code from the app's codebase. In-app ad-blocking solution [12] utilizes app virtualization to strip ads from apps.

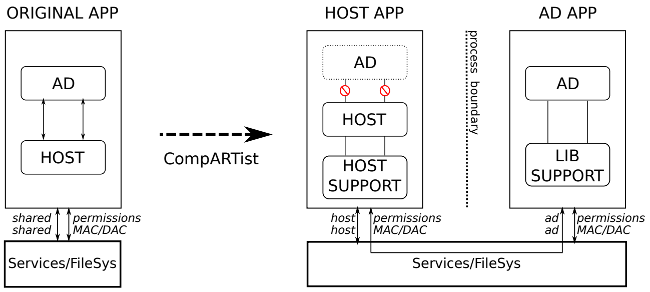


Figure 3: System overview of *CompARTist*.

5 SYSTEM DESIGN

We present the design and implementation of *CompARTist*.

5.1 System Overview

The overall design of our *CompARTist* is depicted in Figure 3. The goal of *CompARTist* is to privilege-separate advertisement libraries from their host apps with a strong security boundary between library and host app. Since Android's privileges are bound to UIDs, we opted in our solution for splitting an ad-supported target app into two different applications, each with a distinct UID. This separates advertisement libraries into a separate process with separate privileges through a distinct UID (F1: ✓). Since advertisement libraries are usually integrated into their host app (see Section 3), the

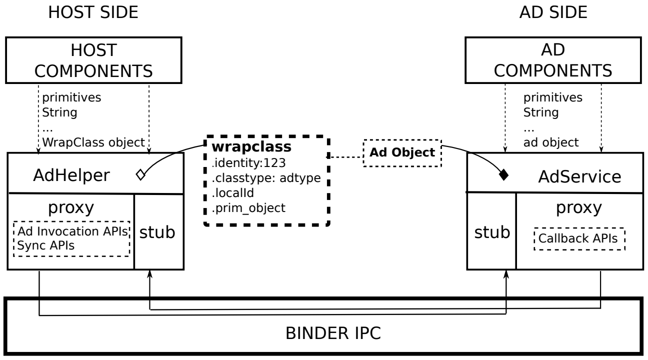


Figure 4: Inter-application Communication Channel

primary challenge for such an approach is to re-integrated the host app and library across process boundaries. While such separation and re-integration can be achieved through firmware extensions or application rewriting (see Section 4), we present a new trade-off in the design space for Android security solutions by establishing such separation and re-integration based on an extension of the dex2oat on-device compiler. Operating entirely at application-level and at compile-time, this approach abstains from firmware modifications (F3: ✓), app repackaging and resigning (F2: ✓), and app developer involvement (F4: ✓) by relying solely on the ability to load the app code produced by an extended compiler backend[1] (F5: ✗).

In the remainder of this section, we explain the design and implementation of the three main components of our solution: 1) a new IPC-based channel between host app and library that makes the previously locally integrated library remotely callable and, further, allows to synchronize the runtime states between library and app (Section 5.2); 2) an extension for the dex2oat compiler that integrates host support for the new communication channel into the host app and replaces the library through an opaque proxy for the separated library (Section 5.3); and 3) a new advertisement service app that encapsulates and privilege-separates the advertisement libraries as well as displays the ads on screen (Section 5.4).

5.2 Inter-Application Communication Channel

Since the originally app-local procedure calls to advertisement libs are not possible anymore in an isolated lib design, we need an inter-application communication channel to deliver such calls remotely across process boundaries. We take advantage of the Binder framework [30], Android's inter-process communication (IPC) mechanism, to replace the original calls to the advertisement library with remote procedure calls and transfer data, such as method parameters, between the host app and advertising service app. Figure 4 illustrates this channel and its components are explained in the following.

5.2.1 Communication Protocol and APIs. The first general challenge for our solution is the handling of data marshalling. On Android, any data that should be transferred via Binder IPC has to be either a primitive type (e.g., integer), String or a complex type, like a class, that implements the Parcelable interface to marshal the complex type into primitive types for transmission. However, library classes that were never intended to be sent via IPC, since they are only used in local invocations, do not implement this interface and are by-design not transmittable via Binder IPC. As a consequence, our channel cannot be used to transmit them, because it is unclear how to marshal and unmarshal those complex library classes. Thus, in *CompARTist*, we build on a generic protocol for remotely creating and operating on objects of library classes: those objects are constructed and stored at the ad service side and references to those objects are passed via IPC to the host app, which can use those references to invoke methods or access fields on the referenced objects. As generic, parcelable container data structure to transmit method parameters, parameter type information, and references to class instances in our protocol, we introduce a heterogeneous key-value store with corresponding serialization and de-serialization logic called *WrapClass*.

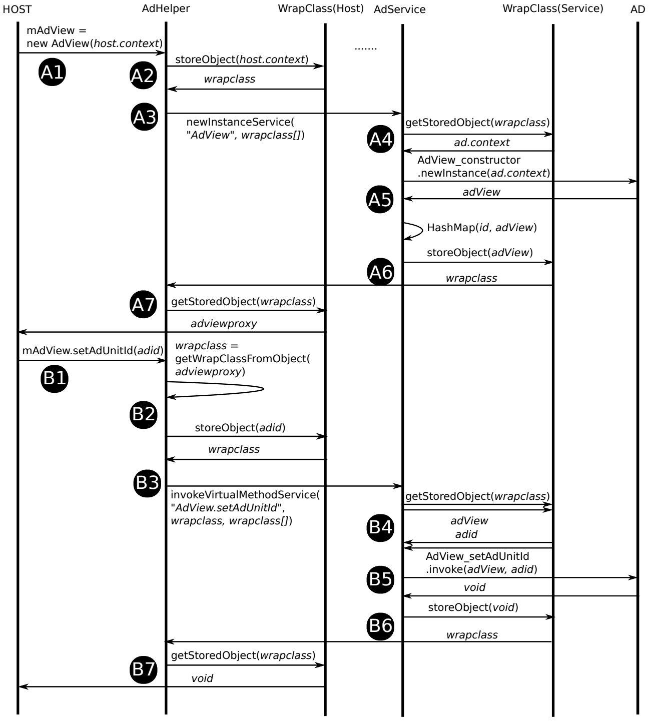


Figure 5: Example protocol run for creating a new AdView instance and calling method setAdUnitId(String) on this instance.

