NativeProtector: Protecting Android Applications by Isolating and Intercepting Third-Party Native Libraries

Yu-Yang Hong, Yu-Ping Wang, Jie Yin

NativeProtector: Protecting Android Applications by Isolating and Intercepting Third-Party Native Libraries

Yu-Yang Hong, Yu-Ping Wang, and Jie Yin

National Laboratory for Information Science and Technology,
Tsinghua University, Beijing, 10084, China
hyy13@mails.tsinghua.edu.cn,wyp@mail.tsinghua.edu.cn,yinjie@mail.tsinghua.edu.cn

Abstract. An increasing number of Android developers are incorporating third-party native libraries in their applications for code reuse, CPU-intensive tasks and other purposes. However current Android security mechanism can not regulate the native code in applications well. Many approaches have been proposed to enforce security of Android applications, but few of them involve security of the native libraries in Android applications.
In this paper, we propose NativeProtector, a system that regulates the third-party native libraries in Android applications. The standalone Android application is separated into two components: the server app and the client app where server app contains the native libraries for providing services from the native libraries while the client app contains the rest parts of the original app. The client app binds to the server app at the launching time, and all native function calls are replaced with interprocess calls to the server app. NativeProtector also generates the stub libraries intercept system calls in server app and enforce security of the native libraries in server app. We have implemented a prototype of NativeProtector. Our evaluation shows that NativeProtector can successfully detect and block the attempts of performing dangerous operations by the third-party native libraries in Android applications. The performance overhead introduced by NativeProtector is acceptable.

Keywords: Android security, Native libraries, Process isolation, Call interception

1 Introduction

Android dominated the smartphone market with a share of 82.8% in the second quarter of 2015 [6]. This trend is benefited from the great increase of third-party Android applications, because they can be easily downloaded and installed. However, numbers of malicious applications also occur to leak user private information and perform dangerous operations. Therefore, preventing privacy leaks and enabling fine-grained control in Android applications are necessary.

Many approaches have been proposed to protect the security of Android applications, but they often focus on Java code, because Android applications are

often written in Java language. In fact, Android also provides JNI (Java Native Interface) for calling native libraries in applications, and many developers tend to use third-party native libraries to reuse existing code or perform CPU-intensive tasks (such as image filtering and video encoding). However, the security of these third-party native libraries is often omitted [19]. In the Android system, these native libraries can access the entire process address space and share all the permissions which the user grants to the whole applications, and they are also uncovered by Java security mechanism. Thus, malicious native libraries are very dangerous for Android security.

To the best of our knowledge, only a few existing approaches focus on the security of native libraries in Android. NativeGuard [21] is a typical framework which uses process isolation to sandbox native libraries of applications. It has two main advantages. Firstly, NativeGuard separates native libraries to another standalone application, so native libraries can not fully access the entire application address space, and the interaction between native libraries and Java code is fulfilled via Android Inter-Process Communication (IPC) mechanism. Secondly, the generated native-library application is no longer granted permissions, so dangerous operations can not be performed.

However, NativeGuard still has some limitations. Firstly, because no permissions are granted to the native-library application, the benign native libraries crash when they need necessary permissions. Secondly, NativeGuard lacks fine-grained control of the native libraries to manager their behaviors.

To ensure the security of native libraries in Android, we propose a practical approach named NativeProtector. On one hand, inspired by NativeGuard, we use the process isolation to prevent the native libraries from accessing the entire application address space and limit the permissions of native libraries. On the other hand, we perform fine-grained control of native libraries by instrumenting the third-party native libraries and intercepting native-library calls to access private data and perform dangerous system calls. In detail, the third-party native libraries are separated as a standalone application, so the access of native libraries to Java code is restricted by fine-grained access control. *By combining isolation and interception, we can ensure the security of native libraries without crashing benign native libraries.* Meanwhile, NativeProtector is very easy to deploy. It can run as a common application without the root privilege because NativeProtector statically instruments the target application. Hence, no modification is required for the Android framework.

The main contributions of this paper are following:

- By combining isolation and interception, we have proposed a practical approach named NativeProtector, to protect Android applications from malicious third-party native libraries.
- We have built a prototype of NativeProtector to separate an application to the native-library application and Java code, and instrument the native libraries to perform fine-grained access control.
- We have evaluated NativeProtector on real-world and manually crafted applications. The experimental results show that NativeProtector is effective

for security and compatible for many applications, and the performance overhead is also acceptable.

