Hindawi
Security and Communication Networks
Volume 2022, Article ID 3923690, 12 pages
https://doi.org/10.1155/2022/3923690

Research Article

A Secure Communication and Access Control Scheme for Native Libraries of Android Applications

Pengju Liu,[1,2] **Guojun Peng**,[1,2] **and Jing Fang**[1,2]

[1]*Key Laboratory of Aerospace Information Security and Trusted Computing of Ministry of Education, Wuhan University, Wuhan, Hubei 430072, China*
[2]*School of Cyber Science and Engineering, Wuhan University, Wuhan, Hubei 430072, China*

Correspondence should be addressed to Guojun Peng; guojpeng@whu.edu.cn

Received 5 March 2022; Revised 22 March 2022; Accepted 29 March 2022; Published 8 April 2022

Academic Editor: Muhammad Arif

The Android system cannot perform fine-grained permission management for TPLs (third-party libraries) in applications. TPLs can use all permissions of the host application, which poses a threat to users and system security. In order to solve this problem, this paper studies the underlying principles of multiple modules in the framework and kernel layers of the Android system. We construct a fine-grained access control scheme for native libraries through technologies such as isolation environment creation, remote function call, and dynamic permission management. We also implement the prototype system. The experimental results show that our proposed scheme can effectively assist developers in managing the permissions of the native libraries in a fine-grained manner and curb the suspicious privileged behaviors of the native libraries. Meanwhile, our proposed scheme is adaptable for high-version Android systems with reasonable overhead, and it will not break the current Android security mechanisms.

1. Introduction

TPLs are essential in the field of mobile application development. Developers can save significant costs by using TPLs. According to statistics [1], an average of more than 60% of the codes of Android applications come from TPLs. And in the context of the IoT (Internet of Things), in addition to smartphones, many other IoT devices are also based on Android. Therefore, it is essential to ensure the security of TPLs of the Android system.

Many studies [2–5] show that TPLs can steal private user data or device information in the background. Among them, [2] states that more than 40% of TPLs pose a threat to user privacy. Although Android system offers lots of access control mechanisms such as sandbox [6], permission model [7], AppOps [8], and SELinux [9], the granularity of these mechanisms is at the application level, which cannot further manage TPLs within the application. There is no security boundary between TPLs and the host application, so all permissions of the host application can be used by TPLs directly. This is the core reason for the threat of TPLs.

The solutions to the problem mentioned above can be summarized into two categories. The first is to isolate the TPLs to run in an environment outside the host application. The TPLs can no longer share the permissions of the host application and thus can only access limited system resources or user data [10–15]. For example, CompARTist [13] splits Android applications at compile-time into isolated, privilege-separated compartments for the host app and the included TPLs. Adcapsule [15] aims at the advertising libraries, which can make Ad libraries run independently. The second approach is to design a more granular permission mechanism that restricts the interaction between TPLs and resources. For example, FLEXDROID [16], Zhan et al. [17], LIB CAPSULE [18], and others enable the Android system to manage TPLs permissions directly by implementing library-level access control mechanisms.

There are many approaches to address the TPLs permission problem. However, none of these approaches have been applied in practice, and Android users still suffer from the aforementioned problems. This is mainly due to the following three reasons. First, TPLs are usually written in

Java, the same language as the Android development. But due to Android's support for JNI (Java Native Interface), TPLs can also be written in Native language, often called Shared Libraries or Native Libraries. In the past, Native Libraries were usually only used to implement some computationally intensive functions due to the performance limitations of mobile devices. However, as the performance of mobile devices improves, Native language is used significantly more frequently in TPLs. As Zhan et al. [19] state, *most existing tools can only handle Java TPLs.* Still, Native code has the same access rights as Java code, so many approaches can be bypassed by Native languages, which is no longer relevant today. Second, Google releases a new version of Android every year, and as of today, there are 12 major system versions of Android. These versions differ in many places, which results in that all approaches can only be deployed on specific versions. As far as we know, the highest supported system version among these approaches is Android 9. According to the latest statistics [20] of Android Studio, Android 10 and above system versions account for more than 50.2%. In other words, the current approaches can only run on a maximum of 49.2% of devices. Third, developers using third-party libraries intend to reduce their development costs. Still, many approaches require developers to modify their application bytecode and even the TPLs code. This runs counter to the developers' intent of using TPLs. These three reasons combine to make the state-of-the-art approaches not very practical. Therefore, we need a practical solution to manage Native library permissions in a fine-grained way.

To solve the permission problem of the shared library, a security-enhanced mechanism for the Android system called SEBox (Security-Enhanced Box) is proposed in this paper, which can run application's shared libraries in a low-permission isolated environment. Compared with the previous work, this mechanism does not require rewriting the code of applications or the shared library and can dynamically manage the permissions of the shared library. It is also compatible with the Android ART (Android Runtime) virtual machine. In this paper, the prototype system is implemented based on Android 10. Experiments are conducted to verify the functionality of SEBox and the reasonableness of the performance overhead. In the process of SEBox implementation, there is no need to modify the Android sandbox mechanism and the permission model. SEBox will not break the security mechanism of the stock Android system, so it is a security-enhanced mechanism for the Android system.

In summary, this paper makes the following contributions:

(i) We propose a scheme called SEBox to restrict the behavior of Native libraries in Android applications. To the best of our knowledge, SEBox has the highest Android system version support compared to other work.

(ii) We have implemented a prototype of SEBox, which focuses on restricting operations in Native libraries using techniques such as isolated environment creation, remote function calls, and dynamic permission management. Developers can conveniently deploy SEBox without any modifications to their application bytecode.

(iii) We evaluated our scheme on real Android devices, including effectiveness and performance tests. The evaluation results show that our scheme can completely limit the privileged behavior of Native libraries in Android applications with a low-performance overhead.

This paper is organized as follows. Section 2 introduces the threat model and the overview of our scheme. Section 3 introduces the necessarily related technologies. Section 4 systematically describes the scheme design and implementation details. Section 5 provides an experimental evaluation of our scheme, including effectiveness testing, performance testing, and comparison of similar works. Section 6 discusses the limitations and possible improvements of our scheme. Section 7 concludes this paper.