We define three kinds of interfaces for our new inter-application communication channel that host app and advertisement service app can use to call each other via above mentioned *WrapClass*-based protocol: *advertisement invocation API*, *callback API*, and *synchronization API*. For each of those interface types, we automatically create Stub and Proxy classes using Android's AIDL[2] feature. Those classes make these communication channels more easily accessible

[1] *CompARTist* requires access to a particular protected directory of an app to replace the oat file that is loaded by the system. Escalated privilege, e.g., root access, is needed merely to overwrite the original oat file.

[2] https://developer.android.com/guide/components/aidl.html

for the host app and ad service app, respectively. The full interfaces for each of those interface types are listed in Appendix A. A particular benefit of these APIs is that they abstract from library specific methods, thus avoiding the need to generate a tailored `Stub` and `Proxy` for every available advertisement library and easing the process of adding support for new libraries.

(1) Advertisement Invocation API. Generally speaking, there are three ways for host components to communicate with the advertisement library (see also Section 3): instance creation, field access, and method invocation. For each of those three operations, the operation type, the operation target, and any optional parameters identify a concrete library invocation event. To better illustrate this, consider the example library invocation in Figure 5 where the host app creates a new `AdView` instance on which it then calls the `setAdUnitId(String)` method. First (A1 in Figure 5), the host requests to create an `AdView` object using the host's context. This request will be processed by *AdHelper*. *AdHelper* uses *WrapClass* to store the host's `Context` instance (A2). Since the `Context` is a non-parcelable class, *WrapClass* will only store the type information of this context parameter, i.e., class type. This *WrapClass* instance forms the container of the original context instance and together with the type information of the referenced target object (i.e., a `Context`), it is passed to the remote advertisement app through our generic IPC API as parameter of a `newInstanceSerivce` (A3) call for "AdView". This API call instructs *AdService* to create a new local object with the type "AdView" (1st argument) and constructor parameters stored in the *WrapClass* (2nd argument). Thus, *AdService* first retrieves the stored object as the local `ad.context` parameter from the *WrapClass* object (A4). With the target class type "AdView" and constructor parameter, a new `AdView` object is created using the *AdService*'s context (A5). Since the channel is agnostic towards the exact library, *AdService* uses the Java reflection API to call the constructor of a class specified by the target class type parameter. This new object is stored locally in a `HashMap`, using a reference ID as key. To reply to the host and return a reference to this new `AdView` instance, *AdService* stores a reference (i.e., ID) together with all type information in a new *WrapClass* that it returns to the host (A6). The host creates a new proxy for this remote `AdView` object using the received type information (A7). The *WrapClass* object will also be stored in the proxy in order to establish the reference from the proxy object to the remote object.

Using such proxies, the host can invoke methods on the referenced remote objects. In Figure 5, the host invokes the `setAdUnitId(String adid)` method on the proxy (B1). To this end, the host stores the `adid` parameter in a *WrapClass* object and retrieves a *WrapClass* to reference the remote `AdView` object (B2). Afterwards (B3), it instructs the *AdService* to invoke the method "setAdUnitId" of the class "AdView" through the `invokeVirtualMethodService` IPC API call, where the first *WrapClass* parameter is the reference to the existing `AdView` instance on which this method should be invoked and the second *WrapClass* parameter is the argument list (i.e., wrapped `adid`). As before, *AdService* will again retrieve all parameters from the received *WrapClass* arguments (B4) and, through the reflection API, call the method on the referenced local `AdView` object (B5). It then stores the return value, here `void`, in a *WrapClass* instance (B6) and returns it to the host (B7).

(2) Synchronization API. Synchronization events only transfer meta information that indicate the supposed lifecycle state and layout of the remote advertisement. It also uses a *WrapClass*-based protocol to transfer those information, similar to invocation of ad libs explained above. The purpose of this API is the continuous synchronization and smooth integration of the remote advertisement view within the *AdService* app. More details about the operations that *AdService* executes in addition to the ad invocations explained above are provided in Section 5.4.

(3) Callback API. As mentioned earlier, integrating callbacks requires a bidirectional communication flow between host and library. To solve this problem, we implement a set of callback specific APIs that the ad service app can use to trigger a callback method in the host app. Thus, in this case the `Proxy` is located in the service app and the `Stub` in the host app. In addition, we have to distinguish two types of callbacks: interfaces and classes. In case the callback is implemented as an extension of a library class, we additionally have to make sure that the concrete implementation's constructor is not calling its parent's constructor and hence invoking library logic in the host. Therefore, we rewrite the constructor to suppress the super call. For the interface case, this is not necessary since there is no super constructor implementation. Otherwise, invoking callback APIs follows the same *WrapClass*-based mechanism we described earlier for the ad invocation API in order to invoke the callback methods of the host.

5.2.2 Communication Endpoints.

The communication protocol is carried out between two communication entities: the host side *AdHelper* within the host app and the *AdService* in the ad service app, which in turn form the shim code between the host app components and our IPC channel as well as between the ad library components and the IPC channel, respectively (see Figure 4).

AdHelper serves as the encapsulation of our newly defined IPC APIs on the host side. *AdHelper* takes care of wrapping and unwrapping data from and to *WrapClass* and bridging the gap between our communication channel and the host components. The interfaces provided by *AdHelper* are used by our compiler-based rewriter to re-integrate the remote library into the host app by replacing local advertisement calls with calls to *AdHelper* (see following Section 5.3). Similar to *AdHelper* on the host side, *AdService* forms the shim between the IPC communication channel and the library's original API on the library side.

5.2.3 Service Connection Between Host App and Ad Service.

In our current model, *AdHelper* binds itself to the *AdService* to establish the communication channel. However, this channel has to be established before any library code can be invoked by the host app in order to ensure the correct functionality of the advertising function of the host app. To solve this problem, we inject during the compilation code into the host app that scans the host app's message queue at application start to obtain the `Binder` handle of the *AdService* and then already initializes the connection to the *AdService* in a very early stage of the app's startup phase, before any *AdHelper* function is invoked, thus ensuring any library invocation finds a valid, established communication channel.

5.3 Compiler-based App Rewriting

In order to utilize our remote isolated advertisement library, we first need to retrofit host applications to actually use the newly introduced communication channel instead of the packaged library. Therefore, we need an application modification framework that can replace invocations of the local library with those to our remote version by redirecting all interaction through the new IPC-based communication channel. Splitting host app and local library, and afterwards reintegrating the host with the IPC channel requires two essential steps: First, we need to identify the boundaries between host and advertisement code. Second, we replace all those interactions with our proxies and wrappers to restore the overall library integration across process boundaries. This results in the host app being agnostic towards the fact that it no longer interacts with the packaged advertisement library but with our remote library through an IPC channel.