The rest of this paper is organized as follows: Section 2 provides some background information on Android security and JNI, we also talk about dynamic loading and linking in Android. In Section 3 we describe the threat model of NativeProtector and defenses provided by NativeProtector. Section 4 goes through details about NativeProtector's implementation. In Section 5 we evaluate effectiveness, compatibility and overhead of NativeProtector . Related work is shown in section 6. Section 7 gives the conclusion of this paper.

2 Background

In this section, we first briefly give an overview of Android security, and then introduce some important concepts in Android to help to better understand NativeProtector.

2.1 Android Security Overview

Android OS is an open-source software stack for mobile devices consisting of a Linux kernel, Android application framework and system applications. In Android, each application runs in a separate sandboxed environment that isolates data and code from other applications which is guaranteed by Linux kernel's process isolation. One application can not access to another application's address space and private data. Inspired by this mechanism, the third-party native libraries can be separated as another application to ensure that they can not access to the entire application's address space or private data.

2.2 Java Native Interface

Similar to the desktop Java program, Android provides the Java Native Interface (JNI) to define a framework for Java code and native code to call each other. Commonly, developers use native libraries in their applications for code reuse, CPU-intensive task or application hardening. Android provides Native Develop Kit (NDK) [2] to allow developers to implement parts of applications in native languages like C and C++. NDK compiles native source code files into shared libraries which can be loaded dynamically under the request of the application's java code. When a native function is invoked, it will be passed a special data structure of type JNIEnv, which allows the native code to interact with the java code [20]. For instance, the native code can use JNIEnv->FindClass("Sample") to locate the Java class "Sample" and call its functions. Inspired by this mechanism, these key JNI related functions can be interposed to intercept the access to private data and dangerous function calls based on predefined policies.

2.3 Dynamic Loading and Linking

Android adopts its own dynamic loader and linker for native libraries. Unlike the desktop Linux operating system, Android's loader do not take the lazy Binding policy, which means the loader will recursively resolve all the external functions when the application is loading into the memory. But the PLT/GOT [7] structure is still used for dynamic linking. In particular, for an ELF file which has some external functions, its call sites to an external function is actually jump instruction to a stub function in the Procedure Linkage Table (PLT). This stub function performs a memory load on a entry in the Global Offset Table (GOT) to retrieve the real target address of this function call. When a native library is loaded, the loader resolves all external functions and fills them in the GOT entries.

3 System Design

In this section, we first give the threat model of NativeProtector, and then we explain how can we prevent the damage from the third-party native libraries and the defenses provided by NativeProtector.

3.1 Threat Model

There are three main advantages for using native libraries or native code in Android applications: a) porting applications between platforms; b) reusing existing libraries or providing libraries for reuse; c) increasing performance in certain cases, particularly for CPU-intensive applications like pixel rendering, image filtering and games. For these advantages, many Android developers tend to incorporate native libraries in their applications. But these native libraries are always developed by the third party, so they can be malicious. Note that a carefully designed Android application may have security check on passing sensitive data to native library. However, the developers usually trust the native libraries and fail to perform such checks.

Similar to existing solutions [21], we assume an adversary model that the third-party native libraries in the applications are not trustworthy, but some native libraries are essential to the functionalities of the applications. We need to ensure the applications with third-party native libraries work as normally as possible when restricting the third-party native libraries' unprivileged operations such as accessing to the private data

In this paper, the developer is trustworthy so that the Java code of the applications is trustworthy. As there are many approaches of enforcing the security of Java code in Android applications [13, 16, 25, 26], it's reasonable that the Java code can be regulated well even though it involving some malicious code.

3.2 Defenses Provided

As shown in Fig. 1, NativeProtector separates the original app to two standalone apps. One of them consists of Java bytecode and the resources of the original

