Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy

Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupe, Mario Polino, Paulo de Geus , Christopher Kruegel, and Giovanni Vigna

Sudeep Nanjappa Jayakumar

Agenda

Introduction

- Google’s Android operating system currently enjoys the largest market share, currently at 84.7%, of all current smartphone operating systems.
- The official app market for Android, the Google Play Store, has around 1.4 million available apps.
- The native code has direct access to the memory of the running process, from this it can completely modify and change the behavior of the Java code.
- An extensive analysis of the native code usage in 1.2 million Android apps. First the static analysis was done on 446k apps using native code and then with the dynamic analysis.

What is Sandboxing?

- Sandbox is a security mechanism for separating running programs. It is often used to execute untested or untrusted programs or code, possibly from unverified or untrusted third parties, suppliers, users or websites, without risking harm to the host machine or operating system.
- A sandbox is implemented by executing the software in a restricted operating system environment, thus controlling the resources (for example, file descriptors, memory, file system space, etc.) that a process may use.

Sandbox Security Relevance

- **Least-Privilege:** The native code of the app should have access only to what is strictly required, thus reducing the chances the native component could extensively damage the system.
- **Compartmentalization:** The native code of the app should communicate with the Java part only using specific, limited channels, so that the native component cannot modify, interact with, or otherwise alter the Java runtime and code in unexpected ways.
- **Usability:** The restrictions enforced by the sandbox must not prevent a significant portion of benign apps from functioning.
- **Performance:** The sandbox implementation must not impose a substantial performance overhead on apps

Contributions

1. A tool is developed to monitor the execution of the native components in android applications and this is used to study the native code usage in the android.
2. The collected data is analyzed and actionable insights are provided in to how the benign apps use the native code . Here the raw data is made available for the community.
3. Finally the results are shown that eliminating permissions of native code is not ideal as the policy would break the apps in the dataset.

Background

To understand the analysis, it is necessary to review the android security mechanisms on how native code is used in android systems, what damage it can cause and previously proposed sandboxing mechanisms.

- Android Security Mechanisms
- Native Code
- Malicious Code
- Native Code Sandboxing mechanisms

Sandboxing Mechanisms

Android Security Mechanisms:

- When apps are installed on an Android phone, they are assigned a new user (UID) and groups (GIDs) based on the permissions requested by the app in its manifest.
- Apps must declare the permissions needed in the manifest, and at installation time the requested permissions are presented to the user, who decides to continue or cancel the installation.

Native Code:

There are four ways in which the Java code of an Android app can execute native code.

1. Exec methods
2. Load methods
3. Native methods
4. Native activity

Sandboxing Mechanisms contd…

Malicious Native code:

- Malicious apps can use native code to hide malicious actions from static analysis of the Java portion of the app.
- Attackers can directly call system calls to execute root exploits is by exploiting vulnerabilities in native code used by benign apps.

Native Code Sandboxing Mechanisms:

- Several approaches have been proposed to sandbox native code execution. For instance NativeGuard and Robusta.
- These approaches move the execution of native code to a separate process.
- Two complementary goals are obtained: (1) the native code cannot tamper with the execution of the Java code and (2) different security constraints can be applied to the execution of the native code.

Analysis Infrastructure

- Design and implementation of a system that dynamically analyzes android applications is used to study the native code.
- Also the native code sandboxing policy is generated automatically.
- Analysis consists an instrumented emulator which records all the events and operations executed within the native code such as invoked syscalls and native to java communication.
- Android system 4.3 is used for the analysis.

Analysis Infrastructure contd…

Static Prefiltering:

- Performing dynamic analysis on all the apps would take more time, so the static analysis was used to filter the apps which had native method, native activity, having a call to exec method, having a call to load method or having an ELF file inside the APK.
- Androguard tool is used for the static analysis, and identify the native methods, it was searched in the dalvik bytecode with the modifier named "native".
- Native activities were identified by two methods:

1. Looking for a NativeActivity in the manifest.
2. Looking for classes declared in the Dalvik bytecode that extend NativeActivity.

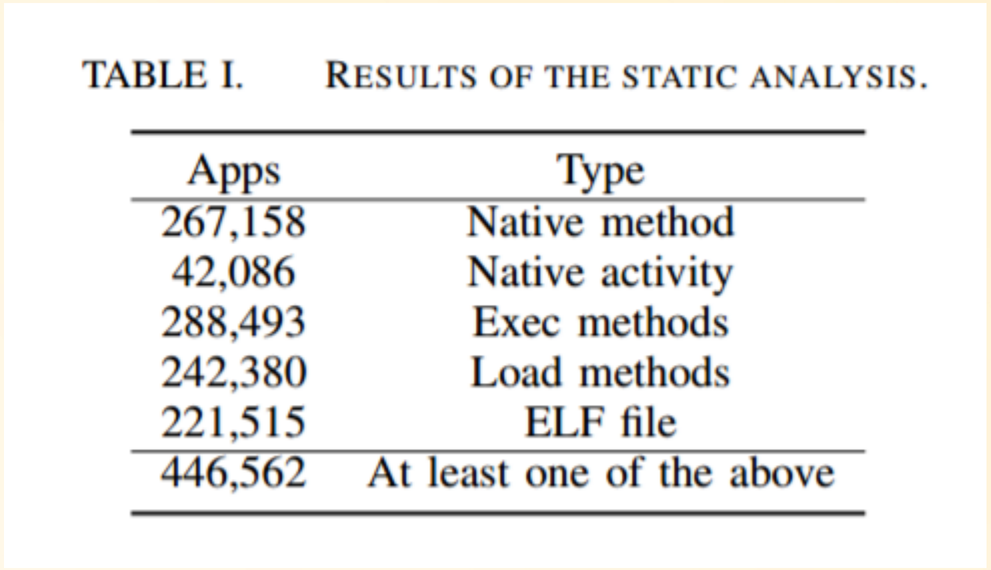


TABLE I. RESULTS OF THE STATIC ANALYSIS.