5.3.1 Library Boundaries. The first step towards dissecting the host application is understanding the exact interaction patterns between app and library. While we discussed general integration techniques in Section 3, we analyzed real-world applications to identify actual code patterns to be able to transform them properly. We distinguish between two cases: First, library objects or data are introduced into the host application by either invoking a method, accessing a field or instantiating a class from the advertisement library. Second, library objects or data that have been introduced to the host code earlier are passed around, characterized by method returns, field access, type checks or type casts within the host application. While only the first case depicts the boundary between host and library, both cases need to be considered when rewriting interactions to use our *AdHelper* instead. Apart from code boundary, special integration cases, such as manifest defined components and customized exception, also need specific proxy support.

5.3.2 Library Substitution. The second step is to utilize the information about the concrete code integration patterns to resect the library code and replace it with our components from our *AdHelper*. Concretely, we utilize an app instrumentation framework that is capable of merging *AdHelper* into the application and replace said code parts with our alternatives. In the following, we will first introduce the general structure of the host-side instrumentation part of *CompARTist* and then deep-dive into the rewriting routines as they pose one of the major challenges in establishing this new remote library connection.

5.3.3 ARTist Instrumentation. In this work, we leverage the Android app instrumentation capabilities of *ARTist* [11][3]. The rewriting part of *ARTist* is built on top of the `dex2oat` compiler of the Android Runtime (ART) introduced in Android 5 Lollipop and provides a modular framework to integrate own instrumentation solutions. We use *ARTist* to modify interactions with the advertisement library to interact with our *AdHelper* instead by utilizing two of *ARTist*'s main features: introducing own instrumentation routines through the *Module* framework and injecting our *AdHelper* into the host app through the library injection capabilities.

Module Framework. ARTist instrumentation is based on the concept of so called *Modules*. A *Module* gets full access to the application's code, allowing for arbitrary modifications, e.g., adding or removing instructions or changing them altogether, which will be reflected in the code after compilation. Internally, *ARTist* utilizes `dex2oat`'s optimization framework to disguise *Modules* as optimizations and let the existing infrastructure execute them. Concretely, a *Module* is then provided with the code of all methods in the compiler's internal intermediate representation (IR), one after another, and can analyze and change it at will, as the compiler believes it is executing a regular optimization algorithm. As it is designed to be utilized for optimization algorithms, the compiler's IR represents a method as a control flow graph of heavily interlinked nodes that closely resemble `dex` bytecode instructions [4].

We leverage this *Module* interface to implement the host side of *CompARTist*. More precisely, we introduce a specialized *Module* to take care of replacing the host-library interactions with corresponding versions from our *AdHelper*.

Library Injection. While the *Module* framework is designed to modify existing code, the injection capability allows to merge arbitrary own code libraries into a target application. *ARTist* automatically takes care of making all APIs from *AdHelper*, as well as other support components available to our module so that we can safely redirect all interactions to this new target.

5.3.4 Module Design. While *ARTist* only provides the integration into the compiler, the main challenge is to design the *CompARTist Module* to seamlessly connect the host application to the communication channel without harming the app's original semantics. Therefore, we will focus here on the design of our rewriting *Module*.

Collecting Instrumentation Targets. From our analysis we know the precise patterns that bootstrap interaction between host and advertisement library. From this point, we need to find all IR code nodes that operate on the obtained library data and modify them accordingly. Since each node in the IR method graph is interlinked with its usages and inputs already, we can apply forward slicing from our starting points to find all code nodes that we need to modify. Derived from our earlier analysis, we define three types of start nodes: class loading, field access and method invocation. As we are operating on method control flow graphs, we can find all those occurrences on a per-method base. In the IR graph, those starting points are marked by the following instructions:

1. `LoadClass` starts a host-lib interaction by loading an advertisement library class that is subsequentially used for, e.g., `InvokeStaticOrDirect` and `NewInstance` instructions.
2. `{Static,Instance}FieldGet` obtains previously-saved advertisement library data from a field in a host component.
3. `InvokeVirtual` receives previously-saved advertisement library data from an invoked host method.

Instrumentation Policies. Equipped with a list of entry nodes, we follow the slice through the method graph and collect every instruction that interacts with the advertisement library. Afterwards,

[3] ARTist is open source software available under Apache 2.0 license (https://github.com/Project-ARTist).

[4] The *ARTist* paper [11] provides in-depth documentation on the intermediate representation

each single node is transformed to use our generic communication channel instead. This is possible since the IR graph provides us with all the structural information required to properly interact with the *AdHelper* API: operation type, operation target and, optionally, parameters. While we learn the operation type from the concrete IR node (e.g., instance creation for `NewInstance` nodes), operation target and parameters are immediately available in the graph, too, and can therefore be provided to the *AdHelper* API.

Example Transformation. Figure 6 describes the code transformation applied to a code snippet that creates and loads a Google Play Service Ads advertisement. The bottom left part of the Figure 6 depicts the intermediate representation of a small method that loads an `AdView`. After loading the advertisement library class (instruction 5), the result of the `LoadClass` node is used to create a new object (instruction 7 and 8). Afterwards, the newly created `Builder` is used to build an `AdRequest` (instruction 9) that is consequently used to load an advertisement (instruction 11).

Starting from the `LoadClass` node, forward slicing provides us with all of the above mentioned nodes that interact with library components. The right part of Figure 6 depicts the transformed version of the advertisement loading code. First, instead of loading and instantiating the original class, the instrumented version uses the `createObjectHelper` method from our *AdHelper* to trigger the instantiation of said object in the remote library (instruction 22). Second, the `invokeMethodHelper` allows to trigger the invoked `build` method remotely (instruction 25), it only requires the name (instruction 24) and class (instruction 21) strings, and the object handle returned from `createObjectHelper` (instruction 22) to be provided as arguments. Third, the `loadAd` is remotely invoked via the `invokeMethodHelper` API (instruction 20).