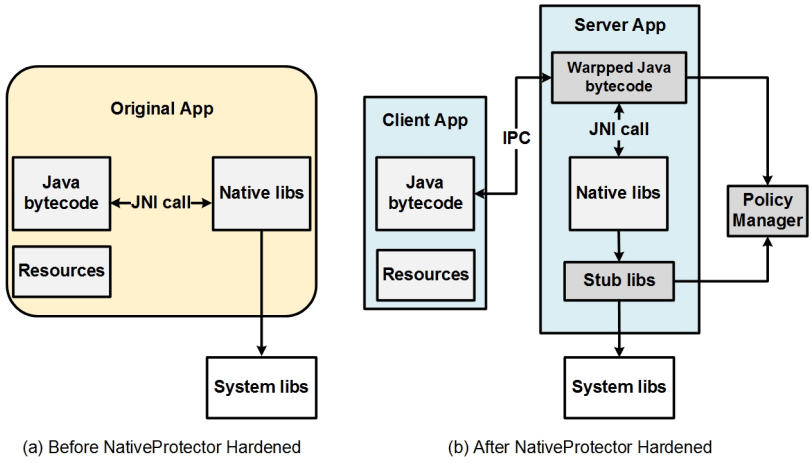

Fig. 1. System architecture of NativeProtector

app (like layouts, values et al). The other consists of the third-party native libraries implemented as an Android service [3]. The Java one acts as the client, and the native one acts as the server. These two apps communicate with each other via Android IPC. In this way, the security of native libraries is controlled by Android's existing process isolation security mechanisms, which means the third-party native libraries can not access to the entire application address space. To support arbitrary applications, where source code is not always available, we craft the application separation process in bytecode instead of source code.

To control the third-party native libraries' access to private data and dangerous operations, NativeProtector inserts hooking libraries into the server app which contains the third-party libraries. These hooking libraries intercept the interactions between the third-party native libraries and the system to enforce various security policies. To control the third-party native libraries' access to private data and dangerous operations by JNI calls, we also intercept the JNI calls called by the native code to enforce security policies.

Use case: For each app to be installed, NativeProtector separates it into two standalone apps. They are both installed on the user's cellphone and the server app is installed as an Android service. When the client app is launched, it starts the server app and binds to the service of the server app. When the client app needs to call the functions in the native libraries, the client app will interact with the server app through IPC.

4 Implementation

We have implemented a prototype of NativeProtector in Java and C programming languages.

4.1 Apk Repackaging

Each Android application is distributed as a package file format which is named APK (Android Application Package). An APK file contains a manifest file named AndroidManifest.xml, the application's code in form of dex bytecode, XML resources like activity layouts, other resources such as images, and the native libraries which are standalone Linux shared object files (.so). To support arbitrary applications, all we worked is on the APK file.

An APK file is actually a ZIP compression package, which can be easily decompressed with decompression tools. Because the Android SDK puts all the application's compiled bytecode in a single file called classes.dex, and the XML files are also compressed, we can not edit the bytecode or XML files directly to add our protection code. We need to take the original APK files, disassemble it to a collection of individual classes and XML files, add NativeProtector's code in them, and then reassemble all the things back to new APK files.

To perform this task we choose *apktool* [5], a tool for Android applications reverse engineering which can decode resources to nearly original form and rebuild them after making some modifications. In NativeProtector *apktool* is taken in the repackaging process. We first take *apktool* to disassemble the APK file to manifest file and resources, the native libraries, and the application's bytecode which is in the *smali* format. Then we use these files to generate the client app source files and server app source files. Meanwhile, we insert our stub libraries into the server app source files to intercept the private data access and dangerous operation for enforcing security policies. Finally, we use *apktool* to reassemble the client app and server app to APK files, and install them on the user's phone.

4.2 Native Libraries Isolation

NativeProtector isolates the native libraries to another application as an Android service. Actually, it generates several AIDL(Android Interface Definition Language [1]) interfaces and assistant smali code, and then modifies the launcher Activity and the JNI calls in the applications to achieve isolation. Fig. 2. shows the detailed process of the isolation of NativeProtector. Next, we illustrate some key points in the isolation process.

Generate Server Application After disassembling the original app's APK file by *apktool*, all smali files of the application are analyzed to record native function information. Then for each native function, NativeProtector creates a corresponding AIDL interface and an Android wrapper function in the server for communicating with the client. When the client calls the native function, there is an IPC call to the corresponding wrapper function with the same parameter, and the wrapper function invokes the native function through the AIDL interface. Finally, the native functions with AIDL interfaces and wrapper functions are reassembled into the server app using *apktool* .