2. Scheme Overview

2.1. Threat Model. The threat model discussed in this paper is shown in Figure 1. The host application is trusted because any sensitive behavior requires the user's consent before being performed. As shown in Figure 1, before the host application can access data such as photo albums, it must be authorized with the user's consent. TPLs are untrusted due to the following reasons. First, TPLs have the necessary conditions to obtain sensitive data covertly. In the Android system, applications can access system services through APIs, and some sensitive information-related APIs are protected by permissions. However, the permissions are based on the application sandbox, and the TPLs run in the same sandbox as their host application. This means that the TPLs have the same permissions as the host application. Therefore, TPLs can directly use the permissions of the host application without any restrictions. Second, TPLs have the motivation to obtain sensitive data. Take the advertising library as an example. To push ads more accurately, they want to get as much information about the user as possible. Third, the codes of TPLs are usually not visible to the application developers, and most developers do not pay attention to the implementation of TPLs. Therefore, malicious code can be easily and covertly inserted into TPLs.

2.2. Architecture of SEBox. In Android, each application runs in its own sandbox environment. To make it easier to describe that sandbox environment, Android assigns a unique UID (User ID) to each sandbox environment. The main idea of SEBox is to create an additional application with a different UID from the host app and use it to load and run the shared libraries. We will refer to this additional application as the "isolated application" later on for ease of description. With the help of sandbox-based access control and authorization mechanism in the Android system, we can manage the permissions of the host application and the shared libraries separately.

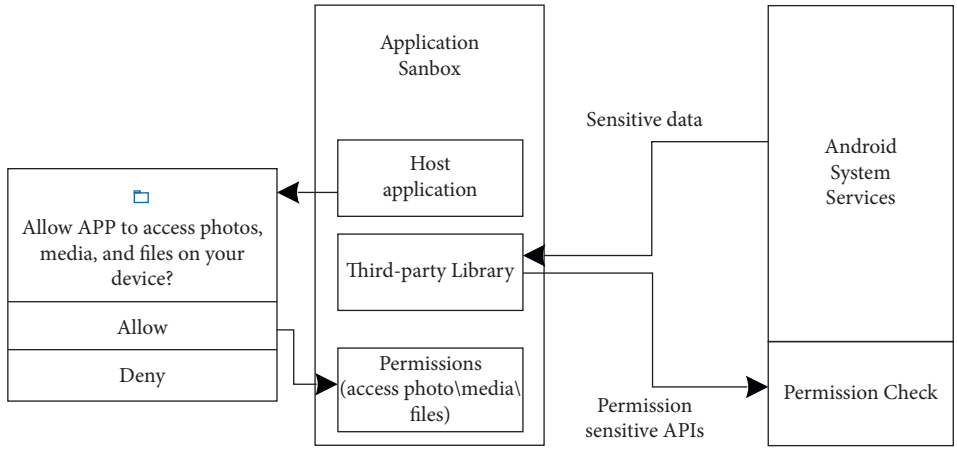

Figure 1: Threat model of the third-party library.

The architecture of SEBox is illustrated in Figure 2. The Manifest file is a configuration file where developers can provide basic information about their application to the Android system. The "original application" is the environment in which the system runs the host application code, and the "isolated application" is an isolated environment in which the shared libraries are run.

First, the developer configures permissions of the isolated environment and which shared libraries need to be isolated in the application's Manifest file. Then SEBox completes the isolation of the shared libraries in two separate work phases. In the application installation phase, the installer module additionally installs the "Isolated Application" based on the information configured in the Manifest file. For example, as shown in Figure 2, the "isolated application" only has *permission2* configured by the developer. In the application runtime phase, SEBox intercepts the shared library loading behavior of the host application and loads the shared libraries into the "isolated application" through the remote loading module. Then, SEBox supports the host application to call remote shared library functions by the remote calling module. In this way, the host application can normally run while the isolated shared libraries can only access limited system resources and sensitive user information. Thus, the permissions of the shared library are controlled by developers in a fine-grained manner.

2.3. Key Technical Points

2.3.1. Isolated Environment Creation. The isolated environment is used to load and run the shared libraries on the one hand and also carries the permissions of the shared libraries on the other hand. The isolated environment is one of our scheme's most basic and essential modules. Many previous works used the *isolated process* mechanism provided by the Android system to load and run shared libraries. However, the *isolated process* cannot obtain any permissions, causing these approaches to require additional modules to ensure the legal permission behavior of the shared libraries. In our scheme, we create a kind of isolated environment called "isolation application" by modifying the underlying modules of the system, which can freely assign permissions and avoid some additional overhead. At the same time, the process of creating the isolated environment is entirely automated by the system and does not require any operation by the developer.

2.3.2. Dynamic Permission Management. The purpose of running shared libraries in isolation is to manage their permissions. In this process, unnecessary permissions should be removed, but reasonable permission requirements should be ensured. On the earlier versions of Android, the shared libraries mainly consisted of computing-intensive functions. They rarely relied on system permissions, so the previous work focused on isolating the shared libraries but ignored the legitimate permission requirements of the shared libraries. As the demand for permissions for shared libraries increases, some shortcomings of the previous work are revealed. In this paper, by providing an interface for permission configuration, developers can accurately specify permissions of the shared libraries, which could ensure the execution of legal behaviors while restricting permissions of the shared libraries. Additionally, in the previous work, shared library permissions are not adjustable after the application is installed. Our scheme proposed achieves dynamic permission management of the shared libraries by working with the Android dynamic authorization mechanism.

2.3.3. Remote Function Call. Since the shared libraries run independently in the "isolation application," the host application will not find the shared library functions. To enable the host application to call the functions in the shared libraries correctly, it is necessary to convert the function calls that are finished through JNI to IPC (Interprocess Communication). Although Android provides Binder [21] IPC mechanism, Binder is just a tool. It is challenging to use Binder efficiently and easily to complete the above process. The previous work keeps the shared library in the host application and rewrites the functions in the shared library as proxy calls to remote functions. In this paper, the related