5.4 Advertisement Service App

The advertisement service app encapsulates the ad library and forms the sandbox for the lib. As a separate app, executed with a distinct UID and in separate process, it effectively privilege-separates the ad lib with a strong security boundary. Additionally, this app is responsible for executing operations requested by the host app on the library or for proxying callback methods from the library to the host app (as explained in Section 5.2). Moreover, it is responsible for displaying the advertisement on screen at the correct position to preserve visual fidelity. To correctly display ads, the *AdService* relies on lifecycle synchronization messages from the host app, e.g., show/hide an advertisement or rotate the advertisement.

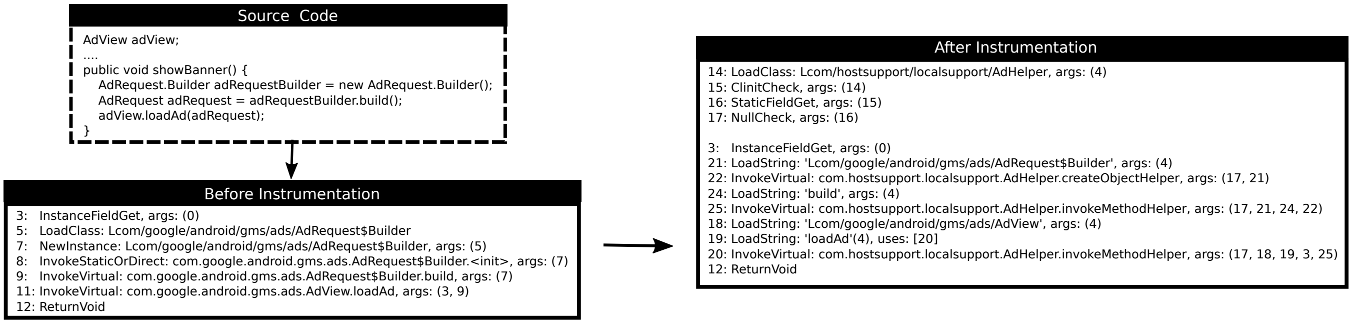
5.4.1 Synchronizing lifecycles and preserving visual fidelity. It is important to preserve the original look-and-feel of the ad library (*visual fidelity*) by serving the advertisement as a part of the host application's user interface. In particular, sharing a screen with the host application is very prevalent in advertisement libraries and therefore needs careful consideration. Most advertisements are directly integrated into the layouts of their host activities and therefore share their lifecycle, such as creation, pausing, and finishing events. Thus, in *CompARTist* we need a mechanism to keep them in sync between the host app and the separately executing ad lib in the ad service app.

Proxy view and floating window. Instead of simply removing the original advertisement `View`, e.g., `AdView`, from the layout of the host, we replace it with a carefully crafted and empty proxy `View`. In order to preserve the dimensions and placement of the remaining GUI elements, this proxy `View` is located at the exact same position as the original ad `View` and occupies the exactly same space. Concurrently, ad service app creates a floating window that is placed on top of the proxy `View`, again occupying the very same position and space as the original ad `View`. It is important to note that the floating window, even though originating from the ad service app, can still be displayed while the host app is running in the foreground. Hence, the floating window effectively covers the same area on screen as the proxy `View` (see Figure 7). In our solution, we use floating window type `TYPE_TOAST` to overlay the proxy space with no additional permission needed. Whenever a lifecycle callback from the Android system arrives at the proxy `View`, such as rotation events between portrait and landscape orientation or create/pause/resume/destroy events, the proxy `View` forwards them via our inter-application communication channel and *AdService* to the floating window. This allows the floating window to stay in sync with the host app's proxy `View`. As a result, while the advertisement is safely compartmentalized in the service app, the user perceives the advertisement as a part of the host app's layout because the occupied space and the lifecycles are synchronized.

The required layout information and lifecycle events are gathered through two user interface callbacks: `OnLayoutChangeListener` and `ActivityLifecycleCallbacks`. Since the proxy `View` is integrated into the host layout and instantiated in the host app's context, it obtains the exact position the advertisement should have on-screen through implementation of the `OnLayoutChangeListener` and synchronizes this information with the remote side. By implementing `ActivityLifecycleCallbacks` for the proxy `View`, it is also straightforward to have synchronized displaying, hiding, and finishing events in the remote advertisement `View`.

Advertisement view inflation. Usually, an advertisement view can either be defined explicitly in a layout file and inflated automatically by the system, or it can be instantiated manually at runtime. While we can handle the runtime case with our rewriting framework, supporting view replacement in case the advertisement instantiation is done by the Android framework itself is more intricate. Modifying the layout file directly is a possible solution, but it would again require to repackage the app and break the app signature. To support view substitution in both cases, at runtime and via layout files while still maintaining the app signature, we use reflection to additionally hook into the inflation mechanism at runtime and inflate our proxy `View` instead of the original advertisement `View`. Using this approach, the layout integration technique in Table 1 can be supported.

5.4.2 Multiplexing host apps. There are two approaches to achieve advertisement pairing while multiplex host apps exist. One ad lib app per app approach, where library runs in its own remote app, can easily enforce per app privileges on the ad lib. This approach, however, is not resource efficient. A centralized advertisement app, which contains all advertisement libraries and serves all rewritten host apps would be more efficient. Since our inter-application communication channel between client and ad service app is built on



```
AdView adView;
....
public void showBanner() {
   AdRequest.Builder adRequestBuilder = new AdRequest.Builder();
   AdRequest adRequest = adRequestBuilder.build();
   adView.loadAd(adRequest);
}
```

```
3:   InstanceFieldGet, args: (0)
5:   LoadClass: Lcom/google/android/gms/ads/AdRequest$Builder
7:   NewInstance: Lcom/google/android/gms/ads/AdRequest$Builder, args: (5)
8:   InvokeStaticOrDirect: com.google.android.gms.ads.AdRequest$Builder.<init>, args: (7)
9:   InvokeVirtual: com.google.android.gms.ads.AdRequest$Builder.build, args: (7)
11: InvokeVirtual: com.google.android.gms.ads.AdView.loadAd, args: (3, 9)
12: ReturnVoid
```

```
14: LoadClass: Lcom/hostsupport/localsupport/AdHelper, args: (4)
15: ClinitCheck, args: (14)
16: StaticFieldGet, args: (15)
17: NullCheck, args: (16)

3:   InstanceFieldGet, args: (0)
21: LoadString: 'Lcom/google/android/gms/ads/AdRequest$Builder', args: (4)
22: InvokeVirtual: com.hostsupport.localsupport.AdHelper.createObjectHelper, args: (17, 21)
24: LoadString: 'build', args: (4)
25: InvokeVirtual: com.hostsupport.localsupport.AdHelper.invokeMethodHelper, args: (17, 21, 24, 22)
18: LoadString: 'Lcom/google/android/gms/ads/AdView', args: (4)
19: LoadString: 'loadAd'(4), uses: [20]
20: InvokeVirtual: com.hostsupport.localsupport.AdHelper.invokeMethodHelper, args: (17, 18, 19, 3, 25)
12: ReturnVoid
```

Figure 6: Intermediate representation of advertisement loading code before and after the *CompARTist* transformation.