Apps	Type
267,158	Native method
42,086	Native activity
288,493	Exec methods
242,380	Load methods
221,515	ELF file
446,562	At least one of the above

Analysis Infrastructure contd…

Dynamic Analysis System:

- After identifying the which apps use the native code, now we need to understand how apps use the native code and for this we use dynamic analysis to monitor several types of actions performed by the apps.
- This includes system calls, JNI calls, Binder transactions, calls to Exec methods, loading of third-party libraries, calls to native activities' native callbacks, and calls to native methods. The system calls were captured using the strace tool.
- To monitor JNI calls, calls to native methods, and library loading, the modification to "libdvm" is done.
- Also monitor the amount of data exchanged between native and Java code is done where measuring the amount of data passed in parameters of calls from native code to Java methods and vice versa, as well as the size of the returned value.
- Also the size of the data is captured to set fields in java objects.

Transitions

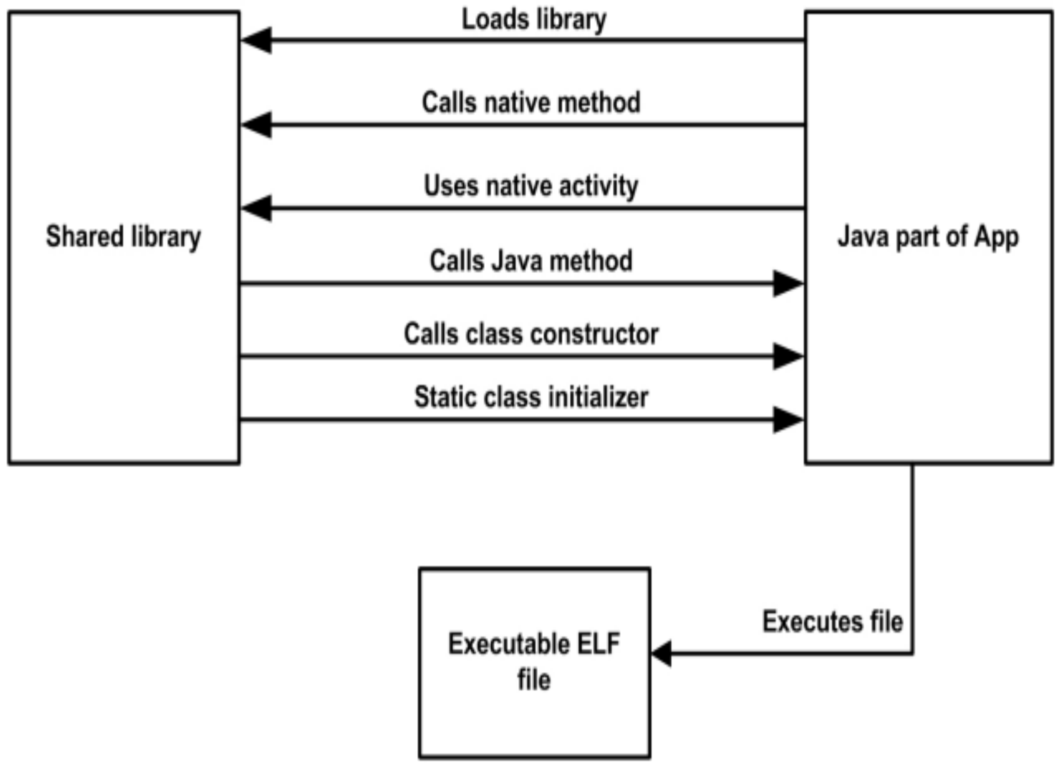


Fig. 1. Possible transitions between native code and Java.