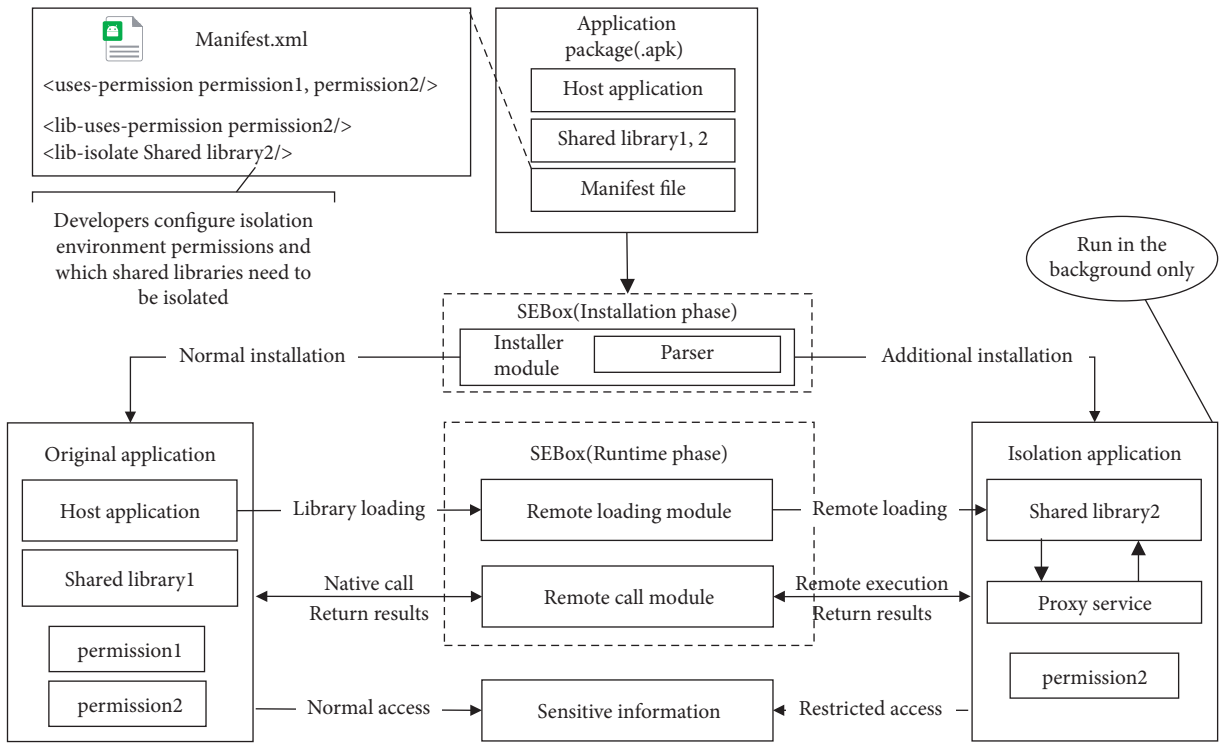

Figure 2: Architecture of SEBox.

underlying logic of the ART is modified so that the Android system can automatically complete the remote call of the host application to the library function.

These key technical points will be introduced in detail in Section 4.

3. Related Technologies

3.1. Android Permission Model. The Android permission model follows a "declare first, use later" rule. If developers want to use permission, they must declare it first in the application's Manifest file. In lower versions of Android, all the permissions declared by the application are granted by the system at install time. Android version 6 introduces a dynamic authorization mechanism, under which permissions can be roughly divided into install-time permissions and runtime permissions. The former is usually some permissions of the general level, which are granted automatically by the system during the application installation. The latter is usually permissions of the dangerous level, which must be dynamically requested from the user during the application runtime.

3.2. Android Package Manager Service. Package manager service (PMS) is one of the core services in the Android system, which manages the information of all applications through two members: *mPackages* and *mSettings*. As shown in Figure 3, *mPackages* is an index table, and *PackageParser.Package* (later referred to as *Package*) of each application can be queried by the application package name through this table. *Package* mainly stores the static

```
package com.android.server.pm;
public class PackageManagerService extends
IPackageManager.Stub{
    final ArrayMap<String, PackageParser.Package>
mPackages;
    final Settings mSettings;
}
public final class Settings {
    final ArrayMap<String, PackageSetting> mPackages;
}
```

Figure 3: Core members of PMS.

information of the application, including the requested permissions, component information, and application package name. *mSettings* also manages the *PackageSetting* of each application through an index table. *PackageSetting* mainly stores dynamic information of the application, including the status of permission and process information.

4. Scheme Design and Implementation

The workflow of our scheme can be divided into two phases, which are the installation phase and the runtime phase. The design and implementation of these two phases will be introduced in detail in this section.

4.1. Installation Phase. The goals of the application installation phase include the following. (1) After the installation, the original application is automatically installed into two parts: the host application and "isolation application." (2) The permissions of the host application and the "isolation

application" are independent of each other, and the permissions of the "isolation application" could be restricted.

4.1.1. Permission Extraction Module. This module is used for extracting and storing the required permissions of the shared libraries, which are used to restrict the isolated environment permissions lately. The permissions declared in the Manifest file are generally all the permissions P required by the application, including the permission P_H needed by the host application and the permission P_N needed by each shared library. The relationship between them is as follows.

$$P = P_H \bigcup P_{N1} \bigcup P_{N2} \bigcup P_{N3} \bigcup \ldots, \quad (1)$$

To manage the permissions of the shared libraries, it is necessary to decompose the application permissions, i.e., to determine the permissions required by each part of the application. In this paper, the developer is trusted in the threat model, so only the permission PN of the shared libraries needs to be determined. Our scheme adds a new shared library permission configuration interface to the Manifest file, through which developers can accurately configure the required permission PN of the shared libraries.

PackageParser is a tool used by Android to parse Manifest files. By modifying the PackageParser, the required permission of the shared libraries configured in the Manifest file will be extracted and stored into the *Package* during the application installation phase. In *Package*, the member *requestedPermissions* is used to store all the permissions P declared by the application. In this paper, a new member *libRequestedPermissions* is added to *Package* to store the required permission P_N of the shared libraries, as shown in Figure 4.

```
public final static class Package{
  public final ArrayList<String> requestedPermission;
  public final ArrayList<String> libRequestedPermissions;
}
```

Figure 4: Permission-related members in *Package* class.

4.1.2. "Isolation Application" Installation Module. This module is used to create the isolated environment, i.e., to install the "isolation application." There are many ways to install applications in Android, but in essence, they are all installed through PMS. We modified the relevant code in PMS so that it can install "isolated application" additionally. PMS generally completes the installation in four steps: *parse*, *scan*, *reconcile*, and *commit*. When PMS is ready to commit the installation results, the application is basically installed. Therefore, we choose to take over the installation process before PMS commits the installation results and starts installing the "isolation application," as shown in Figure 5.