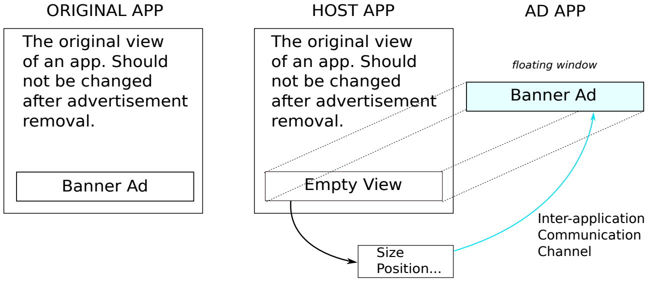

ORIGINAL APP
HOST APP
AD APP
The original view of an app. Should not be changed after advertisement removal.
Banner Ad
The original view of an app. Should not be changed after advertisement removal.
Empty View
floating window
Banner Ad
Size Position...
Inter-application Communication Channel

Figure 7: Synchronization Management

top of service connections using `Binder`, the ad service can identify the current caller app using `Binder.getCallingUid()` together with information provided by the `PackageManager`. By using those client-specific profiles, libraries can be shared between different clients. However, this approach requires a strong domain isolation within the single user-level advertisement app to privilege ad executions according to their host apps(similar to AdDroid's [27] ad system service). Each approach has its own merits and both of them can be adopted to *CompARTist*, since it's just a matter of redirecting the IPC calls.

To prevent a malicious host app from stealing ad revenue through our *CompARTist* by continuously sending synchronization messages that instruct *AdService* to overlay *any other* app with the malicious app's advertisement, the ad service app must be able to make synchronization events plausible. In our current solution, we rely on the simple heuristic that only the host app that is on top of the system's Activity stack, i.e., in foreground on screen (excluding the floating window overlay), is able to send valid synchronization events, since it essentially instructs the *AdService* to be overlayed or finish its own overlay, thus not affecting any other app. The information about the current top Activity can be retrieved by third party applications (like our ad service app) on older Android versions via the `ActivityManager` and on newer Android versions via the `UsageStatsManager`.

5.5 Deployment

Our current design is mainly focused on the idea of providing an application layer-only solution that completely abstains from firmware modification while still providing robust isolation. This in particular means that, as discussed in the beginning of Section 5, we tailored our solution towards fulfilling most of the goals outlined in Table 2. While our ad service app can be installed as a regular application, the deployment of the host-side instrumentation part of *CompARTist* is more intricate.

Requirements. As described in Section 5.3, our app rewriting solution is based on the *ARTist* framework that works on top of a modified compiler. However, with the above mentioned requirements in mind, we have to abstain from replacing the system compiler since this has several drawbacks: It requires modification of the firmware and, in addition, every app installed will automatically be instrumented by our approach. However, we want to allow for selective recompilation of apps, i.e., the user should be able to pick a subset of apps that she wants to have instrumented.

ArtistGui. We achieve this goal by utilizing the freely available ArtistGui[5] Android app that was created to provide a seamless way to make use of *ARTist Modules* from the application layer without requiring firmware modifications. It allows to ship the compiled version of our app modification logic as a binary asset and makes its instrumentation capabilities available through an easy-to-use graphical interface. After applying our modification routines, the instrumentation is completely transparent to the user as she can still start the application from the launcher or other apps as usual.

Dependencies. While the chosen approach abstains from modifying the Android operating system, it still requires at least elevated privileges to be able to convince Android to execute instrumented apps instead of the original ones. We will discuss this requirement and possible solutions as well as alternative deployment strategies for our rewriting part in Section 6.3.

6 DISCUSSION

We discuss our system in terms of the robustness of transformed apps, the performance overhead our changes induce, and its limitations. Further, we identify potential improvements and future research directions.

[5] ArtistGui is open source software available under Apache 2.0 license (https://github.com/Project-ARTist).

6.1 Robustness Evaluation

The applicability of our approach largely depends on its capability to neatly re-integrate the split application and advertisement code across process and sandbox borders. In order to show the robustness of our system, we conducted a large-scale evaluation on free apps from the Google Play Store that contain advertisements by testing them after applying our transformation.

6.1.1 Target Apps. We evaluate the robustness of our approach against a list of applications that contain the Google Play Service Ads library. As it dominates a large fraction of the mobile advertising market, evaluating with this library can indicate compatibility with a major fraction of the market share of apps incorporating advertisements. We started by creating a list of candidate apps from Google Play Store. For generating this list, we utilized the freely available LibScout project [9], which can tell apart the different libraries used in apps. Starting off with top apps from the Google Play Store that incorporate Google Play Service Ads, we filtered out apps that did not meet the prerequisites for testing, e.g., could not be downloaded, were published dysfunctional (i.e., crashed after installation), or are multidex[6].

6.1.2 Testing Setup. Scaling the evaluation of a dynamic approach to thousands of apps is only achievable through automation. For pre-filtering, compiling, and testing all target apps on real devices, we utilize `monkey-troop`[7], an app testing framework designed to evaluate *ARTist Modules*. After filtering, apps are installed on the device, transformed using *CompARTist*, and automatically tested using Android's *monkey tool* [1]. Experiments are conducted on Google Pixel C devices running rooted stock Android 7 Nougat, each having *CompARTist* and our ad service lib installed and configured.

6.1.3 Automated UI Testing. Achieving meaningful code coverage by using automated UI testing tools is still an open problem. We currently utilize Google's *monkey tool* [1] to apply random touch gestures to application activities. However, with this strategy the *monkey* rarely makes it beyond the first few activities, let alone those with input-validated fields. In addition, it can be prevented to start other activities, but it often leaves the app by returning to the homescreen or randomly changes quick settings. Still, we use the *monkey* in our evaluation for the following reasons: First, it provides reproducible executions since it provides us with the seed for its random generator. This allows for applying the same testing in the *filter* and *test* phase, respectively, to prevent mismatches there. Second, code coverage is not crucial here, since we already execute a lot of code at application start (cf. Section 5.2) and therefore starting the app to ensure it does not crash already suffices in most cases. So triggering at least some functionality is not considered mandatory, but a plus.

