Stay in Your Cage! A Sound Sandbox for Third-Party Libraries on Android

Fabo Wang[1,2], Yuqing Zhang[1,2](✉), Kai Wang[2], Peng Liu[3], and Wenjie Wang[2,4]

[1] State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an, China
fbwang@stu.xidian.edu.cn

[2] National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing, China
{zhangyq,wangk}@nipc.org.cn

[3] College of Information Sciences and Technology, Pennsylvania State University, University Park, PA, USA

[4] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

Abstract. Third-party libraries are widely used in Android application development. While they extend functionality, third-party libraries are likely to pose a threat to users. Firstly, third-party libraries enjoy the same permissions as the applications; therefore libraries are over-privileged. Secondly, third-party libraries and applications share the same internal file space, so that applications' files are exposed to third-party libraries. To solve these problems, a considerable amount of effort has been made. Unfortunately, the requirement for a modified Android framework makes their methods impractical.

In this paper, a developer-friendly tool called LibCage is proposed, to prohibit permission abuse of third-party libraries and protect user privacy without modifying the Android framework or libraries' bytecode. At its core, LibCage builds a sandbox for each third-party library in order to ensure that each library is subject to a separate permission set assigned by developers. Moreover, each library is allocated an isolated file space and has no access to other space. Importantly, LibCage works on Java reflection as well as dynamic code execution, and can defeat several possible attacks. We test on real-world third-party libraries, and the results show that LibCage is capable of enforcing a flexible policy on third-party libraries at run time with a modest performance overhead.

1 Introduction

Android application developers are becoming increasingly dependent on third-party libraries, to simplify and speed up development or improve performance. However, when developers import third-party libraries, they often pay most attention to what abilities third-party libraries provide, regardless of their risks.

© Springer International Publishing Switzerland 2016
I. Askoxylakis et al. (Eds.): ESORICS 2016, Part I, LNCS 9878, pp. 458–476, 2016.
DOI: 10.1007/978-3-319-45744-4_23

Third-party libraries differ from system libraries in that it is difficult to guarantee their security.

Third-party libraries may pose a threat to users. First, as the permission system of Android is application-level, when an application is installed, both the host application and third-party libraries in it share the same permissions [1,2]. What's worse, some libraries may request more permissions than they need to complete their jobs [3]. This obviously violates the *Principle of Least Privilege* [4,5], which suggests that every program (module) should run with the least set of privileges necessary to complete a specific job. As a result, third-party libraries may abuse some permissions to collect user data [6,7]. Second, the host application and third-party libraries share the same internal storage, meaning that third-party libraries can operate the application's internal files, which may lead to a privacy leak.

Several approaches have been proposed to separate third-party libraries' permissions from the host application's permissions. Based on process isolation, some approaches [1,8–13] achieve permission separation by isolating the untrusted third-party library into another process. Compac [2] proposes an approach that enforces component-level access control on applications, while enabling developers to assign different permissions to each third-party library. FLEXDROID [14] provides in-app privilege separation for applications, with which developers can grant permissions to third-party libraries and also specify how to change their behavior after detecting a privacy violation. However, these pervious approaches require modifications to the Android framework or the applications (or libraries), which determines that they cannot be adopted widely. Aside from this, some of them are unable to solve the problem of libraries having the ability to steal the application's internal files.

Some work [15–17] has presented a sandbox based on application virtualization, to enforce a fine-grained permission control on applications at run time. However, these approaches cannot be applied to the case of third-party libraries since they target the whole application, and make no distinction between the host application and third-party libraries.

In this paper, we present LibCage, a friendly tool to help developers isolate third-party libraries from the host application in terms of permission and file space. Within the context of the host application, LibCage builds a sandbox for each library. First, each sandbox is assigned to a permission set by developers, so its library cannot perform privileged operations beyond these permissions. Our prototype focuses primarily on the dangerous permissions, which may carry a risk to user data. In fact, LibCage can be extended to support all permissions. Second, each sandbox has its own file space, and its library cannot operate files beyond this space. LibCage doesn't modify the Android framework or libraries, and thus it can be deployed widely. It also works on some unusual situations, such as dynamic loading or Java reflection. It doesn't require developers to modify the existing code or change the way in which they use third-party libraries. All they need to do is integrate LibCage into their projects, create some configurations and initialize LibCage. To the best of our knowledge, LibCage is the first solution to introduce a fine-grained sandbox on third-party libraries.

The contributions of this paper can be summarized as follows:

1. We present a novel approach to isolating each third-party library in an independent sandbox, ensuring that each library has a separate permission set and an isolated file space. Our approach covers Java and native libraries while eliminating the necessity of modifying the Android framework or libraries' bytecode.
2. We implement the prototype named LibCage, using which developers can assign a different permission set to each third-party library, and allocate each library an isolated internal file space.
3. We systematically evaluate the generality, effectiveness and performance overhead of LibCage. Results show that LibCage can effectively enforce a flexible security policy on any third-party libraries while introducing an acceptable performance overhead.