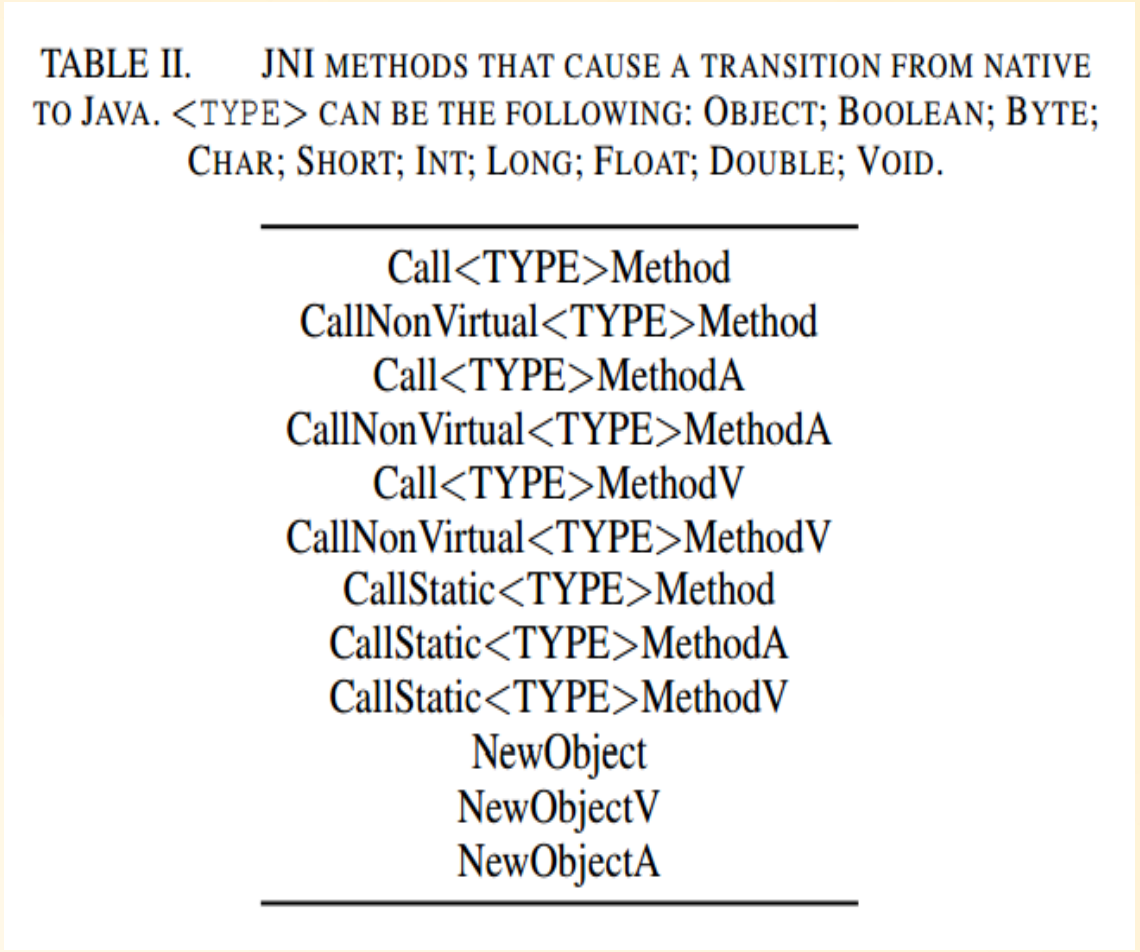


TABLE II. JNI METHODS THAT CAUSE A TRANSITION FROM NATIVE TO JAVA. <TYPE> CAN BE THE FOLLOWING: OBJECT; BOOLEAN; BYTE; CHAR; SHORT; INT; LONG; FLOAT; DOUBLE; VOID.

Call<TYPE>Method
CallNonVirtual<TYPE>Method
Call<TYPE>MethodA
CallNonVirtual<TYPE>MethodA
Call<TYPE>MethodV
CallNonVirtual<TYPE>MethodV
CallStatic<TYPE>Method
CallStatic<TYPE>MethodA
CallStatic<TYPE>MethodV
NewObject
NewObjectV
NewObjectA

Transitions

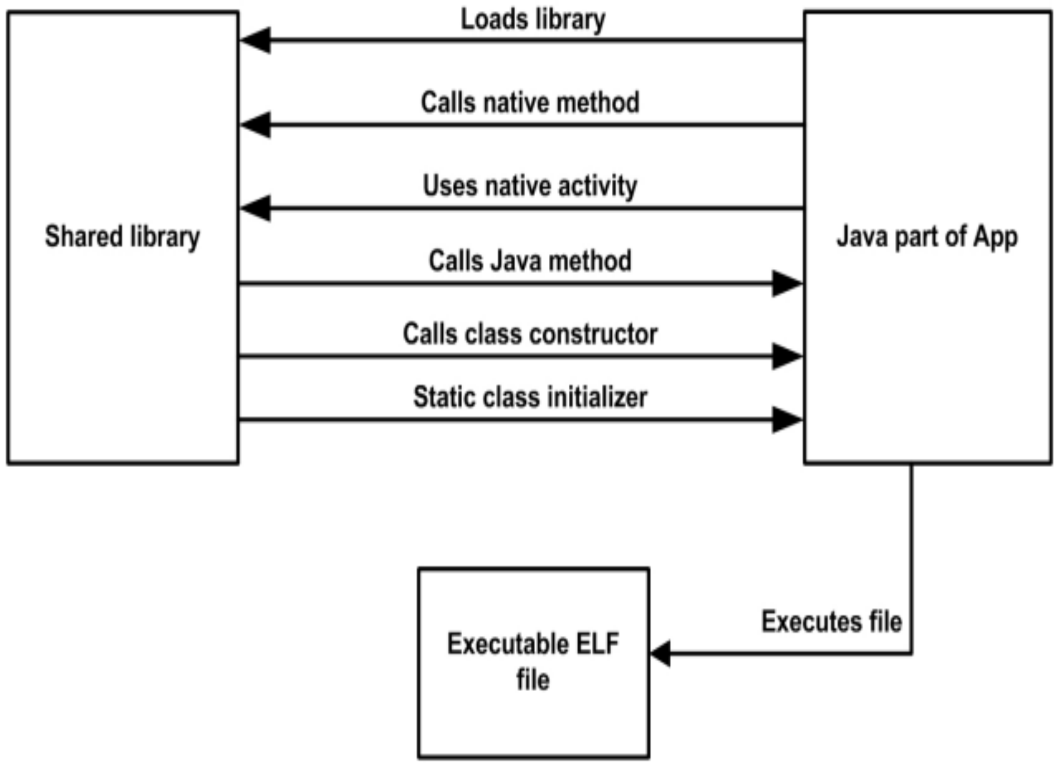


Fig. 1. Possible transitions between native code and Java.