6.1.4 Results. We used the described testing infrastructure to test 3861 apps on real devices, out of which 325 apps were removed because they did not meet the above mentioned criteria for testing. Figure 8 shows the results of our large-scale robustness evaluation.

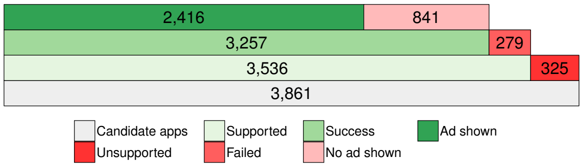


Figure 8: Breakdown of our robustness evaluation on applications using the Google Play Services Ads library.

Table 3: Performance evaluation results for the app compartmentalization transformation (averaged over 50 runs).

	Baseline (ms)	Transformed (ms)
Application Start	6.52	149.44
Banner	2025.35	2101.50
Interstitial (Loading)	1923.05	2084.44
Interstitial (Displaying)	117.13	125.40
Interstitial (Overall)	2040.18	2209.84

Out of 3536 apps, 3257 were checked, installed, tested, instrumented and retested successfully, yielding a success rate of 92.11% that indicates the robustness of our approach and the compatibility with a large fraction of the current ad-using app landscape.

The topmost row provides some more insight into the drawbacks of our current evaluation design. Although we successfully instrumented 92.11% of the tested applications, only 2416 out of 3257 (74.18%) reached a state that actually requested an advertisement during testing. As discussed in Section 6.1.3, the `monkey` is a limiting factor here since not all applications show advertisements in the first few `Activities` that are within reach. However, please note that even for those apps that did not request an advertisement, the application successfully established a connection to the remote advertisement library, which already involves heavy rewriting. Hence we expect a large fraction of those apps that did not reach ad request code during testing to nevertheless be compatible with our approach.

6.2 Performance Evaluation

We analyze the performance of our design by comparing the modified version of the app, which connects to and interacts with the remote advertisement library, to the unchanged version of the app under test. We apply microbenchmarking to analyze the three scenarios we are particularly interested in: *Application Start*, *Banner Advertisments* and *Interstitial Advertisements*.

6.2.1 Testing Setup. For our microbenchmarks, we again focus on apps utilizing Google Play Services advertisements. In order to measure the immediate impact of our modification, we create a sample application that integrates banner and interstitial ads according to Google's developer manual [7]. At the same time, we embed benchmarking code into the application itself, so that it can measure the precise time required to invoke certain functionality. With this approach, we can precisely compute the overhead by repeatedly invoking the app in its regular and also transformed state,

[6] Our implementation does not support multidex apps (apps packaging more than one dex file).

[7] monkey-troop is open source software available under Apache 2.0 license (https://github.com/Project-ARTist).

and compare the results. Experiments are conducted on Google Nexus 6 device running rooted stock Android 7 Nougat.

6.2.2 Results. Table 3 shows the performance overhead measured during our tests for the different scenarios.

Application Start. As explained in Section 5.2, we need a connection to our remote advertisement service from the very beginning, hence we block the application until we obtain the service handle. More precisely, we wait for a fixed amount of 100 ms before scanning the message queue to ensure the service Binder is available. After establishing a connection, the host side spends some time on client-specific initializations before returning, hence also blocking the app. The combined one-time overhead of service Binder scanning, IPC roundtrips, and client-specific initialization is depicted in Table 3.

Banner Advertisements. Since loading and displaying banner advertisements is a synchronous task, the microbenchmark starts when the advertisement is requested and ends as soon as the banner reports that it was successfully loaded, effectively providing an end-to-end measurement. For banners, our modifications to the application introduce an acceptable overhead of 3.62%.

Interstitial Advertisement. Due to the increased size of interstitials and in contrast to, e.g., banner advertisements, the developer documentation [6] suggests to preload the advertisement as early as possible to ensure it is available when the app decides to display it. The implications of this decoupling are twofold: First, only the loading phase involves network communication. After the advertisement has been downloaded, the displaying phase is completely independent of the network and can therefore provide reliable test results that are more likely to be reproducible. Second, if we assume that the majority of application developers using the Google Service Play Ads follow this advice, loading will be handled asynchronously in the background and therefore small deviations will not impede the user experience. Taking those implications into account, we decided to separately measure and report microbenchmark results for interstitial advertisement loading and displaying time. As depicted in Table 3, the overall measurement is dominated by the loading part. Since those results are heavily influenced by the network operations involved, we take the measured overhead of 7.74 % with a grain of salt. However, the measurements at least indicate that our instrumentation does not have a major impact on the loading performance. For the advertisement displaying benchmark, as expected, we can see a small overhead of 6.59 % added by our approach due to additional computations and IPC roundtrips, which is still within a range that is hardly perceivable by end users.

6.3 Deployment Alternatives

We discuss alternative deployment strategies in terms of their shortcomings and the specific use cases motivating them.

6.3.1 Host-side Alternatives. Depending on the concrete use cases, *CompARTist* can be retrofitted to achieve a different subset of the goals from Table 2. We present alternative implementation strategies that replace or combine the host part of *CompARTist* with existing work.

Instrumentation Frameworks. In terms of instrumentation capabilities, *ARTist* is on par with long-established Android app instrumentation frameworks, most prominently bytecode rewriting approaches. Hence, in case of the concrete use case does not require preservation of application signatures, the host-side rewriting can be fully implemented in one of the existing instrumentation frameworks [13, 17, 18, 24, 29, 37] without affecting the communication channel or library side.

Virtualization Techniques. Filesystem virtualization alone, as provided by existing virtualization solutions [10, 15], is not sufficient when it comes to retrofitting the host app to utilize our communication channel, because it essentially treats the application as a black box. While those solutions operate at application granularity and therefore only see the app interacting with the middleware or kernel, modifying applications to utilize our *AdHelper* needs instruction-granularity so we can distinguish between host and library code and rewrite interactions accordingly. Nevertheless, we can combine virtualization techniques either directly with *ARTist*, as suggested in the paper [11], or with one of the alternative approaches mentioned above, i.e., application rewriting frameworks. By using systems such as Boxify [10] or NJAS [15], we can therefore avoid the requirement for elevated privileges.