From the view of the system, installation is the process that the application's dynamic and static information is organized in the form of *PackageSetting* and *Package* and stored in the corresponding management table of PMS. In our scheme, installing the "isolation application" is completed in four steps as follows.

Step 1: Create a Package for the "isolation application." The "isolation application" is similar to the host application in terms of many pieces of information, so the Package of the "isolation application" is created by copying that of the host application and modifying the different pieces of information in it. There are two essential modifications: a. Add the symbol "+" at the end of the package name of the "isolation application," so that our system can easily find the "isolation application" corresponding to the host application. For example, if the package name of the host application is "com.android.dialer," then set the package name of the "isolation application" to "com.android.dialer+." b. Modify the permission request information of the "isolation application." The Android permission mechanism is not modified in this paper, and the system still uses requestedPermissions for permission request judgment (introduced in Section 3.1.3). The new member libRequestedPermissions added in Section 3.1.1 is only used to temporarily store the permissions extraction results of the shared library and does not play a functional role. Therefore, the information in requestedPermissions needs to be replaced by the information in libRequestedPermissions in the Package of the "isolation application."

Step 2: Create PackageSetting for the "isolation application," which is similar to the process in Step 1.

Step 3: Register a new UID for the "isolation application," which can be done by the function registerAppIdLPw provided by the system. After the UID is successfully registered, the value of the UID needs to be updated in PackageSetting and Package.

Step 4: Add Package and PackageSetting of the "isolation application" to the system, i.e., store the information created in Step 1 and Step 2 into the two core members of the PMS, which are mPackages and mSettings.

4.1.3. Permission Management of Shared Libraries. The permissions declared by the developer in the Manifest file will eventually be stored in requsetedPermissions, but it does not indicate that the application has these permissions. The system actually records the status of an application for particular permission through PackageSetting.

In earlier versions of Android, after the initialization of PackageSetting, the status of the application's permissions is all False. And during the commit phase of the installation, the system grants these permissions. That is, the status will change to be True. In the high version of Android with the dynamic authorization mechanism, after the installation of the application, the status of the normal-level permission recorded in PackageSetting will be updated to True. In contrast, the status of the dangerous-level permission remains False and needs to be dynamically requested from the user. If the application dynamically applies for dangerous permission, the system will first check whether the permission is in requestedPermissions. If it is not, the system

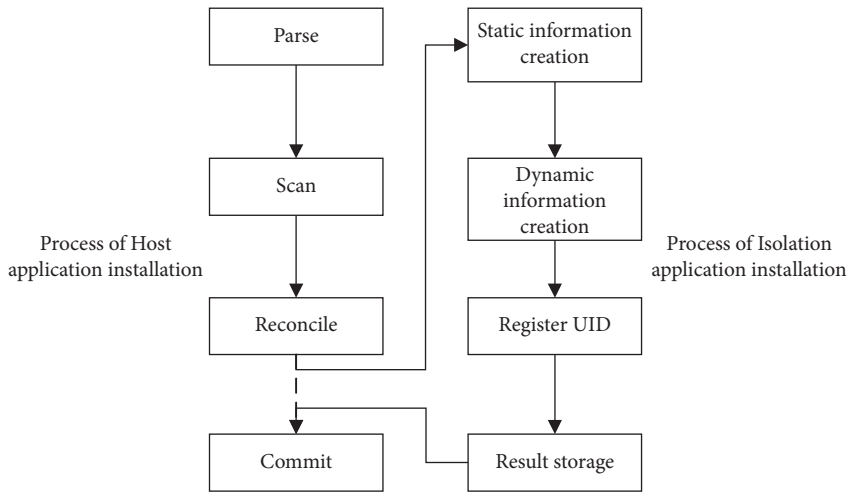

Figure 5: Installation process of the host application and "isolation application."

will consider that the application does not declare this permission and directly terminate the granting process.

In the installation of the "isolation application," its PackageSetting is copied from the host application, so the permissions remain the same as those of the host application after the installation. They have only some normal-level permissions, which do not pose a security threat. When the application is running, the dangerous permissions that the "isolation application" can request are limited because the permissions in its requestedPermission are replaced with the permissions configured by the developers. Alternatively, these permissions can be withdrawn by the user after authorization.

After the above process, the application is installed in two parts, including a host application and an "isolated application." During the runtime, the "isolation application" can only obtain the permissions configured by the developer at most, satisfying the two design goals of the installation phase.

4.2. Runtime Parse. The goals of the application runtime include the following. (1) When the host application loads a shared library, the relevant shared library should be loaded into the "isolation application." (2) When the host application calls a shared library function, the request should be sent to the "isolation application" automatically. The function should be executed in the "isolation application" and the result should be returned to the host application.

4.2.1. Shared Library Remote Loading Module. This module is used for remotely loading the shared library to the "isolation application." The workflow of this module is shown in Figure 6, which consists of the following 10 steps.

Step 1: The host application loads the shared libraries via ART.

Step 2: If the current shared library needs to be isolated, SEBox blocks the loading process of the native shared library and sends the remote loading request to the SharedLibraryLoad service. This is a new system service added by our scheme, used to coordinate the remote loading of shared libraries.

Step 3: The SharedLibraryLoad service determines if the isolated environment is ready. If the "isolation application" is not started, the SharedLibraryLoad sends the request of starting the "isolation application" to AMS (Activity Manager Service). AMS is one of the core Android system services, mainly used to manage the lifecycle of applications.

Step 4: AMS starts the "isolation application." In this process, the "isolation application" can be started as a background process by configuring the start parameters.

Step 5-6: After the "isolation application" is successfully started, the execution process will be returned to AMS, which will further send the start result to the *SharedLibraryLoad* service.

Step 7–10: The *SharedLibraryLoad* service creates a request and delegates ART to load the shared library into the "isolation application" and then sends the result to the host application.

After the above steps, the host application will mistakenly believe that its process has successfully loaded the shared library. Still, the shared library has been remotely loaded to the "isolation application," satisfying the first design goal of the runtime phase.

4.2.2. Native Function Remote Calling Module. The shared library is loaded into the "isolation application." When the host application calls a function in the shared library, the function should be executed in the "isolation application" and the result should be returned to the host application. This module is used for coordinating the above process, i.e., native method remote calls.