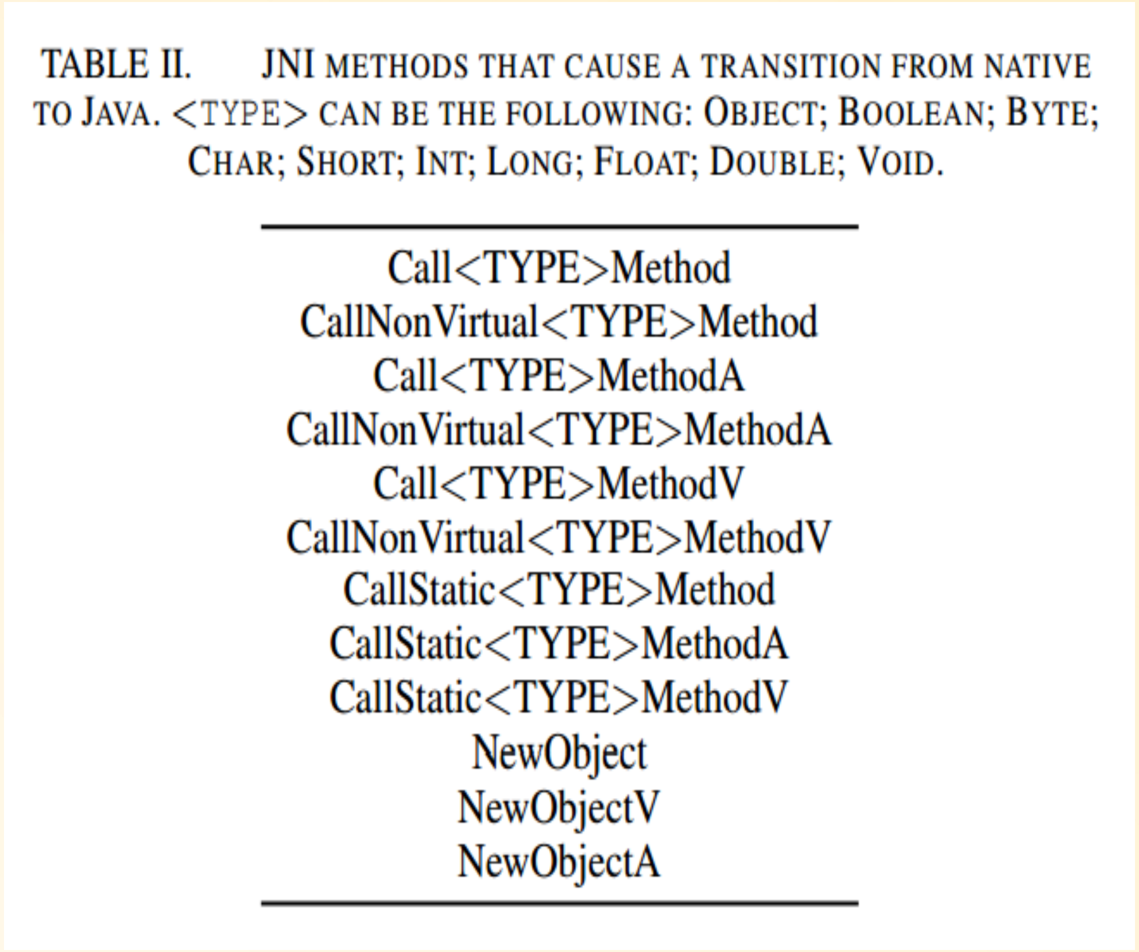


TABLE II. JNI METHODS THAT CAUSE A TRANSITION FROM NATIVE TO JAVA. <TYPE> CAN BE THE FOLLOWING: OBJECT; BOOLEAN; BYTE; CHAR; SHORT; INT; LONG; FLOAT; DOUBLE; VOID.

Call<TYPE>Method
CallNonVirtual<TYPE>Method
Call<TYPE>MethodA
CallNonVirtual<TYPE>MethodA
Call<TYPE>MethodV
CallNonVirtual<TYPE>MethodV
CallStatic<TYPE>Method
CallStatic<TYPE>MethodA
CallStatic<TYPE>MethodV
NewObject
NewObjectV
NewObjectA

Evaluation & Insights

- Analysis is limited to 2 minutes to keep it feasible and Google Monkey to stimulate the app with random events, and we then automatically generated a series of targeted events to stimulate all activities, services, and broadcast receivers defined in the application.
- During dynamic analysis, 33.6% (149,949) of the apps identified by static analysis as potentially having native code actually executed the native code.
- Also they have manually analyzed statically & dynamically, 20 random apps that were having native code. 8 apps were unreachable from the java code and the remaining apps too complex to manually inspect.

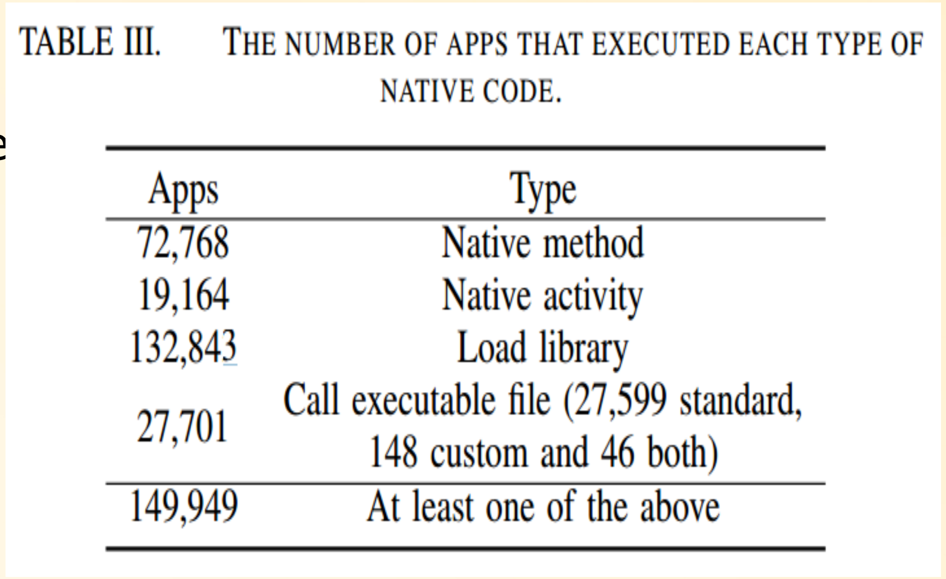


TABLE III. THE NUMBER OF APPS THAT EXECUTED EACH TYPE OF NATIVE CODE.