2 Android Security and Related Work

2.1 Android Security

Android provides several mechanisms to ensure the security of user data and device resources. We describe several main mechanisms as follows.

Application Sandboxing. Android adopts the Linux sandbox mechanism. When an application is installed, Android will give it an unique Linux user ID (UID). Each application runs in its own isolated process and doesn't have the privilege to interact with other processes directly (as can happen through IPC mechanism).

Permission System. By default, an application doesn't have any permissions to do privileged things, e.g., getting the location information. In order to be privileged, an application must explicitly declare the related permissions. On devices running Android 5.1 or lower, an application asks for users' approval at installation. Users either grant all the permissions to it or cancel the installation. Once an application is installed, it can use the permissions at any time and users can not revoke any permissions unless they uninstall it, which is called All-or-Nothing. To remedy this situation, Android 6.0 introduces dynamic permission assignment, which demands that an application should request permissions at run time and allows users to revoke any permissions at any time.

File Access Control. Android devices have two file storage areas, i.e., internal and external storage. External storage is world-readable, which means that all applications have access to files in this storage. By contrast, internal storage is private. When an application is installed, the system will assign an internal storage to it and this internal storage is accessible only by this application.

Operating files in the external storage requires the permission `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`, while operating files in the internal storage requires no permissions.

It is worth mentioning that under the current permission model, permissions are assigned at the application level, so the incorporated libraries and the host application have exactly the same permissions. It is impossible to grant permissions to only part of an application. Moreover, the libraries and the host application share the same internal file space, and libraries' access to internal files is out of the system's control.

2.2 Related Work

In this section, we describe related work in two categories: permission separation for third-party libraries and fine-grained sandboxing for applications.

Permission Separation for Third-Party Libraries. To separate the permissions of third-party libraries from the host application's permissions, several approaches have been proposed. AdSplit [8] and AFrame [9] isolate an advertisement library into another `Activity` (also in a different process) by extending the Android framework. In order to display both the advertisement and the host application, AdSplit allows two activities to share the screen, while AFrame supports embedded activities. Leveraging bytecode rewriting, Dr. Android and Mr. Hide [10] moves the untrusted code from the host application into another application. Targeting advertisement libraries, AdDroid [1] integrates advertisement libraries into the Android framework, introducing new advertising API and corresponding permissions. SanAdBox [12] diverts advertisement libraries into standalone applications with separated permissions.

Focusing on native libraries, NativeGuard [11] automatically repackages the target application and divides it in two: the client application, which holds the Java code and the resource; and the service application, which takes the native libraries. NativeProtector [13] isolates native libraries in a similar way to NativeGuard, and further performs fine-grained control of native libraries by instrumenting them and intercepting their calls.

These approaches achieve permission separation based on process isolation. They also provide storage isolation, ensuring the isolation of third-party libraries' file space. By contrast, Compac [2] enforces component-level access control on applications by extending Android's permission model. Developers and users can use it to assign different permissions to each component (e.g., third-party library). Based on hardware fault isolation, FLEXDROID [14] extends Android's permission system to provide in-app privilege separation for applications. With FLEXDROID, developers can grant permissions to third-party libraries and also can specify how to control their behavior after detecting a privacy violation. However, Compac and FLEXDROID don't provide storage isolation. Additionally, these work (a comparison can be seen in Table 1) require either modifications to the Android framework or the applications' (or libraries') bytecode, and thus they cannot be employed widely.

Table 1. A comparison of some related work

	Target	No framework modification	No bytecode rewriting	Permission separation	Storage isolation
AdSplit	Ad lib	✗	✓	✓	✓
AFrame	Ad lib	✗	✓	✓	✓
Dr. Android and Mr. Hide	All libs	✓	✗	✓	✓
AdDroid	Ad lib	✗	✓	✓	✓
SanAdBox	Ad lib	✓	✗	✓	✓
NativeGuard	Native lib	✓	✗	✓	✓
NativeProtector	Native lib	✓	✗	✓	✓
Compac	Java lib	✗	✓	✓	✗
FLEXDROID	All libs	✗	✓	✓	✗
LibCage	All libs	✓	✓	✓	✓

Fine-Grained Sandboxing for Applications. Several work has proposed a fine-grained sandbox to constrain the run-time permissions or behaviors of applications without modifications to the Android framework. Based on application virtualization and process-based privilege separation, Boxify [15] runs untrusted applications in a de-privileged, isolated process (leveraging the `isolatedProcess`[1] feature). AppCage [16] confines the run-time behaviors of applications by interposing and regulating an application's access to sensitive APIs. Specifically, AppCage builds a dex sandbox by hooking into the application's instance of Dalvik Virtual Machine to confine the access to sensitive framework APIs, and native sandbox leveraging software fault isolation (SFI) [19] to ensure that native libraries cannot escape the sandbox. NJAS [17] loads and executes the code of a given application within the context of a monitoring application and achieves sandbox by means of system call interposition (using `ptrace` mechanism).