The relevant structures in the ART virtual machine are modified in this paper. Each Java method has a corresponding *ArtMethod* Object in ART, which describes

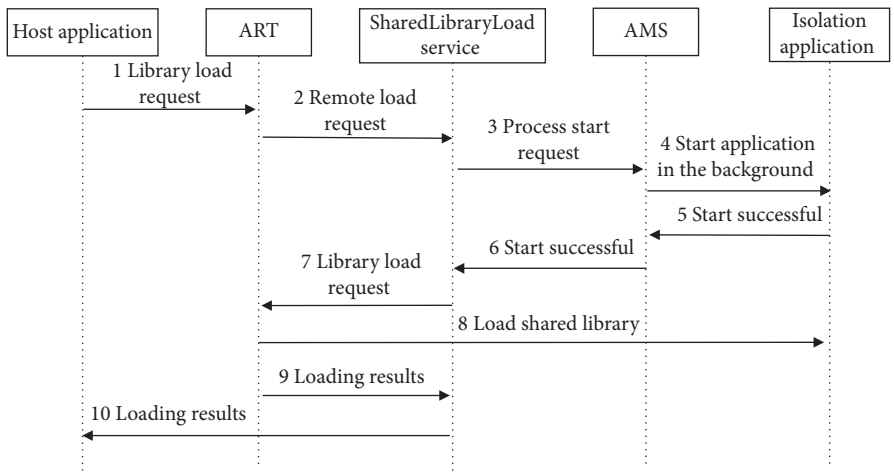

Figure 6: Workflow of loading remote shared library.

information about this method, including the machine code entry address. For Java methods implemented by Native language, the *ArtMethod* has two variables related to the machine code address, as shown in Figure 7. The machine code entry address of the method will point to the JNI machine code entry address, and the JNI machine code entry address will further point to the address of the relevant function in the shared library. If the method is not yet registered, its JNI machine code entry address will point to a piece of ABI code. In the stock Android system, the function of this ABI code is to find the target function in the shared library and update the JNI machine code entry address to the target function address. If the target function is not found, it will throw an exception. In SEBox, if the ABI code does not find the target function, the system will prevent the exception from being thrown and update the JNI entry address directly to a new function *callRemoteMethod* added in this paper. This function packages the name of the target function, call parameters, and other information to *Parcel* and then sends it to the *NativeMethodProxy* service via Binder. The *Parcel* is a container mainly used to store serialized data. *NativeMethodProxy* service is a new system service added in this paper, which is used for continuing to find and call the target function in the "isolation application" and finally returning the execution result to the host application. An exception will be thrown if the target function is still not found in the "isolation application."

In the above process, modified ART will autonomously complete the remote function call, satisfying the runtime phase's second design goal.

5. Evaluation

We evaluated SEBOX comprehensively from three perspectives. First, we constructed an actual threat scenario to test the functionality of SEBOX, i.e., to verify whether SEBOX can effectively intercept the overstepping behavior of a malicious native library. Second, SEBox was evaluated for performance on a representative benchmark suite and several benchmark apps. Finally, SEBox is compared to other similar works. SEBox is implemented based on AOSP (Android Open Source Project) 10.0.0_r1. The experiments were conducted on two Pixel2 smartphones running SEBox and the stock Android (AOSP 10.0.0_r1), respectively. All performance numbers were averaged over ten runs.

5.1. Functionality Test. To evaluate the functionality of SEBox, we constructed an actual threat scenario based on the threat model proposed in the overview section, as shown in Figure 8.

The left part of Figure 8 shows the UI of the sample application. Once the "Get Current Time" area is clicked, the current time will be displayed in the white space. The middle part of Figure 8 shows the code logic of this application. The developer obtains the current time through the function getTime() provided by the third-party shared library and displays the result through the TextView (A UI component for Android). In the threat model of this paper, the TPL is not trusted, in which it is easier to insert malicious code. As shown in the lower part of Figure 8, the malicious code makes reflection calls on the API *getLine1Number* to get the user's phone number and displays the result through the system log. Although this API is protected by the permission READ_PHONE_STATE of the Android system, the shared library can call this API as long as the host application has this permission. We install the application and grant it READ_ PHONE_ STATE permission on the stock Android system and SEBox. The system log output is shown in the right part of Figure 8 after clicking the "Get Current Time" area several times. Whenever a user obtains the current time in the stock Android system, the phone number will be leaked, while in the system based on SEBox, the shared library is isolated and runs in a low-permission environment. The library's malicious code cannot work due to the lack of necessary permissions. Therefore, SEBox can effectively manage the permissions of shared libraries and prevent them from overusing permissions.

5.2. Performance Overhead. SEBOX added the feature of fine-grained permission management of the shared library to the stock Android system, which inevitably reduced the

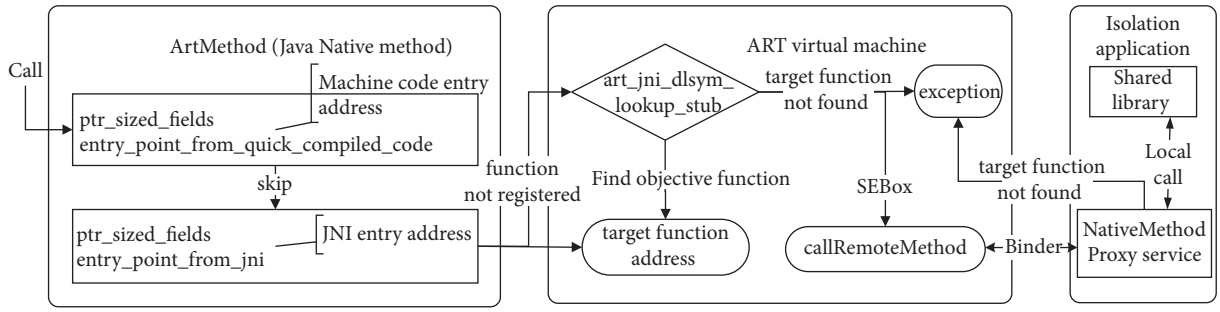

Figure 7: Flow of native method remote call.