Apps	Type
72,768	Native method
19,164	Native activity
132,843	Load library
27,701	Call executable file (27,599 standard, 148 custom and 46 both)
149,949	At least one of the above

Native code Behavior

- The actions were split into those performed by shared libraries (including those performed during library loading, native methods, and native activities) and those that are the result of invoking custom, executable, and binaries through Exec methods.
- They have also presented the actions performed using standard binaries (i.e., not created by the app), but in this case based on their names and parameters, instead of looking at the system calls.

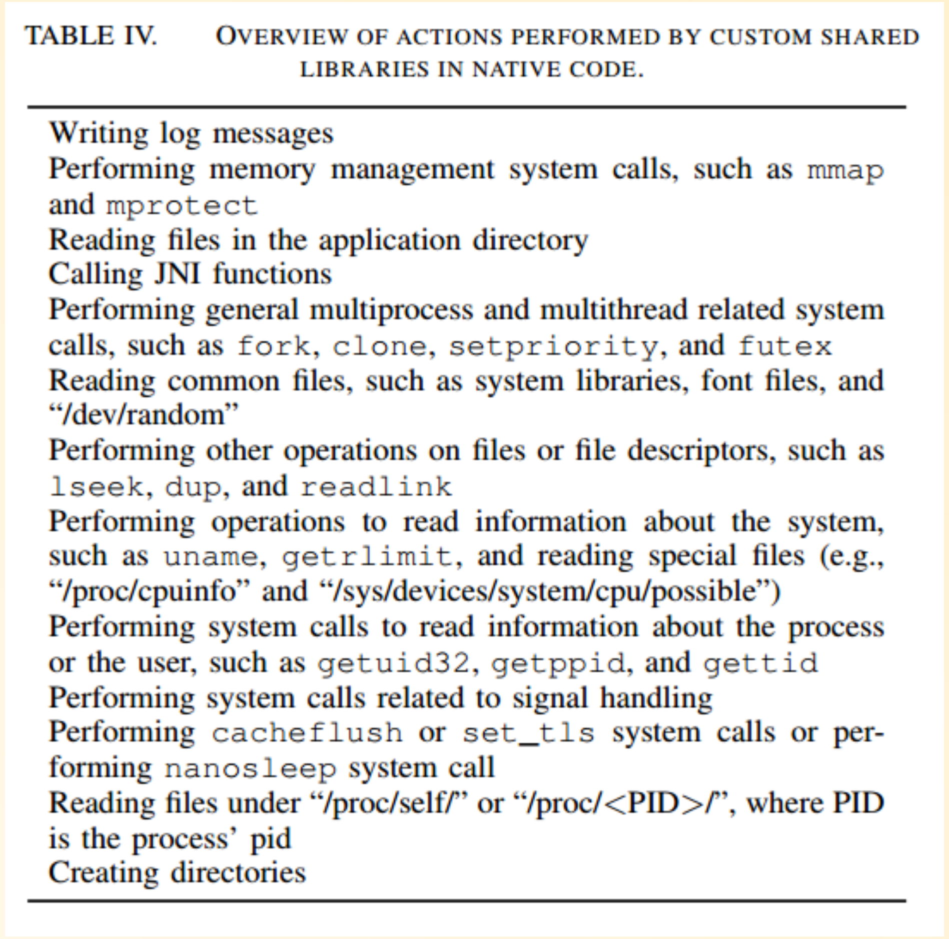


TABLE IV. OVERVIEW OF ACTIONS PERFORMED BY CUSTOM SHARED LIBRARIES IN NATIVE CODE.

Writing log messages
Performing memory management system calls, such as `mmap` and `mprotect`
Reading files in the application directory
Calling JNI functions
Performing general multiprocess and multithread related system calls, such as `fork`, `clone`, `setpriority`, and `futex`
Reading common files, such as system libraries, font files, and "/dev/random"
Performing other operations on files or file descriptors, such as `lseek`, `dup`, and `readlink`
Performing operations to read information about the system, such as `uname`, `getrlimit`, and reading special files (e.g., "/proc/cpuinfo" and "/sys/devices/system/cpu/possible")
Performing system calls to read information about the process or the user, such as `getuid32`, `getppid`, and `gettid`
Performing system calls related to signal handling
Performing `cacheflush` or `set_tls` system calls or performing `nanosleep` system call
Reading files under "/proc/self/" or "/proc/<PID>/", where PID is the process' pid
Creating directories

Native code Behavior

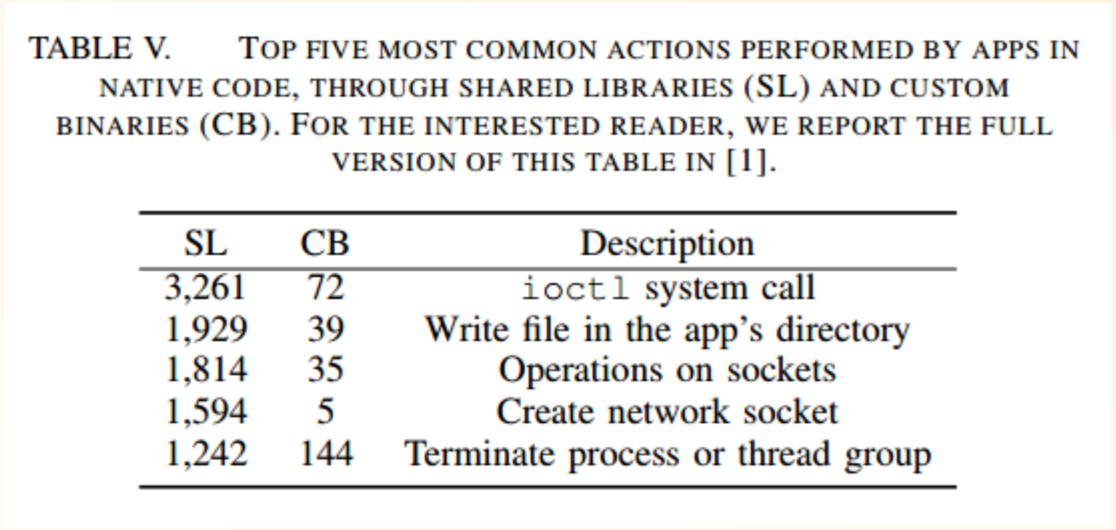


TABLE V. TOP FIVE MOST COMMON ACTIONS PERFORMED BY APPS IN NATIVE CODE, THROUGH SHARED LIBRARIES (SL) AND CUSTOM BINARIES (CB). FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