However, these sandboxes cannot be applied to confining the permissions of third-party libraries, because they target the entire application without making any distinction between behaviors of the host application and of third-party libraries. Apart from this, they are unable to control file access.

3 Design

3.1 Objectives

Third-party libraries are not sufficiently trustworthy, though they enable some features and abilities developers desire. In this paper, we focus on the following threats introduced by untrusted third-party libraries:

[1] `IsolatedProcess` is an attribute of a service. If set to true, the service will run under a special process that is isolated from the rest of the system and has no permissions [18].

T1. Permission Sharing. As discussed in Sect. 2.1, at run time, the incorporated third-party libraries and the host application share the exact same permissions. If a library is malicious, then it may abuse the permissions, and steal private information. Some surveys [14] have shown that even benign libraries may use the host application's permissions without documenting them.

T2. File Exposure. Third-party libraries and the host application occupy the same internal file space, so that all the internal files are exposed to third-party libraries. This may result in untrusted libraries grabbing the application's files, which store the user data.

In this paper, we aim to provide a tool for helping developers eliminate the threats above. We identify the following objectives:

O1. Permission Separation. Developers can assign permissions to each third-party library, so that the library will not enjoy all of the host application's permissions.

O2. File Space Isolation. Each library is allocated an isolated internal file space and has no access to other file space (e.g., the host application's internal space).

O3. No Framework or Library Modification. For greater usability, our approach does not rely on any modifications to the Android framework or third-party libraries.

O4. Least Manual Effort for Developers. Our approach only requires minimal manual effort on the part of developers, such as permission configuration.

3.2 Approach

We isolate each third-party library by creating a sandbox for it within the host application's context (depicted in Fig. 1). The key insight is that we add a permission checker in an application, whose job is to validate a library's permissions when this library tries to perform a sensitive operation (sensitive operations are those related to collecting user privacy, such as accessing the contact data). In addition, checker is responsible for judging whether a library's access to a file is legal. If we detect that a library tries to perform an over-privileged operation (i.e., an operation beyond its permissions) or access files not belonging to it, we will block this operation. And thus we can confine the libraries' run-time permissions and control their file access.

The main challenge is *how to monitor third-party libraries' behaviors*. Based on system call interception, we interpose their invocations to sensitive methods or functions (In this paper, methods mean the framework APIs and functions refer to the system C native functions) and redirect these invocations to our monitoring proxies. In this way, we can monitor all sensitive invocations of libraries. Since the host application, libraries and our tool are in the same process, interception doesn't rely on modification to the Android framework or the code of these libraries (O3: ✓).

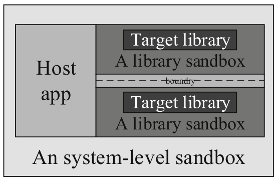

Fig. 1. Our library sandbox

The next challenge we encounter is *how to safely block an operation of a third-party library*. In fact, blocking an illegal operation directly will cause unexpected results (e.g., crash of the library, or even the entire application). In our system, we block an illegal operation in the following way. We first classify all sensitive methods and functions into two categories by the return type (void or not). For those methods or functions without a return result, the proxies simply return, so that the original invocations cannot proceed. For those methods or functions with a return result, the proxies return a mock value to avoid a crash. A mock value will not result in a privacy leak. Thus we can prohibit a permission by blocking the corresponding operation (O1: ✓).

Since external file access is controlled by permissions related to file operations, we only consider the internal file access here. In order to isolate third-party libraries' internal file space, we allocate each library an internal space under the path of the application's internal space. If this library tries to create a new file in the application's internal space, we will change the file's path to its own space. Therefore all files that belong to a library reside in this library's internal space. If the library attempts to access files beyond its space, we deny the attempt (O2: ✓).

Having described our idea, we now present the architecture of our tool named LibCage at a high level (depicted in Fig. 2). LibCage is composed of four entities: Interceptor, Controller, Checker and `Policy.xml` file.

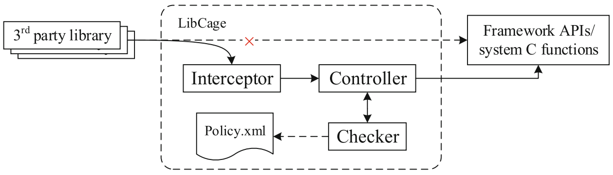

Fig. 2. The architecture of LibCage

Interceptor is responsible for intercepting the calls to the Android framework APIs or system C functions and redirecting them to Controller. For greater efficiency, the interception will be done automatically once an application starts. Controller handles the redirected calls and acts as a mandatory proxy between third-party libraries and the framework APIs or system C functions. Controller queries Checker as to whether a call is legal. If a call is permitted, Controller will call back the original method or function. Otherwise, Controller blocks this call.

Checker's job is to judge whether a third-party library has the permission to perform related operations according to the `Policy.xml` file. To do this, it maintains a mapping between methods or functions and permissions required by these methods or functions. It also determines whether an internal file access is legal.

`Policy.xml` is a configuration file under the assets folder, describing the dangerous permissions an application declares and the permissions assigned to each third-party library. Developers can set flexible security policies on third-party libraries by configuring this file.