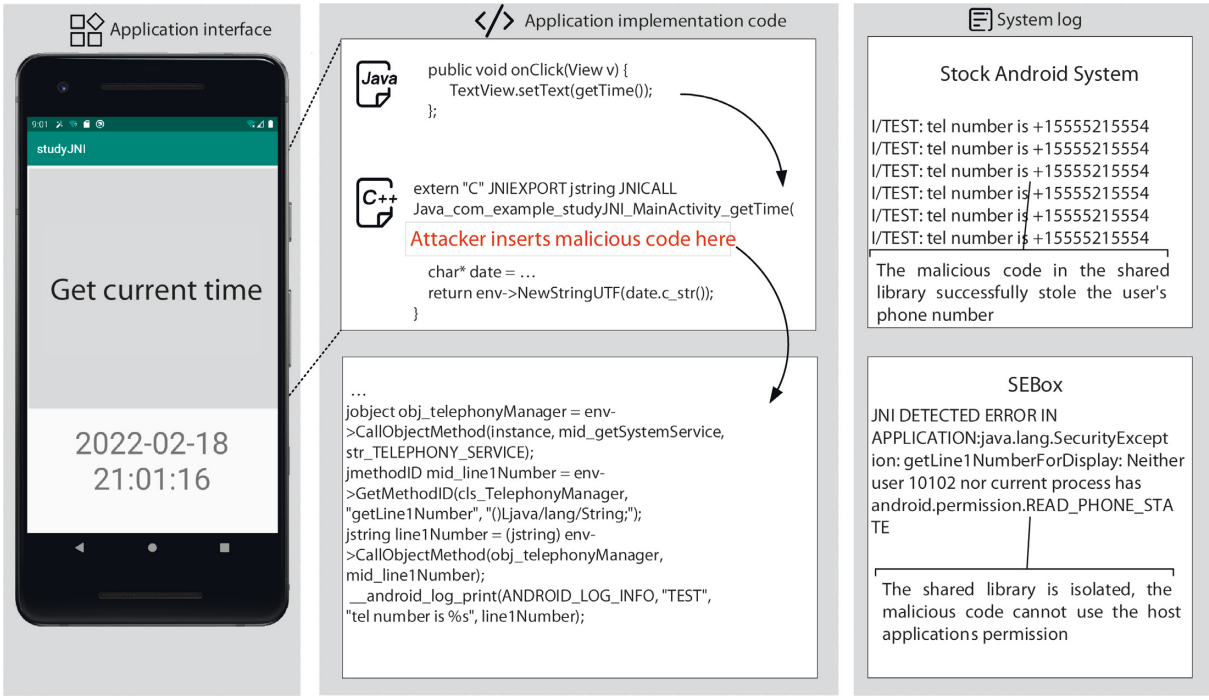

Figure 8: An actual threat scenario from Native library.

operational efficiency of the Android system. Compared with the stock Android system, our most significant changes are in the following two perspectives. First, we modified the application installation process. SEBox will additionally install an application called "isolated application." Second, we modified the process of calling shared library functions. The original in-process native method calls done through JNI are converted to Binder-based interprocess calls. Therefore, we first test the performance overhead of the above process and then test the overall performance of our system.

Overhead of Application Installation. To evaluate the app installation overhead, we collected a total of 200 applications from the APKPure store. These applications are collected from the "top 100 popular applications" and the "top 100 popular games" charts. The application installation time is closely related to the size of the APK (Android Package) file. As a simple example, the Android system will create local storage space for the application and copy the APK file to the local storage space during installation. The efficiency of this process depends on the size of the APK. Therefore, to accurately evaluate the overhead of application installation, we first performed a statistical analysis of the APK size in our dataset, and the result is shown in Figure 9. The result shows that the highest percentage of apps is in the APK size range from 20 MB to 200 MB, accounting for more than 60%. In addition, game applications tend to have a larger APK size, most likely because they usually contain more resource files. Next, we conducted application installation performance tests for different sized application datasets separately. We randomly selected an application in each dataset, tested its installation time on both the stock Android system and SEBOX, and then repeated this process ten times. Table 1 shows the following results. (1) Whether the stock Android system or SEBox, the installation time increases with APK file size. Intuitively, a larger APK will contain more contents (e.g., codes, components, and resources). Therefore, the

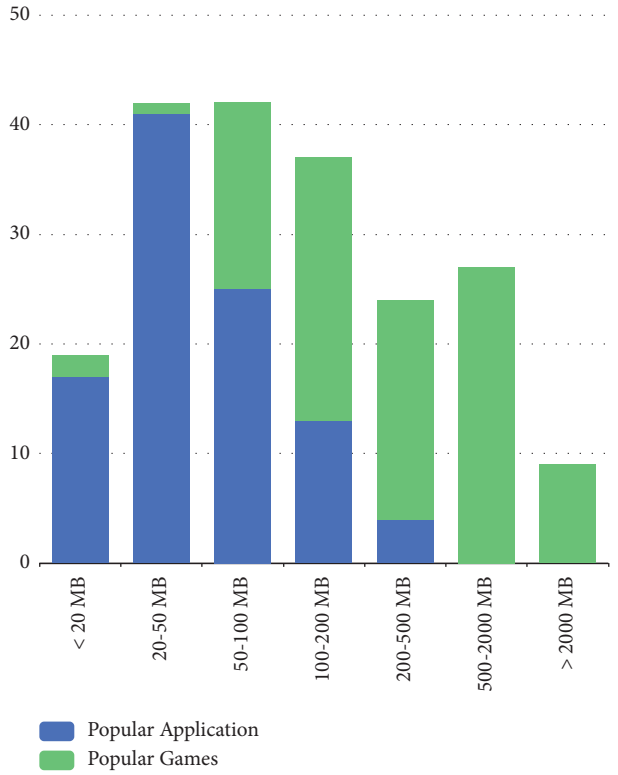

Figure 9: Statistics result of APK size. The horizontal axis represents the APK size (MB), and the vertical axis represents the number of applications in our data set with APK size in that range. The colors indicate different types of applications. Blue is for general applications, and green is for games.

Table 1: The performance overhead of application installation.