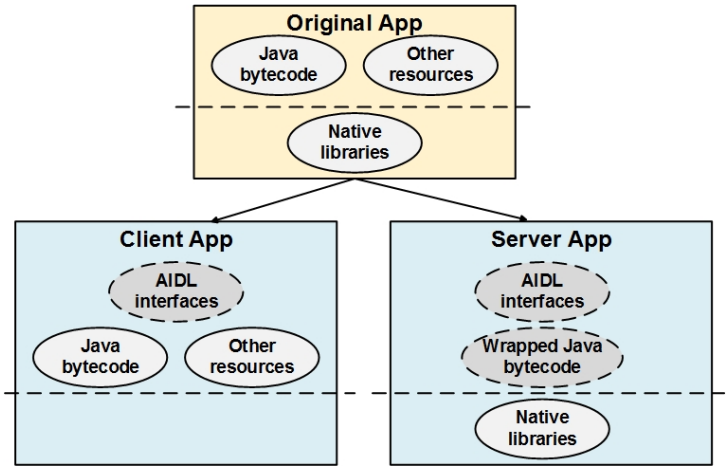

Fig. 2. Isolation Process of NativeProtector

Generate Client Application After the server app is generated, we use the generated AIDL interfaces to craft the client app. We analyze all the smali files of the original app automatically to change the native function calls to corresponding AIDL calls. Then we use *apktool* to reassemble the Java bytecode of the orignal app which has been modified as above mentioned, the AIDL interfaces corresponding to the server app, and the other resource files of the original app to the client app as an APK file.

Modify the Launcher Activity To make the client app works normally we need to ensure that at the very beginning in the client app, the server app's service is bound to the client app and the client app can call the native functions normally. Launcher activity is the starting execution point of an Android application. A callback method of the activity named onCreate is automatically invoked by the system when the application is launched. So NativeProtector locates this callback method in the launcher activity, and inserts code to bind to the server app to ensure that the client app is launched together with the server app.

4.3 Native Libraries Interception

In the process of the native libraries isolation, NativeProtector instruments the native libraries for intercepting the private data access and dangerous operations. This interception is implemented as some hooking libraries which are packed in the server app, and we let the server app load the hooking libraries and perform the hooking operation for native libraries interception. NativeProtector also provides a policy manager employing various security policies to enforce security. Next we detail the native libraries interception process of NativeProtector.

Efficient Interception The PLT/GOT structure is used for dynamic linking in Android. The call sites to external functions in Android native libraries are jump instructions to stub functions in PLT, and the stub functions then perform memory load on the entries in the GOT to retrieve the real address of these functions. We can exploit this mechanism to perform the required interposition. We scan every loaded native library and overwrite each GOT entry with pointer to our stub function. In this way, we can intercept the private data access and dangerous operations.

Policies As we are able to insert hooking libraries which can intercept the private data access and dangerous operations to monitor the third-party native libraries, we can introduce the various security policies in NativeProtector to enforce security. At present, we implement some typical security policies in NativeProtector. In theory, NativeProtector can perform as a flexible framework to adopt many more useful security policies.

In NativeProtector, we monitor mainly three kinds of operations, including accessing private data, connecting to remote server and sending SMS (Simple Message Service) messages. We consider those operations, because they cover the two ends of path that may leak private data. The policies taken by NativeProtector to regulate the third-party native libraries in these three operations are described below:

- **Private Data Policy** This policy protects the user's private data such as IMEI, IMSI, phone number, location, stored SMS messages and contact list. These private data can be accessed from the system services provided by Android Framework APIs. These Android Framework APIs call a single call to the *ioctl()* function. NativeProtector intercepts the calls to *ioctl()*, and parses the data passed in the calls to determine which service is accessed. Then we can know which private data is accessed by the native libraries and decide whether allow the private data access of the native libraries.
- **Network Policy** Android uses *connect()* function for socket connection. All Internet connections in Android call this function eventually. NativeProtector can intercept this function to control the Internet access of the native libraries. We restrict the native libraries to connect to only a specific set of IP addresses and prevent the native libraries from connecting to malicious IP addresses. With these whitelist and blacklist policies, we can ensure the native libraries are regulated int network, and the normal use of the native libraries is not affected. The whitelist and blacklist can be managed by user, or be retrieved from a trusted server.
- **SMS Policy** In Android framework, applications can not send SMS messages on their own. The applications must invoke RPCs to the SMS service through Binder. All the interactions with Binder call *ioctl()* function eventually. As we mentioned above, NativeProtector intercepts the calls to *ioctl()*. Thus we can get the destination number and the content of the SMS message to inform the users, and decide whether to allow this operation or not. Meanwhile we can take a blacklist policy to prevent the native libraries from

sending SMS messages to premium numbers. As NativeProtector only block sending SMS message to premium numbers, calling to those numbers is not blocked unless the phone call policy is adopted.