Libcage doesn't require much manual effort; all developers need to do is incorporate LibCage into their projects, configure the `Policy.xml`, and add some code to initialize LibCage. When an application starts, LibCage will automatically sandbox each library used in this application (O4: ✓).

In reality, a third-party library may contain Java code and native code (i.e., using JNI). In fact, LibCage works on this library as well because regardless of how Java code interacts with native code, both of them are under the control of LibCage (They cannot directly call the framework APIs or system C functions). If a library consists of Java code and native code, we assign the same permissions to them, and allocate them the same internal file space.

3.3 Policy

LibCage ensures that a third-party library runs in an independent sandbox with only the permissions developers grant, while other over-privileged operations are denied. We are interested primarily in the dangerous permissions [20] (listed in Table 3 in the Appendix), since they may carry a risk to user data. For example, the `READ_CONTACTS` permission allows an application to read contacts, which could be abused by a malicious third-party library, resulting in a leak of user privacy. In fact, our approach can be easily extended to support all permissions; it simply requires more engineering effort.

As described above, LibCage relies on system call interception, including calls to Java methods and native functions. We intercept all methods or functions related to the dangerous permissions. Similar to [21], we identify the Java methods and the permissions required by these methods. For example, the method `getDeviceId()` requires the permission `READ_PHONE_STATE`. We also dig out the native methods related to the dangerous permissions, such as functions used to operate files[2] and *ioctl* function, through which an IPC is sent [22].

LibCage can be used to enforce a flexible policy on third-party libraries. When using LibCage, developers need to list the dangerous permissions, specify the information of third-party libraries, and assign a separate permission set to

[2] Technically, operating an internal file doesn't require any permissions. According to the file path, LibCage distinguishes whether the file being operated is internal.

```
<LibCage>
   <!--the dangerous permissions declared by your application -->
   <permission name="READ_CONTACTS" />
   <permission name="ACCESS_COARSE_LOCATION" />
   <!--for each library used, create a tag lib-->
   <lib>
      <!--specify the library information-->
      <pkgname="lib.package.name"/>
      <soname="nativelib.so"/>
      <!--the dangerous permission assigned to this library-->
      <permission name="ACCESS_COARSE_LOCATION" />
   </lib>
</LibCage>
```

Fig. 3. An example of configuring the `Policy.xml` file

each library. This is done by configuring the `Policy.xml` file[3]. Figure 3 illustrates the process.

In this example, the application has the `READ_CONTACTS` and `ACCESS_COARSE_LOCATION` permissions. We want to use a third-party library, which consists of a Java library[4] whose package name is `lib.pacakge.name` and a native library whose name is `nativelib.so`. We have assigned the `ACCESS_COARSE_LOCATION` permission to this library. At run time, this library (both the Java part and native part) can access the location information, but won't be able to read users' contact data. Moreover, an isolated internal file space is allocated to this library, and the library only has access to that space.

4 Implementation

4.1 Interceptor

Interceptor's job is to interpose third-party libraries' calls to target methods or functions and dispatch them to Controller. This takes place once an application starts (before any third-party libraries call target methods or functions). We implement the interception on both the Java library and native library.

Java Library. Interception is implemented by manipulating the internal data structures of target methods. At present, the Android system has two different runtimes: Dalvik on previous versions and Android Runtime (ART) which is introduced in Android 4.4 and set default after 5.0 [23]. Since the implementation on ART and Dalvik is similar, here we only describe the implementation on ART. ART maintains a data structure called `Class` (a C++ data struct) and a structure called `ArtMethod` for each method declared in this class. When a method is called, ART searches for the `Class` related to the class in which

[3] In our system, a permission name is shortened by removing the prefix. For example, the full name of `READ_SMS` is `android.permission.READ_SMS`.

[4] A Java library may include several packages or native libraries; for a package or native library, developers should add a tag.

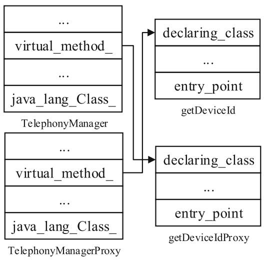

Fig. 4. Intercepting `getDeviceId()` method on ART

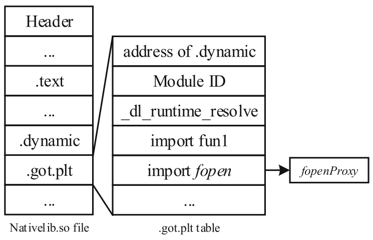

Fig. 5. Intercepting *fopen* function on the native library

this method is declared, and then the corresponding `ArtMethod`, which will be executed. By manipulating the data structure of a target method, LibCage interposes all calls to this method and redirects them to Controller.

Figure 4 shows an example of intercepting the `getDeviceId()` method, which provides access to the unique device ID (which can be used to identify a user). To interpose this method, Interceptor obtains the pointer of the related `ArtMethod` in the `Class` of `TelephonyManager` class and replaces it with the pointer of `getDeviceIdProxy()` method. As a result, when a library tries to call `getDeviceId()` method, this invocation is actually redirected to `getDeviceIdProxy()` method.