Apk size (MB)	Percentage (%)	Stock Android (s)	SEBox (s)	Delay (s)	Overhead (%)
<20	9.50	5.110	6.491	1.381	27.06
20–50	21.00	10.739	12.716	1.977	18.41
50–100	21.00	12.322	14.419	2.097	17.02
100–200	18.50	16.138	17.696	1.558	9.65
200–500	12.00	24.950	26.537	1.587	6.36
500–1000	13.50	39.653	41.331	1.678	4.23
>2000	4.50	63.465	66.175	2.710	4.27
Average	—	—	—	1.814	13.32

system needs to spend extra time processing these contents. (2) *Delay* describes the time required for SEBox to install the "isolation application." The value of delay is relatively small and stable in all datasets. The reason for the low delay is that the steps we take to install the "isolation application" are quite simple. The primary step is to create some data structures in memory. And the reason for the stability is that these data structures are independent of how much content is contained in the APK. (3) In some datasets, we have an overhead of up to 27.03%. However, considering the proportion of each dataset in practice, our system's average delay and overhead are 1.814 s and 13.32%, which are acceptable values.

Overhead of Remote Function Call. SEBox runs the Native library in the "isolation application." And the local calls between the host application and the Native library are converted into interprocess remote calls. This incurs additional processing time due to the interprocess communication. MiBench [22] is a free, commercially representative embedded benchmark, which provides many benchmark programs and corresponding datasets. We selected seven typical programs related to consumer electronics from MiBench and evaluated SEBox on them. Table 2 presents the results. SEBox shows a reasonable overhead (7.06% on smaller datasets and 4.71% on larger datasets) on these benchmark programs. Considering the reasons for performance overhead, in addition to the interprocess communication, there is another important reason. When a Native function is registered in the stock Android system, the host application can directly find the corresponding function address to call, while in SEBox, we need to look up the target function address in the remote "isolation application" during each call. It is worth noting that programs on large

Table 2: The performance overhead of remote function call.

BenchMark programs	Data size (kB)	Overhead (%)	Data size (kB)	Overhead (%)
jpeg	7	17.42	20	14.23
lame	174	1.34	2586	0.76
mad	41	7.89	373	2.41
tiff2bw	2224	6.61	6656	4.86
tiff2rgba	2224	3.59	6656	1.79
tiffdither	2224	8.37	6656	6.56
tiffmedian	2224	4.19	6656	2.34
Average	—	7.06	—	4.71

datasets show fewer performance overheads. This is mainly because the programs usually spend relatively less time on small data, so the performance overhead caused by SEBox will account for a larger percentage.

Overhead on BenchMark Apps. In addition to the parts we evaluated above, SEBox also introduces overhead in some other aspects, such as "isolation application" startup and remote loading of shared libraries. Therefore, we chose GeekBench [23] and AnTuTu [24] to measure the overall overhead of our system, both of which are prevalent benchmark applications. Geekbench tests device performance by performing four tasks (e.g., *Crypto*, *Integer*, *Floating Point*, and *Memory*), focusing more on computing performance. The Antutu benchmark app performs various tasks to give a final result, and it is more concerned with the overall system performance. The points score of SEBox is shown in Table 3. The overhead introduced by SEBox is only 3.55% on average.

In summary, SEBox introduces less performance overhead while enhancing the security of the stock Android system.

5.3. Comparison of Similar Approaches. Several approaches are similar to our scheme, all of which aim to achieve TPLs privilege deescalation. We compare these approaches from three perspectives. Table 4 shows the comparison details of 8 different similar approaches.

Native Library Supported. The Native library also runs in the same sandbox as the host application and shares all the permissions. Therefore, support for the Native library is an essential feature. However, many permission separation approaches do not support Native libraries, leading to the fact that malicious TPLs could bypass these approaches through native code. For example, AdCapsule mainly adopts Binder hooking, in-VM API hooking, and GOT (Global Offset Table) hooking to reliably hook the privacy-related APIs. However, TPLs can load a Native library to call the managers directly. Among these approaches, NativeGuard is the first work focusing on security threats of Native libraries. However, the shared libraries isolated by NativeGuard cannot obtain any permissions. A study by Afonso et al. [25] shows that placing the native libraries with no permission is not ideal because the native methods also require permissions for Java method calls through JNI and system calls. Libcapule uses the *isolated process* mechanism to isolate shared libraries, which also has the above problem because the *isolated process* cannot be granted any permissions. Although they solve this problem through additional modules, they also introduce additional performance overhead. In our scheme, the shared library is run in the "isolation application," an environment created by ourselves that can be flexibly assigned permissions.

Applicability. We compare the applicability of these approaches in two aspects. The first is the highest supported version. Our scheme could be deployed on Android version 10, the highest of all approaches. Considering the year in which each approach was published, it is not objective to compare applicability only from the highest supported version. For example, in 2014, the highest Android version was only 5.0. Therefore, we also compare applicability by device coverage, which we calculate from the official Android version distribution statistics. Due to the well-known Android fragmentation problem [26], the results are generally low. The approach with the highest device coverage is NativeGuard, while our solution ranks third at 26.57%.

Modification Mode. There are usually two modification modes: (1) modify the bytecode of applications and (2) modify the Android system. The former is generally easier to deploy than the latter, but the latter is more convenient from the developers' point of view. We complete the automation of many processes by only modifying the Android system. To effectively manage native library permissions, developers only need to add some configurations in their application Manifest file.

In summary, our scheme supports flexible permission management for the Native library, does not require developers to modify their applications, and has the highest supported version. Our scheme has advantages in functionality, applicability, and convenience compared with other approaches.

6. Discussion

In this section, we will discuss possible limitations, potential improvements, and other application scenarios of our system.

In order to restrict the permissions of shared libraries, we first need to know what permissions are really required for the shared library. To do this, we provide developers with some interfaces in the application Manifest file, which can be

Table 3: The performance overhead on benchmark apps.

Benchmark Apps	Stock Android (pts)	SEBox (pts)	Overhead (%)
GeekBench	1124	1076	4.27
AnTuTu	168591	1638120	2.83

Table 4: Comparison of existing library permission separation approaches.

Features	SanAdBox	NativeGuard	FlexDroid	CompARTist	Adcapsule	SplitSecond	LibCapsule	SEBox
Year	2013	2014	2016	2017	2018	2019	2021	2022
Native library supported	×	✓	×	×	×	✓	✓	✓
Highest supported version	4.0.4	4.3	4.4	7	6	7.1	9	10
Device coverage	23.30%	37.75%	22.63%	30.83%	23.55%	19.23%	18.26%	26.57%
Bytecode modification	✓	✓	×	×	✓	×	✓	×
System modification	×	✓	✓	✓	×	✓	×	✓

✓ means yes; × means no.

used to configure which shard library needs to be isolated and what permissions the shared library can use. However, there are many different kinds of shared libraries. It is difficult for a developer to know exactly what permissions are required for each shared library, especially for some less experienced developers. Although many TPLs specify the required permissions in their development documents, according to [16]'s research, 17 of the 20 popular TPLs they analyzed used undocumented permissions. Therefore, it is not a friendly way to let developers configure permissions themselves. We plan to explore techniques that efficiently help developers determine the minimum permission requirement of shared libraries. This could be achieved by scanning permission-sensitive APIs in shared libraries' import tables and path-sensitive strings in libraries. For example, if the string "/mnt/sdcard/a.dex" and API "open" are found, we will assume the shared library needs EXTERNAL_STROAGE permission.