SL	CB	Description
3,261	72	`ioctl` system call
1,929	39	Write file in the app's directory
1,814	35	Operations on sockets
1,594	5	Create network socket
1,242	144	Terminate process or thread group

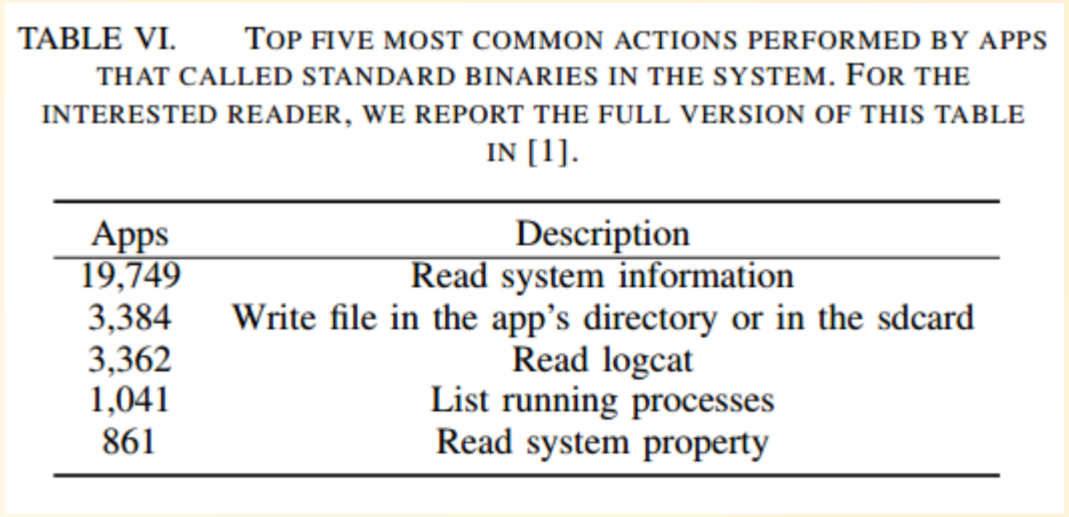


TABLE VI. TOP FIVE MOST COMMON ACTIONS PERFORMED BY APPS THAT CALLED STANDARD BINARIES IN THE SYSTEM. FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

Apps	Description
19,749	Read system information
3,384	Write file in the app's directory or in the sdcard
3,362	Read logcat
1,041	List running processes
861	Read system property

Native code Behavior

- Around 3,669 apps that perform an action requiring Android permissions from native code.
- The below table presents the top five most popular permissions used, how many apps use them, and how we detected its use.

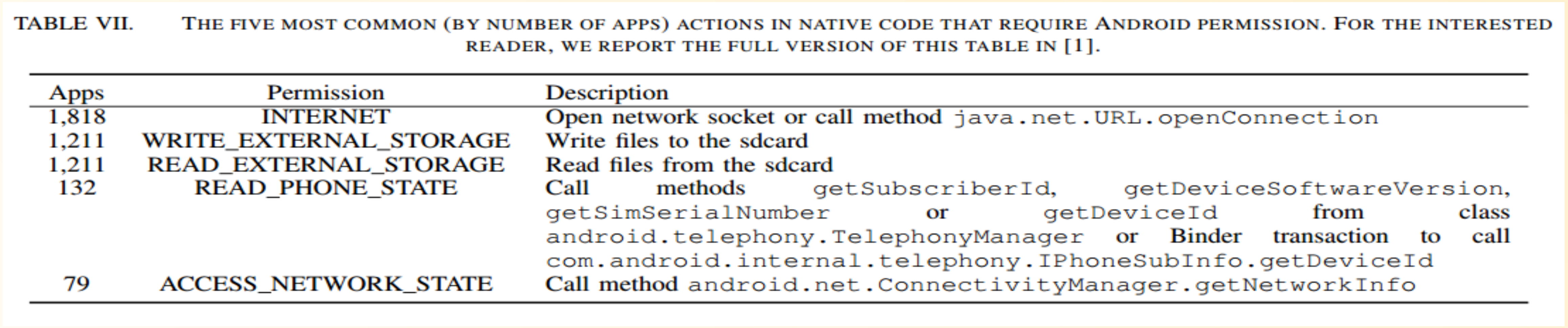


TABLE VII. THE FIVE MOST COMMON (BY NUMBER OF APPS) ACTIONS IN NATIVE CODE THAT REQUIRE ANDROID PERMISSION. FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

Apps	Permission	Description
1,818	INTERNET	Open network socket or call method `java.net.URL.openConnection`
1,211	WRITE_EXTERNAL_STORAGE	Write files to the sdcard
1,211	READ_EXTERNAL_STORAGE	Read files from the sdcard
132	READ_PHONE_STATE	Call methods `getSubscriberId`, `getDeviceSoftwareVersion`, `getSimSerialNumber` or `getDeviceId` from class `android.telephony.TelephonyManager` or Binder transaction to call `com.android.internal.telephony.IPhoneSubInfo.getDeviceId`
79	ACCESS_NETWORK_STATE	Call method `android.net.ConnectivityManager.getNetworkInfo`

- we can draw two important conclusions:

1. If the native code is separated in a different process, it is necessary to give some permissions to the native code.
2. The permissions of the native code can be more strict (less permissive) than the permissions of the Java code.