Once we manipulate a target method, all calls to this method in the same process will be interposed, which means that the interception is once for all (we don't need to deal with the libraries one by one). Since that Java code can't directly modify the process memory, we implement the interposition in C++ code and compile them to a native library.

Native Library. Similar to the traditional Linux model, Android native libraries are relocatable ELF (Executable and Linkable Format) files, which will be mapped into the process's address when loaded [22]. For the sake of memory and code size, these libraries adopt the dynamic linking mechanism. If a library imports an external function, the .got.plt table will contain a respective stub. This imported function will be resolved when the library is loaded and its address will be saved in its stub in the .got.plt table. When an imported function is called, the system first retrieves its address from its stub in the .got.plt table and then executes the code. This indirection can be exploited to implement interception neatly. It's sufficient to replace the stub of an external function in the .got.plt table with the pointer to our monitoring function [24], so that calls to this function will be intercepted and redirected to our function.

Figure 5 shows an example of intercepting *fopen* function, which is used to open a file. To intercept the calls to *fopen* function, LibCage replaces its address in .got.plt table with the address of *fopenProxy* function, a proxy function defined in Controller. As a consequence, when Nativelib.so tries to call *fopen* function, this invocation is actually redirected to the *fopenProxy* function.

Modifying the .got.plt table of one native library differs from interception on Java libraries in that it will not affect other native libraries. In order to interpose all untrusted native libraries' sensitive calls, LibCage manipulates their .got.plt tables one by one.

4.2 Controller

Controller is responsible for taking over the redirected calls and handling the results of these calls. Thanks to Interceptor, all sensitive calls are forwarded to Controller, which is a mandatory layer between third-party libraries and framework APIs or system C functions. To correctly handle the redirected call from Interceptor, Controller contains a proxy method or function for each target method or function.

Java Library. Usually, one Java method call carries two important pieces of information: the receiver object, representing which object calls this method (except for the `static` method), and the passed parameters. After obtaining this information from a redirected call, Controller queries Checker whether this call should be permitted or denied. If this call is permitted, Controller calls back the original method with the receiver object and the passed parameters. Otherwise, Controller blocks this call.

As previously mentioned, all libraries' calls to a target method are intercepted. We first identify the caller by analyzing the stack trace. If it is the host application, Controller calls back the original method immediately. Otherwise, it's a third-party Java library. Controller gets this library's package name and further queries Checker what to do next.

Native Library. Similar to how Controller works on Java libraries, Controller first gets the passed parameters of the redirected calls (native functions don't have the concept of receiver object). Then it identifies who the caller is by analyzing the back trace. If it's the host application, Controller calls back the original function immediately. Otherwise, it's a native library, and Controller decides whether to call back the original function or to block this call by asking Checker.

4.3 Checker

Checker judges whether a call from a third-party library to a method or function is legal or not based on the permissions assigned to this library by developers. Developers grant permissions to each third-party library by configuring the `Policy.xml` file in the assets folder (A demonstration can be seen in Fig. 3).

Checker maintains a Java HashMap, of which the key is the name of a method or function, and the value is the permission required by the method or function. For example, `getDeviceId()` method maps to the `READ_PHONE_STATE` permission. When Controller queries whether a call is legal, Checker searches for the permission set assigned to this library by parsing the `Policy.xml` file, and judges whether the permission the call requires is contained in the permission set. If it is, the operation is defined as legal.

When Checker decides whether a file access is permitted, it first identifies whether the file being operated is in internal storage, according to the file path. If this file is located in external storage, Checker judges whether the library is assigned the right permission. If so, this file operation is legal. If the file is located in internal storage, Checker judges whether it is in the isolated file space allocated to the library. If so, this file operation is legal.

5 Evaluation

In this section, we describe how we evaluate LibCage in terms of generality, effectiveness, and the performance overhead it brings. These experiments are performed on Samsung Galaxy Note 5, running 5.1.1, and MI 3, running 4.4.4.

5.1 Generality

First, we tested the generality of LibCage on various Android versions. Different Android versions may have different DVM or ART internal data structure. For example, the `ArtMethod` structure in Android 5.1 differs from that in Android 5.0. To adapt to the changes of DVM (or ART) in different versions, LibCage maintains different data structures for different versions. We tested on 4.0, 4.1, 4.2, 4.3, 4.4, 5.0, 5.1, 6.0, and the results showed that LibCage works well on these versions.

Second, we assert that LibCage can be applied to an arbitrary library. LibCage covers both Java libraries and native libraries. However, there are thousands of libraries in the real world, some of which may be programmed in an unusual way, such as dynamic loading or Java reflection. We describe how LibCage deals with these situations.

Dynamic Loading. Some Java libraries may use dynamic loading to execute external DEX (Dalvik executable format) files at run time. LibCage works in this situation too, because the loaded code still needs to call the sensitive APIs which are interposed by LibCage.