6.3.2 System-centric Deployment Strategy. Consider the use case of a custom ROM that ships a modified Android operating system. From the perspective of a ROM developer, application layer-only focus and preserving app signatures are of no concern anymore since *CompARTist* can be fully embedded into the firmware itself. Replacing Android's default `dex2oat` compiler with an *ARTist* version that runs our *CompARTist Module* already suffices because each application is automatically retrofitted to use the remote version of its advertisement libraries. This alternative deployment path might be beneficial for, e.g., security-focused ROMs employing a hardened version of Android.

6.4 Limitations

We discuss those limitations inherent to our approach, as well as those of our prototypical implementation.

6.4.1 Approach Limitations. As a result of design decisions during the creation of our system, we have to deal with some limitations that are inherent to our approach. While we created *CompARTist* with the idea in mind that we could compartmentalize arbitrary Android libraries, it might be infeasible to apply our approach to more strongly-coupled and deeply-integrated libraries such as, e.g., Guava [8]. In contrast to advertisement libraries that have a well-defined interface to the app and only communicate rarely, reconnecting a deeply-integrated library through IPC might require proxying a large number of classes, consequently raising the performance overhead significantly and possibly impairing user experience. While this paper presents a new and robust approach to sandbox libraries in general, it is more suited towards isolating loosely-coupled components, such as advertisement code.

6.4.2 Implementation Limitations. Beside limitations in our prototypical implementations, *CompARTist* also inherits implementation shortcomings of the *ARTist* system that is utilizes.

ARTist. Even though dex2oat is available since Android 5 Lollipop, *ARTist* utilizes the Android 6 version of the Optimizing backend, hence only later versions (currently Marshmallow and Nougat) can be supported. In particular, *CompARTist* is built on top of the Android 7 version of *ARTist*. A further downside resulting from utilizing *ARTist* is the requirement for root or in general elevated privileges. However, as discussed above, depending on the use case there are alternative deployment strategies with relaxed requirements available.

CompARTist. One shortcoming of *CompARTist* itself is that it works with a whitelist of supported advertisement libraries, hence it cannot support new libraries out of the box. Even though our current design allows for the fast creation of the required remote advertisement library package, it still requires an expert to explicitly add support for additional libraries. While the advertisement market is not strongly fragmented at the time of this writing, new libraries might pop up in the future and the above mentioned extra effort can be done and shared by, e.g., the community. Another missing feature is the support for multidex files, as already hinted at in the evaluation section. Without proper support for apps with multiple dex files, larger applications cannot be recompiled with our current prototype.

6.5 Future Work

We outline possible improvements to our existing prototype and indicate future research directions.

6.5.1 *Improvements.*

We list some possible improvements to *CompARTist* as well as for the evaluation pipeline.

Obfuscation Support. In order to replace existing intra-app ad library calls with calls to *AdHelper*, our *ARTist Module* scans the target application's code for the invocation of library methods. While obfuscation hides the real method name (that could be obtained from the library documentation), the structural information, such as inheritance and package structures, are still available. Hence, the robustness of our library call detection can be improved by incorporating techniques such as those suggested in [9].

UI Testing Automation. While our evaluation infrastructure takes measures to avoid a lot of common pitfalls in automated on-device testing, one of its weaknesses is Android's own *monkey tool* that is utilized to exercise the UI of applications. Even though *monkey* is sufficient to show the feasibility of our approach, the fact that touch events break out of the boundary of the application under test can result in undesired or even undefined behavior[8]. For example, we observed the interruption of our experiments triggered by the *monkey* disabling the usb debugging option in the developer setting or even going as far as factory-resetting the testing device. Those incidents clearly show that in order to provide a reliable test infrastructure, a superior UI exerciser tool is required. Possible candidates for replacing *monkey* could be DroidMate [23] or Brahmastra [14].

Library Detection. LibScout [9] has shown the problem of identifying libraries inside their host applications to be solvable with high probability. While we currently assume the advertisement library in the host application to be known beforehand, extending *CompARTist* with such a library detection would greatly improve its usability.

Callee-side Rewriting. The current implementation of *CompARTist* scans for invocations of advertisement library APIs and replaces them with Proxy methods from our support library. However, caller-side rewriting of method calls misses invocations triggered by reflection or from native code. Shifting our approach to callee-side rewriting, i.e., rewriting the call sites of APIs by replacing their logic with a redirection to our proxies, is a promising approach to solve this problem.

6.5.2 *Research Prospects.*

We do not only consider *CompARTist* to be a standalone tool but also a foundation for further interesting research projects.

Library Hotpatching. The predominance of well-established advertisement networks, such as Google Play Service Ads, results in lots of code duplication among applications since many ship the same statically-linked advertisement library. At the same time, updating the advertisement library is left as a task to the developers that, as related work has shown [9], are statistically speaking likely to delay those updates or in extreme cases omit them altogether. By utilizing *CompARTist*, it is possible to enforce dynamic linking of advertisement libraries by having exactly one adapted instance of each library (version) running in a dedicated application context and replacing statically-packaged libraries with references to the remote one. While we show in this paper that *CompARTist* is already capable of applying this transformation, a system-centric repository of advertisement libraries needs to be created and maintained. Such a system allows for centralized, system-wide updates of advertisement libraries that are completely transparent to the app developer, effectively taking them out of the loop. While library updates are not always backwards compatible, this system can be utilized to apply, e.g., security patches that do not change the public API.

Beyond Advertisements. We already discussed that compartmentalization of advertisement libraries is possible since they only use a well-defined set of techniques to integrate into host apps, which is not true for deeply wired libraries like Guava. However, there might be other types of libraries in-between that could be susceptible to our approach, including all the above mentioned opportunities such as the possibility for system-centric updates and the compartmentalization of untrusted code.

7 CONCLUSION

This work introduces *CompARTist*, a compiler-based library compartmentalization solution to remedy the unsatisfactory situation of privacy and security threats induced by advertisement libraries. Our solution splits the original app into host and advertisement library components and moves the library to a dedicated app to create a strong security barrier. We apply inter-process communication and lifecycle synchronization to seamlessly reintegrate both components without impairing user experience. Our evaluation proves the robustness of our approach by successfully applying our transformation routines to 3257 apps from Google Play Store. In

[8] In theory the *monkey* should be able to restrict touch events to activities of the app under test. However, this feature seems to be flawed.

conclusion, we introduce a new approach to library compartmentalization that abstains from system or app modifications.