Java-Native Code Interactions

- For better understanding native code from the Java code of the apps, they have measured the number of interactions per millisecond between Java and native code, i.e., the number of calls to JNI functions, calls to native methods, and Binder transactions.
- The mean of interactions per millisecond is 0.00142, whereas the variance is 0.00003 and the maximum value is 0.22. NativeGuard's performance evaluation with the Zlib benchmark shows a 34.36% runtime overhead for 9.81 interactions per millisecond and 26.64% for 3.96 interactions per millisecond.
- Additionally, they have measured the number of bytes exchanged between the Java code and native code per second. The mean of bytes exchanged per second is 1,956.55 (1.91 KB/s) and the maximum value is 6,561,053.27 (6.26 MB/s).
- Only 11 apps exchanged more than 1 MB/s.
- The amount of data exchanged between java and native code would not incur a significant overhead.

Usage of the su Binary

- To have great control over the system, the users need to perform rooting in order to perform few actions such as uninstalling the pre-installed apps.
- Some of these apps use the "-c" argument of su to specify a command to be executed as root.
- These actions did not work properly during dynamic analysis, so we cannot obtain more information on their behavior.

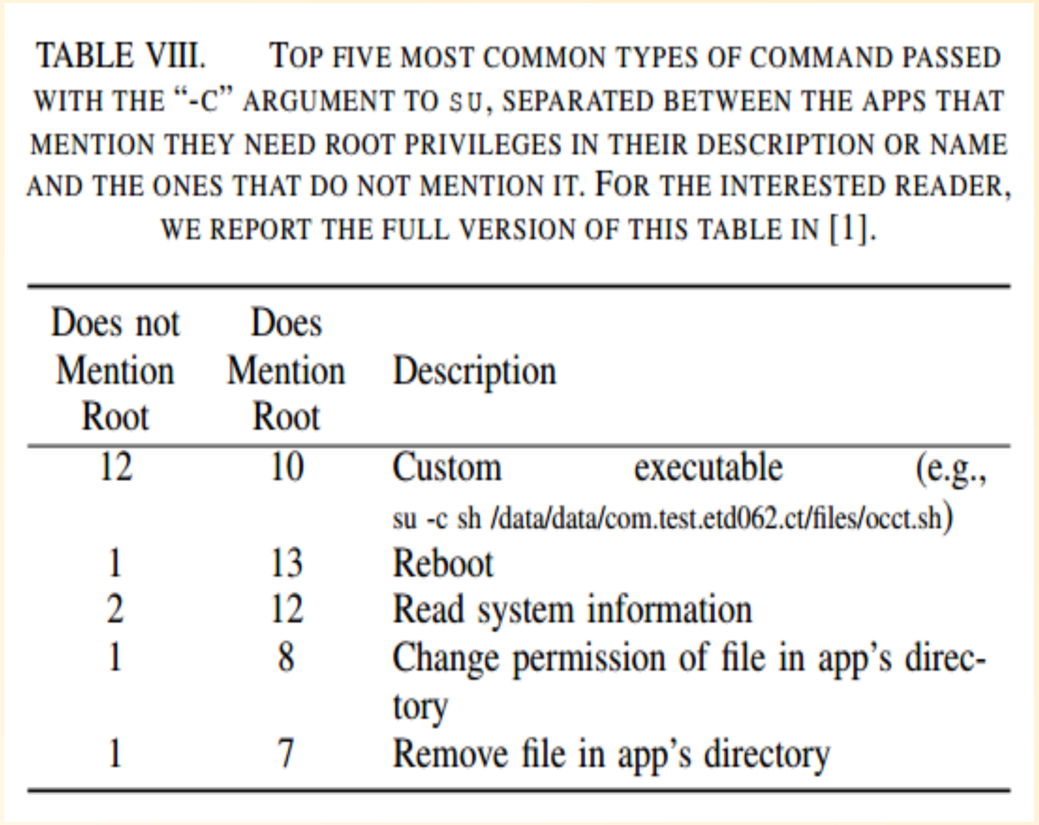


TABLE VIII. TOP FIVE MOST COMMON TYPES OF COMMAND PASSED WITH THE "-C" ARGUMENT TO SU, SEPARATED BETWEEN THE APPS THAT MENTION THEY NEED ROOT PRIVILEGES IN THEIR DESCRIPTION OR NAME AND THE ONES THAT DO NOT MENTION IT. FOR THE INTERESTED READER, WE REPORT THE FULL VERSION OF THIS TABLE IN [1].

Does not Mention Root	Does Mention Root	Description
12	10	Custom executable (e.g., su -c sh /data/data/com.test.etd062.ct/files/occt.sh)
1	13	Reboot
2	12	Read system information
1	8	Change permission of file in app's directory
1	7	Remove file in app's directory

JNI Calls Statistics

This table presents the types of JNI functions that were used by the apps and how many apps used them.

This table presents what groups of methods from the framework were called, along with the amount of apps that called methods in each group.