Java Reflection. Some Java libraries may use Java reflection to hide the actual API they are calling. To deal with this situation, LibCage intercepts the key method `invoke`, which is used in reflection to call a method. By analyzing the redirected call to `invoke` method, LibCage obtains the actual calling method. If LibCage determines that this call is over-privileged, it will block the call.

We have tested these unusual situations, and the results were successful.

5.2 Effectiveness

In order to evaluate the effectiveness of our prototype, we developed several libraries, including Java libraries and native libraries, performing some sensitive operations related to the dangerous permissions (listed in Table 3 in the

Appendix) and some file operations. The test application also declared all the permissions needed to perform these operations. We then defined a set of security policies and used LibCage to enforce these policies on the test libraries. Results showed that LibCage can limit the run-time permissions of third-party libraries and also showed that LibCage can forbid a library's attempt to access files beyond its file space.

To better demonstrate the results, we tested LibCage on third-party libraries from real world. First, we downloaded 250 applications from the "Top free in Android Apps" chart in Google Play [25], regardless of their categories. We then decoded these APK files and collected the top 10 third-party libraries (depicted in Fig. 6). The detailed permissions they require are listed in Table 4 in the Appendix. Some of these libraries require two levels of permissions: mandatory permissions (developers must list these) and optional permissions (used to provide more functionality; developers can list these optionally).

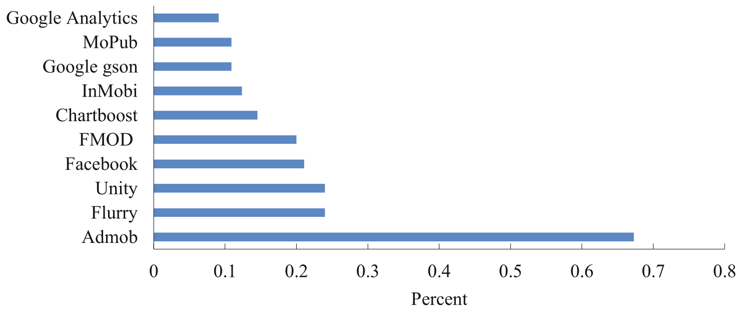

Fig. 6. Popularity of the top 10 libraries in our dataset

We developed a test application with these libraries and execute it manually. First, we enforced a normal policy on each library, that is, we granted all the required permissions to these libraries while the application's other permissions are prohibited. The results were exciting. LibCage forces these libraries to run with only their own permissions, without rendering them dysfunctional.

Second, we enforced a more rigid policy on the libraries. Note that some of them require optional permissions. For example, Flurry and InMobi require the `ACCESS_FINE_LOCATION` permission, which can be used to track a user. We tested these libraries without granting them optional permissions, though the host application still listed those permissions (If the host application doesn't list these optional permissions, these libraries obviously cannot use them. In this test, we checked whether LibCage could cut off the libraries' optional permissions even when they were available), and the results also showed that these libraries worked well.

These tests showed that LibCage can confine the run-time permissions of third-party libraries and can also cut off unnecessary permissions.

5.3 Performance Overhead

We also evaluated the run-time performance overhead introduce by LibCage. We measured the time required to complete a method or function invocation with and without LibCage. For Java libraries, we tested on these operations in relation to the dangerous permissions (see Table 3 in the Appendix), including querying contacts, getting location, and getting device ID. For native libraries, we tested on some functions used to operate files. The results are shown in Table 2.

Table 2. Overhead introduced by LibCage

	Operation	Without LibCage	With LibCage	Overhead
Java method	Read SMS (100 runs)	2539 ms	2720 ms	7.14 %
	Get location (1 k runs)	949 ms	981 ms	3.40 %
	Get device ID (1 k runs)	397 ms	423 ms	6.59 %
	Query contacts (100 runs)	2885 ms	3076 ms	6.63 %
	Delete a file (1 k runs)	2539 ms	2720 ms	7.29 %
	Open camera (10 runs)	1292 ms	1332 ms	3.11 %
	Record an audio (10 runs)	2374 ms	2395 ms	0.88 %
Native function	Open a file (10 runs)	1333 ms	1383 ms	3.75 %
	Edit a file (200 runs)	119 ms	125 ms	5.33 %
	Read a file (10 runs)	103 ms	110 ms	7.13 %

Intercepting calls to Java method or native function imposes an overhead less than 10 %, among which the highest is 7.15 % and the lowest is 0.88 %. For a single method or function invocation, the overhead is less than 4 ms, which is so negligible that users can't perceive it. We believe that this overhead is acceptable.

5.4 Security Analysis

In this section, we explore some possible attacks to bypass or subvert LibCage and describe how we defend against them.

Manual Resolution. A native library can link the system C libs manually, get the pointer of the target function, and call the function through this pointer. In this case, the .got.plt table of the library doesn't contain the stub of this function, so LibCage becomes invalid. To remedy this situation, LibCage interposes the *dlsym* function, which is used to resolve (get the address of) a function. Then LibCage can know which function is being resolved. LibCage will return the address of the proxy function if this function is sensitive, and thus the call to this function is interposed. Then the operation can continue as usual.