5 Evaluation

In this section, we evaluate the effectiveness, compatibility and performance of our prototype of NativeProtector. The experiments are performed on a Google Nexus 4 phone running Android 4.4.2.

5.1 Effectiveness

To show the effectiveness of NativeProtector, we have manually designed a demo app. This demo app contains a malicious but inseparable native library, which is used by the demo app for network connection. But this malicious native library gets the location information of the phone and send to a known malicious server. The demo app uses native system calls to perform this process instead of Java APIs.

This demo app demonstrates that the third-party native libraries may make use of the permissions assigned to the applications and cause security violations. As for NativeGuard [21], it simply separates the native libraries to another server app, and gives no permissions to the server app. This approach indeed restricts the third-party native libraries' access to all the application's process address which can change the execution of java code, but it also leads to the result that the third-party native libraries can not work any more. In this app, the native library can not connect to the network, even for the benign servers.

NativeProtector can improve the situation by combining separation and interception. For this demo app, NativeProtector separates the native library to the server app and generates hooking libraries to the server app. The server app has the permissions same as the original app, so the third-party native library can work well. But when the third-party native library accesses to the phone location (private data in the test), and when the third-party native library connects to the malicious IPs in the blacklist, the hooking libraries will intercept these dangerous function calls and block them. In this way, NativeProtector can help the app to use the third-party native libraries functionality and prevent applications from the malicious third-party native libraries.

5.2 Compatibility

To test the compatibility of our NativeProtector prototype, we have collected 20 popular apps from APKPure[4] store , a applications store which can ensure the applications download from it are the same as Google Play store. These applications are downloaded from the leaderboard of the hot free applications chart of the store in November 2015. NativeProtector successfully separates 15 of 20 applications. The five failed applications are due to the *apktool*. Two of them

fail in the disassembling stage before NativeProtector's separation, and three of them fail in the assembling stage after NativeProtector's separation. Then we manually test the applications after separation by playing with the instrumented applications and using the functionalities of the instrumented applications. In the test, all the 15 applications processed by NativeProtector can work well. During the testing, we found that there are response delays in some applications. But on the whole, NativeProtector introduces acceptable overhead and does not affect the app's functionalities. Details about the evaluated applications are presented in Table 1.

Table 1. Apps used in Compatibility test (∗: failed by *apktool*)

App	Size	Package name
DuoLingGo	9 M	com.duolingo
ES File Browser∗	5.9 M	com.estrongs.android.pop
Shadowsocks	3.7 M	com.github.shadowsocks
IMO	6.7 M	com.imo.android.imoim
Arrow Desktop	4.5 M	com.microsoft.launcher
Microsoft Outlook	18.1 M	com.microsoft.office.outlook
Snapseed	20.7 M	com.niksoftware.snapseed
Brightest LED Flashlight	5.1 M	com.surpax.ledflashlight.panel
Tiger VPN	5.8 M	com.tigervpns.android
Tumblr	21.1 M	com.tumblr
Twitter	21.7 M	com.twitter.android
HideMe VPN	5.8 M	io.hideme.android
OpenVPN Connect	2.3 M	net.openvpn.openvpn
New York Times∗	1.2 M	org.greatfire.nyt
WiFi key∗	6.3 M	com.halo.wifikey.wifilocating
Touch VPN	4.7 M	com.northghost.touchvpn
Firefox	40.1 M	org.mozilla.firefox
AliExpress Shopping App∗	10 M	com.alibaba.aliexpresshd
Clean Master	16.6 M	com.cleanmaster.mguard
WiFi Toolbox∗	7.7 M	mobi.wifi.toolbox

5.3 Performance

For NativeProtector, a security framework which takes process isolation and system call interception, the runtime overhead depends greatly on the intensity of context switches and API invocations we intercepted (like *connect()*). In section 5.2, not mach delay is felt when testing in the real-world applications hardened by NativeProtector. To quantify the performance overhead, we talk about the NativeProtector's performance overhead in the extreme cases. We crafted two artificial application to represent the two extreme cases. One is testing the influence of the intensity of context switches on the NativeProtector's performance

overhead, and the other is testing the NativeProtector's overhead affected by the API invocations we intercepted.