The initial objective of SEBox is to defeat shared library-caused threats. However, as a security-enhanced and integral Android system, SEBox provides not only a fine-grained permission mechanism to confine the potential damage caused by shared libraries but also an isolated environment with controllable permissions, which can be further used for the dynamic analysis of applications. Thus, not only can SEBox be a hardened OS, but also it can be used as a platform for application analysis, especially for inspecting shared library's behavior.

7. Conclusion

This paper designed and implemented a system named SEBox to defend intra-application threats caused by untrusted shared libraries in applications. By separating these shared libraries into a new application with a few permissions and providing IPC-based interactive mechanisms for host apps and shared libraries, SEBox successfully gets shared libraries isolated. Consequently, applications running in SEBox are immune from shared library-caused threats.

Evaluations on SEBox proved its effective defense capability and reasonable performance overhead. Furthermore, SEBox works transparently for application developers and does not break the app's integrity. SEBox can be either deployed as a security-enhanced OS for personal use or used as a platform for application analysis.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (62172308, U1626107, 61972297, and 62172144).

References

[1] H. Wang and Y. Guo, "Understanding third-party libraries in mobile app analysis," in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 515-516, IEEE, 2017.

[2] C. Schindler, M. Atas, T. Strametz, J. Feiner, and R. Hofer, "Privacy leak identification in third-party Android libraries," in *Proceedings of the 2022 Seventh International Conference on Mobile and Secure Services (MobiSecServ)*, pp. 1–6, IEEE, 2022.

[3] J. Wang, Y. Xiao, X. Wang et al., "Understanding malicious cross-library data harvesting on android," in *Proceedings of the 30th USENIX Security Symposium*, pp. 4133–4150, 2021.

[4] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements, ..., and privacy leaks," in *Proceedings of the The 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*, California, CA, USA, 2018.

[5] M. Hammad, H. Bagheri, and S. Malek, "DelDroid: an automated approach for determination and enforcement of least-privilege architecture in android," *Journal of Systems and Software*, vol. 149, no. 1, pp. 83–100, 2019.

[6] S. Kumar and S. K. Shukla, "The state of Android security," in *Proceedings of the Cyber Security in India*, pp. 17–22, Springer, Singapore, 2020.

[7] I. M. Almomani and A. A. Khayer, "A comprehensive analysis of the android permissions system," *IEEE Access*, vol. 8, pp. 216671–216688, 2020.

[8] M. Hatamian, J. Serna, K. Rannenberg, and B. Igler, "Fair: fuzzy alarming index rule for privacy analysis in smartphone apps," in *Proceedings of the International Conference on Trust and Privacy in Digital Business*, pp. 3–18, Springer, Cham, 2020.

[9] S. Smalley and R. Craig, "Security enhanced (SE) android: bringing flexible MAC to android," *Ndss*, vol. 310, pp. 20–38, 2013.

[10] H. Kawabata, T. Isohara, K. Takemori et al., "Sanadbox: sandboxing third party advertising libraries in a mobile application," in *Proceedings of the 2013 IEEE International Conference on Communications (ICC)*, pp. 2150–2154, IEEE, 2013.

[11] M. Sun and G. Tan, "Nativeguard: protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & mobile Networks*, pp. 165–176, Oxford, UK, 2014.

[12] Y. Liang, *Research on Key Technologies of Software Binary Code Reuse*, Wuhan University, Wuhan, China, 2016.

[13] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The art of app compartmentalization: compiler-based library privilege separation on stock android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1037–1049, 2017.

[14] J. Lee, A. V. Raja, and D. Gao, "Splitsecond: flexible privilege separation of android apps," in *Proceedings of the 2019 17th International Conference on Privacy, Security and Trust (PST)*, pp. 1–10, IEEE, New Brunswick, NB, Canada, 2019.

[15] X. Zhu, J. Li, Y. Zhou, and J. Ma, "AdCapsule: practical confinement of advertisements in android applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 3, pp. 479–492, 2018.

[16] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "FLEXDROID: Enforcing in-app privilege separation in android," in *Proceedings of the NDSS*, 2016.

[17] J. Zhan, Q. Zhou, X. Gu, Y. Wang, and Y. Niu, "Splitting third-party libraries' privileges from android apps," in *Proceedings of the Australasian Conference on Information Security and Privacy*, pp. 80–94, Springer, Cham, 2017.

[18] J. Qiu, X. Yang, H. Wu, Y. Zhou, J. Li, and J. Ma, "LibCapsule: Complete Confinement of Third-Party Libraries in Android Applications," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[19] X. Zhan, T. Liu, L. Fan et al., "Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review," *IEEE Transactions on Software Engineering*, 2021.

[20] W. Jules: Google's Latest Android Version Distribution Numbers Show 11 in Dead Heat with 10, 2021, https://www.androidpolice.com/googles-latest-android-version-distribution-numbers-show-11-in-dead-heat-with-10/.

[21] Google, "Using Binder IPC," 2022, https://source.android.com/devices/architecture/hidl/binder-ipc.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization*, pp. 3–14, IEEE, 2001.

[23] P. Labs, "A cross-platform benchmark," 2020, https://www.geekbench.com/.

[24] A. Labs, "Antutu Benchmark," 2022, https://www.antutu.com/en/index.htm.

[25] V. Afonso, A. Bianchi, Y. Fratantonio et al., "Going native: using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Proceedings of the The Network and Distributed System Security Symposium*, pp. 1–15, 2016.

[26] P. Mutchler, Y. Safaei, A. Doupé, and J. C. Mitchell, "Target fragmentation in Android apps," in *Proceedings of the 2016 IEEE Security And Privacy Workshops (SPW)*, pp. 204–213, IEEE, 2016, May.