Privilege Escalation. LibCage judges whether a library's behavior is over-privileged according to the `Policy.xml` file. One concern is that a library may tamper with this file, and then its privilege can be escalated up to the application's privilege (that is, it may access all permissions). In fact, the `Policy.xml` file cannot be modified by any libraries at run time, meaning that no libraries can bypass the permission check, since files in the assets folder are read-only.

Modify Mapped Bytecode. A library may change the loaded code by writing the mapped bytecode. LibCage foils this attack, since the functions used to manipulate memory (e.g., *mmap* function, used to map files into memory and *mprotect* function, used to specify the protection level for a memory page) are interposed and this attempt is denied.

6 Discussions

In our work, we focus on dangerous permissions, since our goal is to protect user privacy. In fact, LibCage can be easily extended to support all permissions. It's just a matter of engineering efforts. Our approach can limit the run-time permissions of third-party libraries and cut off their additional permissions which are unnecessary to complete their jobs. However, developers still need to list all permissions required by a third-party library in the Manifest file. Our next objective is to eliminate this need.

Moreover, the crash of a third-party library will result in the breakdown of the entire application, which is undesirable for both developers and users. [19] proposed software fault isolation (SFI) to make sure that faults in one module cannot render a software system unreliable. Based on SFI, several work [26,27] has been done to sandbox untrusted native libraries. In our future work, we will adopt the idea of software fault isolation to ensure that crashes of third-party libraries cannot influence the availability of the host application.

7 Conclusion

The security of third-party libraries has attracted much attention because of their prevalence in recent years. In this paper, we proposed a novel sandbox for third-party libraries, which ensures that each library runs in a respective sandbox. Each sandbox is assigned to a separate permission set by developers and its library cannot carry out privileges beyond this permission set. In particular, developers can cut off the unnecessary permissions of a third-party library. Further, each sandbox is allocated an isolated file space while its library has no access to any other space. We have implemented this approach in a tool called LibCage and evaluate it in terms of generality, effectiveness and performance. The results show that LibCage can be adopted to any library, its effectiveness is remarkable, and the performance overhead it introduces is negligible.

Acknowledgments. This work is supported in part by the National Natural Science Foundation of China (61272481, 61572460), the National Key Research and Development Project (2016YFB0800703), the National Information Security Special Projects of National Development, the Reform Commission of China [(2012)1424], and the Open Project Program of the State Key Laboratory of Information Security (2015-MS-06) and China 111 Project (No. B16037). Peng Liu is supported by NSF SBE-1422215, ARO W911NF-13-1-0421 (MURI), and ARO W911NF-15-1-0576.

A Appendix

A.1 The Dangerous Permissions

Table 3 lists the dangerous permissions and the corresponding descriptions.

Table 3. Dangerous permissions LibCage enforces

Permission group	Permission	Description
CALENDAR	READ_CALENDAR	Read calendar data
	WRITE_CALENDAR	Edit calendar data
CONTACTS	READ_CONTACTS	Read a contact
	WRITE_CONTACTS	Edit or delete a contact
LOCATION	ACCESS_FINE_LOCATION	Get fine location
	ACCESS_COARSE_LOCATION	Get coarse location
PHONE	READ_PHONE_STATE	Read phone state
	CALL_PHONE	Make a phone call
	READ_CALL_LOG	Read call logs
	WRITE_CALL_LOG	Edit or delete call logs
SMS	SEND_SMS	Send a SMS
	READ_SMS	Read a SMS
	RECEIVE_SMS	Receive a SMS
STROGE	READ_EXTERNAL_STORAGE	Read a file
	WRITE_EXTERNAL_STORAGE	Edit or delete a file
CAMERA	CAMERA	Take a picture or video
MICROPHONE	RECORD_AUDIO	Record an audio

A.2 Interception on DVM

Different from Android Runtime (ART), Dalvik Virtual Machine (DVM) maintains a data structure called `ClassObject` (a C data struct) for each Java class and a structure (called `Method`) for each method declared in this class. When a method is called, DVM searches for the `ClassObject` related to the class in which this method is declared, and then the corresponding `Method`, which will be executed. Similar to the implementation on ART, we manipulate the data structure of a target method, and replace its pointer with the pointer of data structure of the proxy method.

A.3 The Details of Test Third-Party Libraries

Table 4 lists the details of permissions required by third-party libraries we tested on (✓: mandatory permission, O: optional permission, ×: not required).