Firstly, we crafted an application which compresses a medium size file (about 5.6MB) stored on the phone with the popular Zlib library. The Java code of the application divides this file to small segments and passes one to Zlib which perform the compression. Then the Java code passes the next one to Zlib. When Zlib library is sandboxed by NativeProtector, the segments size has a great impact on the performance overhead, as small segment size means frequent context switches between Java and native code. Table 2 presents the overhead introduced by enabling NativeProtector in Zlib benchmark application that compresses the file with different segment size. All the performance time is the average over 5 tests. The results table illustrates that higher overhead comes up with smaller segments size. And the overhead can be 81.79% when the segments size is 1KB. But in the real world, few applications will perform context switches as frequently as this test. They utilize native libraries to perform CPU-intensive operations like image compression and conversion, without frequently making context switches. Thus NativeProtector brings acceptable overhead to the real-world applications.

Table 2. Performance on Zlib benchmark application

Segments Size	Without NativeProtector	With NativeProtector	Overhead
1KB	21905ms	39822ms	81.79%
2KB	15093ms	23948ms	58.67%
4KB	10585ms	15507ms	46.50%
8KB	8993ms	11399ms	26.75%
16KB	8494ms	9585ms	12.84%

Secondly, we crafted an application which use native libraries to connect to a remote server and send a message. This application tries to connect the remote sever and send message ten times. After these operations are done, it logs the time consumed. We tested both the original application and the application hardened by NativeProtector for five times against the randomness. Table 3 shows the result of this test. We can see that NativeProtector introduces an overhead of 50% in this test, which we believe is acceptable as real-world applications will not invoke the APIs NativeProtector intercepted frequently.

In summary, the evaluation demonstrates that NativeProtector can enforce security on the native libraries in Android applications with acceptable overhead in most real-world applications where context switches between Java and native code are not frequent and the API invocations we intercepted are not frequently used.

Table 3. Performance on API invocations

	First test	Second test	Third test	Forth test	Fifth test	Average
Without NP	608ms	543ms	512ms	438ms	619ms	544ms
With NP	960ms	743ms	788ms	672ms	916ms	816ms
Overhead	50%					

6 Related Work

6.1 Android App Security

With the increasing popularity of Android and the increasing malware threats, many approaches to secure Android applications have been proposed. Some of them extend the Android framework to perform fine-grained control of Android applications at runtime [10, 11, 14, 18, 22, 28]. AppFence [14] retrofits the Android operating system to return mock data of the sensitive resources to the imperious applications. CRePE [11] allows both the user and authorized third parties to enforce fine-grained context-related polices. Deepdroid [22] intercepts Binder transactions and traces system calls to provide portability and fine-grained control. The other approaches perform security enforcement at the application layer[8, 9, 12, 17, 23, 27]. Aurasium [23] uses native library interposing to enforce arbitrary policies at runtime. Appcage [27] hooks into the application's Dalvik virtual machine instance to sandbox the application's dex code and uses SFI (Software Fault Isolation) to sandbox the application's native code. Boxify [9] combines the strong security guarantees of OS security extensions with the deployability of application layer solutions by building on isolated processes to restrict privileges of untrusted applications and introducing a novel application virtualization environment. NativeProtector takes the application layer approach. It can run in the user's phone and does not need to modify the Android framework, so it can be easly deployed.

6.2 Untruseted code Isolation

Android's UID-based sandboxing mechanism strictly isolates different applications in different processes. With this strong security boundary naturally supported by Android, many approaches have been proposed to isolate untrusted code in another process [15, 19, 21, 24]. Dr. Android and Mr. Hide [15] revokes all Android platform permissions from the untrusted applications and applies code rewriting techniques to replace well-known security-sensitive Android API calls in the monitored application with calls to the separate reference monitor application that acts as a proxy to the application framework. AdSplit [19] and AFrame [24] isolate the third-party advertising libraries which is the JAR format into separate processes. NativeGuard [21] isolates native libraries into a non-privileged application. But the benign native libraries which need permission to perform legal task can not work any more, because it lacks the fine-grained

access control of the native libraries. NativeProtector adopts the idea of isolation, and instruments the native libraries to take fine-grained access control of native libraries.