ACKNOWLEDGMENT

This work was supported by the German Federal Ministry of Education and Research (BMBF) via funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345, 16KIS0656).

REFERENCES

[1] 2016. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html. (2016). Accessed: 2017-08-28.

[2] 2017. AdAway. https://adaway.org/. (2017). Accessed: 2017-08-28.

[3] 2017. AdblockBrowser. https://adblockbrowser.org/. (2017). Accessed: 2017-08-28.

[4] 2017. AdblockPlus. https://adblockplus.org/. (2017). Accessed: 2017-08-28.

[5] 2017. AdGuard. https://adguard.com/en/welcome.html. (2017). Accessed: 2017-08-28.

[6] 2017. Google Play Services: Interstitials. https://developers.google.com/mobile-ads-sdk/docs/dfp/android/interstitial. (2017). Accessed: 2017-08-28.

[7] 2017. Google Play Services: Setup. https://developers.google.com/android/guides/setup. (2017). Accessed: 2017-08-28.

[8] 2017. Guava. https://github.com/google/guava. (2017). Accessed: 2017-08-28.

[9] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *CCS'16*. ACM, 356–367.

[10] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android. In *USENIX Security'15*. 691–706.

[11] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. 2017. ARTist: The Android runtime instrumentation and security toolkit. In *EuroS&P'17*. IEEE, 481–495.

[12] Michael Backes, Sven Bugiel, Philipp von Styp-Rekowsky, and Marvin Wißfeld. 2017. Seamless In-App Ad Blocking on Stock Android. In *MoST'17*. IEEE.

[13] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. Appguard–enforcing user requirements on android apps. In *TACAS'13*. Springer, 543–548.

[14] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *USENIX Security'14*. 1021–1036.

[15] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. NJAS: Sandboxing unmodified applications in non-rooted devices running stock android. In *SPSM'15*. ACM, 27–38.

[16] Drew Davidson, Yaohui Chen, Franklin George, Long Lu, and Somesh Jha. 2017. Secure Integration of Web Content and Applications on Commodity Mobile Operating Systems. In *ASIACCS'17*. ACM, 652–665.

[17] Benjamin Davis and Hao Chen. 2013. RetroSkeleton: retrofitting android apps. In *MobiSys'13*. ACM, 181–192.

[18] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. 2012. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *MoST'12* 2012, 2 (2012), 17.

[19] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. 2016. Free for all! assessing user data exposure to advertising libraries on android. *NDSS'16* (2016).

[20] Erik Derr. 2017. https://projects.cispa.uni-saarland.de/derr/libscout. (2017). Accessed: 2017-08-28.

[21] Úlfar Erlingsson. 2003. *The inlined reference monitor approach to security policy enforcement*. Technical Report. Cornell University.

[22] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *WISEC'12*. ACM, 101–112.

[23] Konrad Jamrozik and Andreas Zeller. 2016. DroidMate: a robust and extensible test generator for Android. In *MOBILESoft'16*. IEEE, 293–294.

[24] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *SPSM'12*. ACM, 3–14.

[25] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys'15*. ACM, 89–103.

[26] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. 2016. The price of free: Privacy leakage in personalized mobile in-app ads. *NDSS'16*.

[27] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. Addroid: Privilege separation for applications and advertisers in android. In *ASIACCS'12*. ACM, 71–72.

[28] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS'14*, Vol. 14. 23–26.

[29] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. 2014. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *ARES'14*. IEEE, 40–49.

[30] Thorsten Schreiber. 2011. Android binder. http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf. (2011).

[31] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. 2016. FlexDroid: Enforcing in-app privilege separation in android. In *NDSS'16*.

[32] Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *USENIX Security'12*, Vol. 2012.

[33] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS'13*, Vol. 310. 20–38.

[34] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. 2016. What mobile ads know about mobile users. In *NDSS'16*.

[35] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. Investigating user privacy in android ad libraries. In *MoST'12*, Vol. 10.

[36] Mengtao Sun and Gang Tan. 2014. Nativeguard: Protecting android applications from third-party native libraries. In *WiSec'14*. ACM, 165–176.

[37] Rubin Xu, Hassen Saïdi, and Ross J Anderson. 2012. Aurasium: practical policy enforcement for android applications. In *USENIX Security'12*, Vol. 2012.

[38] Wenbo Yang, Juanru Li, Yuanyuan Zhang, Yong Li, Junliang Shu, and Dawu Gu. 2014. APKLancet: tumor payload diagnosis and purification for android applications. In *ASIACCS'14*. ACM, 483–494.

[39] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. Aframe: Isolating advertisements from mobile applications in android. In *ACSAC'13*. ACM, 9–18.

[40] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Hybrid user-level sandboxing of third-party android apps. In *ASIACCS'15*. ACM, 19–30.

APPENDIX

A APIS OF OUR COMMUNICATION CHANNEL

Listings of our communication channel API, noted in AIDL.

Listing 1: Callback API

```
void invokeListenerCallbackHelper(int objectId,
     String method);
void invokeListenerCallbackHelper_1(int objectId,
     String method, in WrapClass param);
void invokeListenerCallbackHelper_2(int objectId,
     String method, in WrapClass param_1, in
     WrapClass param_2);
void invokeListenerCallbackHelper_3(int objectId,
     String method, in WrapClass param_1, in
     WrapClass param_2, in WrapClass param_3);
void invokeListenerCallbackHelper_4(int objectId,
     String method, in WrapClass param_1, in
     WrapClass param_2, in WrapClass param_3, in
     WrapClass param_4);
```

Listing 2: Advertisement Invocation API

```
WrapClass getStaticFieldService(String ctype, String
     field);
WrapClass invokeStaticMethodService_2(String ctype,
     String method,in WrapClass[] params);
WrapClass invokeStaticMethodService(String ctype,
     String method);
WrapClass invokeVirtualMethodService_2(String ctype,
     String method, in WrapClass object, in
     WrapClass[] params);
WrapClass invokeVirtualMethodService(String ctype,
     String method, in WrapClass object);
WrapClass newInstanceService_2(String ctype, in
     WrapClass[] params);
WrapClass newInstanceService(String ctype);
```

Listing 3: Lifecycle API

```
void removeWindow(int viewId, boolean destroy);
void createWindow(int viewId, in Rect rect);
void updateWindow(int viewId, in Rect rect);
```