Table 4. The details of permissions required by third-party libraries in our dataset

Library	INTERNET	ACCESS_NETWORK_STATE	ACCESS_COARSE_LOCATION	ACCESS_FINE_LOCATION	WRITE_EXTERNAL_STORAGE	NFC	ACCESS_WIFI_STATE	BLUETOOTH	VIBRATE	WRITE_CALENDAR	RECORD_AUDIO	READ_PHONE_STATE	READ_CALENDAR	GET_TASKS	CHANGE_WIFI_STATE
AdMob	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×
Flurry	✓	O	×	O	×	×	×	×	×	×	×	×	×	×	×
Unity	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Facebook	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Fmod	✓	×	×	×	✓	×	×	×	×	×	×	×	×	×	×
Chartboost	✓	✓	×	×	O	×	✓	×	×	×	×	O	×	×	×
InMobi	✓	✓	O	O	×	×	×	×	×	×	×	×	O	O	O
Google Gson	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×
MoPub	✓	✓	✓	O	O	O	O	O	O	O	O	×	×	×	×
Google Analytics	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×

References

1. Pearce, P., Felt, A.P., Nunez, G., Wagner, D.: Addroid: privilege separation for applications and advertisers in android. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, pp. 71–72. ACM (2012)
2. Wang, Y., Hariharan, S., Zhao, C., Liu, J., Du, W.: Compac: enforce component-level access control in android. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, pp. 25–36. ACM (2014)
3. Stevens, R., Gibler, C., Crussell, J., Erickson, J., Chen, H.: Investigating user privacy in android ad libraries. In: Workshop on Mobile Security Technologies (MoST). Citeseer (2012)
4. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proc. IEEE **63**(9), 1278–1308 (1975)

5. Gries, D., Schneider, F.B.: Monographs in computer science (2008)
6. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 95–109. IEEE (2012)
7. Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.-R.: Unsafe exposure analysis of mobile in-app. advertisements. In: Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 101–112. ACM
8. Shekhar, S., Dietz, M., Wallach, D.S.: Adsplit: separating smartphone advertising from applications. In: USENIX Security Symposium, pp. 553–567 (2012)
9. Zhang, X., Ahlawat, A., Du, W.: Aframe: isolating advertisements from mobile applications in android. In: Proceedings of the 29th Annual Computer Security Applications Conference, pp. 9–18. ACM (2013)
10. Jeon, J., Micinski, K.K., Vaughan, J.A., Fogel, A., Reddy, N., Foster, J.S., Millstein, T.: Dr. android and mr. hide: fine-grained permissions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 3–14. ACM (2012)
11. Sun, M., Tan, G.: Nativeguard: protecting android applications from third-party native libraries. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, pp. 165–176. ACM (2014)
12. Kawabata, H., Isohara, T., Takemori, K., Kubota, A., Kani, J.-I., Agematsu, H., Nishigaki, M.: Sanadbox: sandboxing third party advertising libraries in a mobile application. In: 2013 IEEE International Conference on Communications (ICC), pp. 2150–2154. IEEE (2013)
13. Hong, Y.-Y., Wang, Y.-P., Yin, J.: NativeProtector: protecting android applications by isolating and intercepting third-party native libraries. In: Hoepman, J.-H., Katzenbeisser, S. (eds.) SEC 2016. IFIP AICT, vol. 471, pp. 337–351. Springer, Heidelberg (2016). doi:10.1007/978-3-319-33630-5_23
14. Seo, J., Kim, D., Cho, D., Kim, T., Shin, I.: Flexdroid: enforcing in-app. privilege separation in android (2016)
15. Backes, M., Bugiel, S., Hammer, C., Schranz, O., von Styp-Rekowsky, P.: Boxify: full-fledged app. sandboxing for stock android. In: 24th USENIX Security Symposium (USENIX Security 15) (2015)
16. Zhou, Y., Patel, K., Wu, L., Wang, Z., Jiang, X.: Hybrid user-level sandboxing of third-party android apps. Memory **2200**(0500), 0e00 (2015)
17. Bianchi, A., Fratantonio, Y., Kruegel, C., Vigna, G.: Njas: sandboxing unmodified applications in non-rooted devices running stock android. In: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 27–38. ACM (2015)
18. Android service element (2016). http://developer.android.com/intl/zh-cn/guide/topics/manifest/service-element.html
19. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: ACM SIGOPS Operating Systems Review, vol. 27, pp. 203–216. ACM (1994)
20. System permissions (2016). http://developer.android.com/intl/zh-cn/guide/topics/security/permissions.html#normal-dangerous
21. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 217–228 (2012)
22. Rubin, X., Saldi, H., Anderson, R.: Aurasium: practical policy enforcement for android applications. In: USENIX Security Symposium, pp. 539–552 (2012)
23. Android 5.0 behavior changes (2016). http://developer.android.com/intl/zh-cn/about/versions/android-5.0-changes.html

24. Redirecting functions in shared elf libraries (2016). http://www.codeproject.com/Articles/70302/Redirecting-functions-in-shared-ELF-libraries#_Toc257815978
25. Top free in android apps (2016). https://play.google.com/store/apps/top
26. Wu, Y., Sathyanarayan, S., Yap, R.H.C., Liang, Z.: Codejail: application-transparent isolation of libraries with tight program interactions. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 859–876. Springer, Heidelberg (2012)
27. Yee, B., Sehr, D., Dardyk, G., Chen Bradley, J., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: a sandbox for portable, untrusted x86 native code. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 79–93. IEEE (2009)