7 Conclusion

We have presented the design, implementation and the evaluation of NativeProtector, a system to regulate the third-party native libraries in Android applications. NativeProtector separates the native libraries and generated hooking libraries to another server app, and the rest part of the original app is generated as the client app. The client app binds to the server app at the launching time, and all native function calls are replaced with the IPCs to the server app. The hooking libraries intercept system calls in server app to enforce security of native libraries in the server app. We have implemented a prototype of NativeProtector. Our evaluation shows that NativeProtector can successfully regulate the third-party native libraries in Android applications and introduces acceptable overhead.

References

1. Android Interface Definition Language (AIDL). http://developer.android.com/intl/zh-cn/guide/components/aidl.html
2. Android NDK. http://developer.android.com/intl/zh-cn/ndk/index.html
3. Android Service. http://developer.android.com/reference/android/app/Service.html
4. apkpure.com. https://apkpure.com
5. Apktool. http://ibotpeaches.github.io/Apktool/
6. Android market share in Q2,2015. http://www.idc.com/prodserv/smartphone-os-market-share.jsp
7. PLT and GOT, the key to code sharing and dynamic libraries. https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html
8. Backes M., Gerling S., Hammer C., et al.: AppGuard–enforcing user requirements on android apps. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 2013: 543-548.
9. Backes M., Bugiel S., Hammer C., et al.: Boxify: Full-fledged app sandboxing for stock Android. In: Proceedings of 24th USENIX Security Symposium (USENIX Security 15), 2015.
10. Bugiel S., Heuser S., Sadeghi A. R.: Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In: Proceedings of 22th USENIX Security Symposium (USENIX Security 13), 2013: 131-146.
11. Conti M., Nguyen V. T. N., Crispo B.: CRePE: Context-related policy enforcement for Android. In: Proceedings of Information Security, 2011: 331-345.
12. Davis B., Sanders B., Khodaverdian A., et al.: I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In: Proceedings of Mobile Security Technologies, 2012.
13. Enck W., Gilbert P., Han S., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of ACM Transactions on Computer Systems (TOCS), 2014, 32(2): 5.

14. Hornyack P., Han S., Jung J., et al.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011: 639-652.
15. Jeon J., Micinski K. K., Vaughan J. A., et al.: Dr. Android and Mr. Hide: fine-grained permissions in android applications. In: Proceedings of the 2nd ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2012: 3-14.
16. Nauman M., Khan S., Zhang X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. ACM, 2010: 328-332.
17. Rasthofer S., Arzt S., Lovat E., et al.: Droidforce: enforcing complex, data-centric, system-wide policies in android. In: Availability, Reliability and Security (ARES), 2014 Ninth International Conference on. IEEE, 2014: 40-49.
18. Russello G., Jimenez A. B., Naderi H., et al.: Firedroid: hardening security in almost-stock android. In: Proceedings of the 29th Annual Computer Security Applications Conference. ACM, 2013: 319-328.
19. Shekhar S., Dietz M., Wallach D. S. AdSplit: Separating Smartphone Advertising from Applications. In: Proceedings of 21th USENIX Security Symposium, 2012: 553-567.
20. Siefers J., Tan G., Morrisett G.: Robusta:Taming the native beast of the JVM. In: Proceedings of the 17th ACM conference on Computer and communications security. ACM, 2010: 201-211.
21. 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. ACM, 2014: 165-176.
22. Wang X., Sun K., Wang Y., et al.: DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In: Proceedings of 22nd Annual Network and Distributed System Security Symposium (NDSS'15). The Internet Society. 2015.
23. Xu R., Sadi H., Anderson R. : Aurasium: Practical Policy Enforcement for Android Applications. In: Proceedings of 21th USENIX Security Symposium, 2012: 539-552.
24. Zhang X., Ahlawat A., Du W. AFrame: isolating advertisements from mobile applications in Android. In: Proceedings of the 29th Annual Computer Security Applications Conference. ACM, 2013: 9-18.
25. Zhang Y., Yang M., Xu B., et al.: Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013: 611-622.
26. Zhao Z, Osono F.C.C.: TrustDroid: Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking. In: MALWARE, 2012, pp. 135143.
27. Zhou Y., Patel K., Wu L., et al.: Hybrid User-level Sandboxing of Third-party Android Apps. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.
28. Zhou Y., Zhang X., Jiang X., et al.: Taming information-stealing smartphone applications (on android). In: Proceedings of the 4th International Conference On Trust and Trustworthy Computing, 2011: pp 93-107.