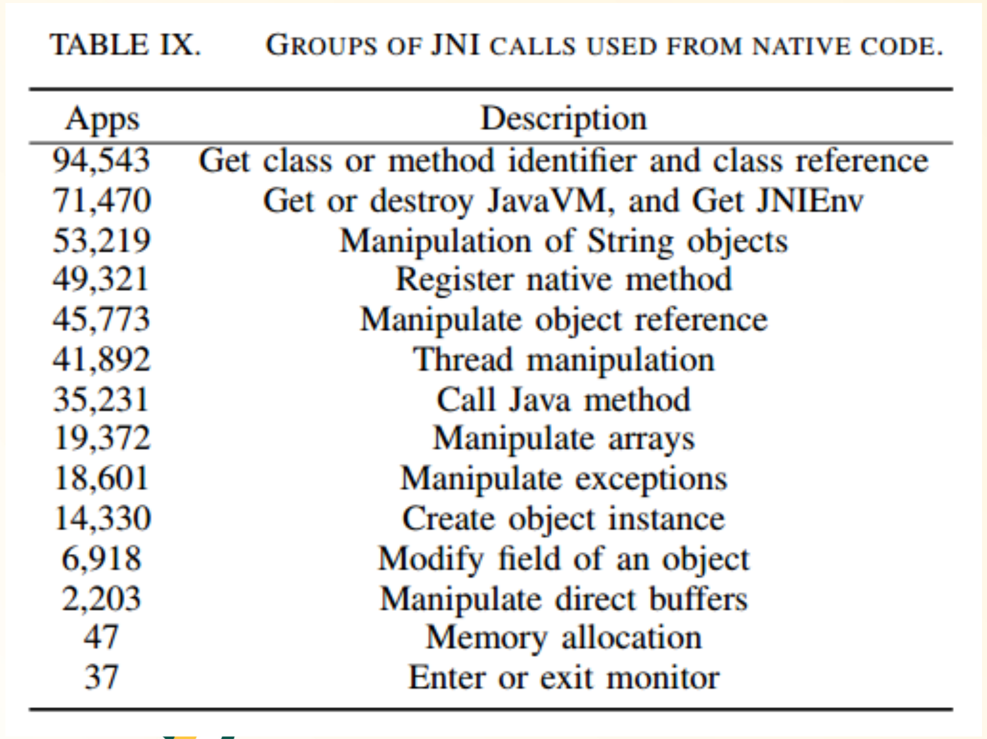


TABLE IX. GROUPS OF JNI CALLS USED FROM NATIVE CODE.

Apps	Description
94,543	Get class or method identifier and class reference
71,470	Get or destroy JavaVM, and Get JNIEnv
53,219	Manipulation of String objects
49,321	Register native method
45,773	Manipulate object reference
41,892	Thread manipulation
35,231	Call Java method
19,372	Manipulate arrays
18,601	Manipulate exceptions
14,330	Create object instance
6,918	Modify field of an object
2,203	Manipulate direct buffers
47	Memory allocation
37	Enter or exit monitor

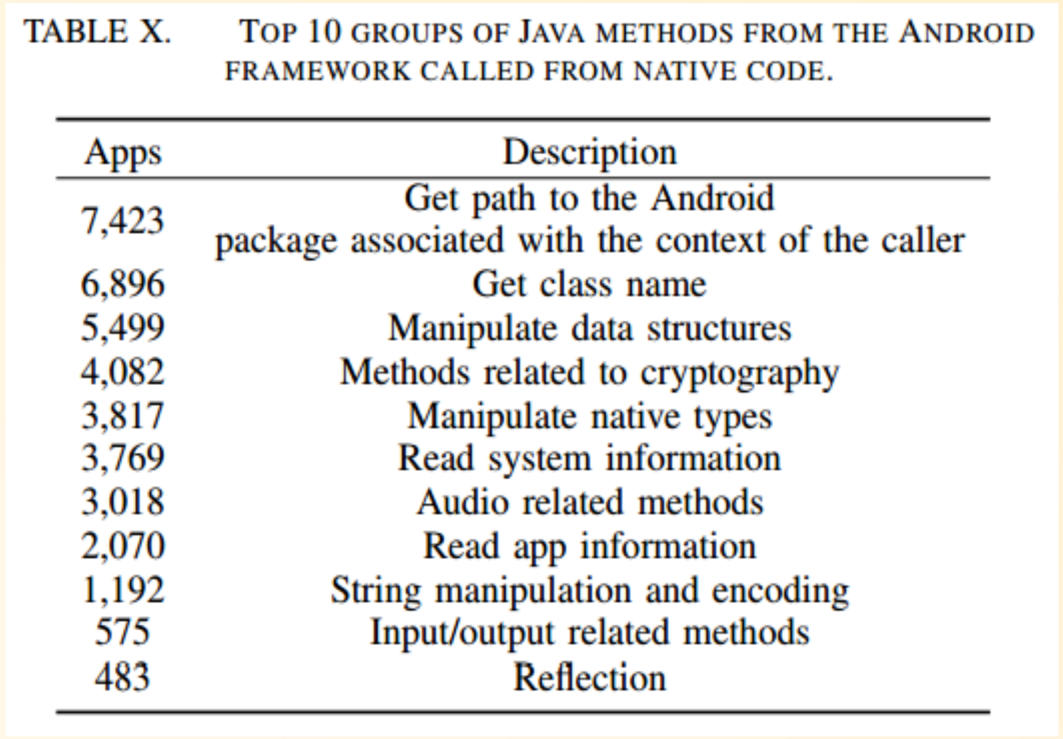


TABLE X. TOP 10 GROUPS OF JAVA METHODS FROM THE ANDROID FRAMEWORK CALLED FROM NATIVE CODE.

Apps	Description
7,423	Get path to the Android package associated with the context of the caller
6,896	Get class name
5,499	Manipulate data structures
4,082	Methods related to cryptography
3,817	Manipulate native types
3,769	Read system information
3,018	Audio related methods
2,070	Read app information
1,192	String manipulation and encoding
575	Input/output related methods
483	Reflection

Binder Transactions

- 1.64% (2,457) of the apps that reached native code during dynamic analysis performed Binder transactions.
- The most common class remotely invoked by this process is IServiceManager, which can be used to list services, add a service, and get an object to a Binder interface.
- All apps that used this class obtained an object to a Binder interface and two apps also used it to list services. This data shows that using Binder transactions from native code is not common.

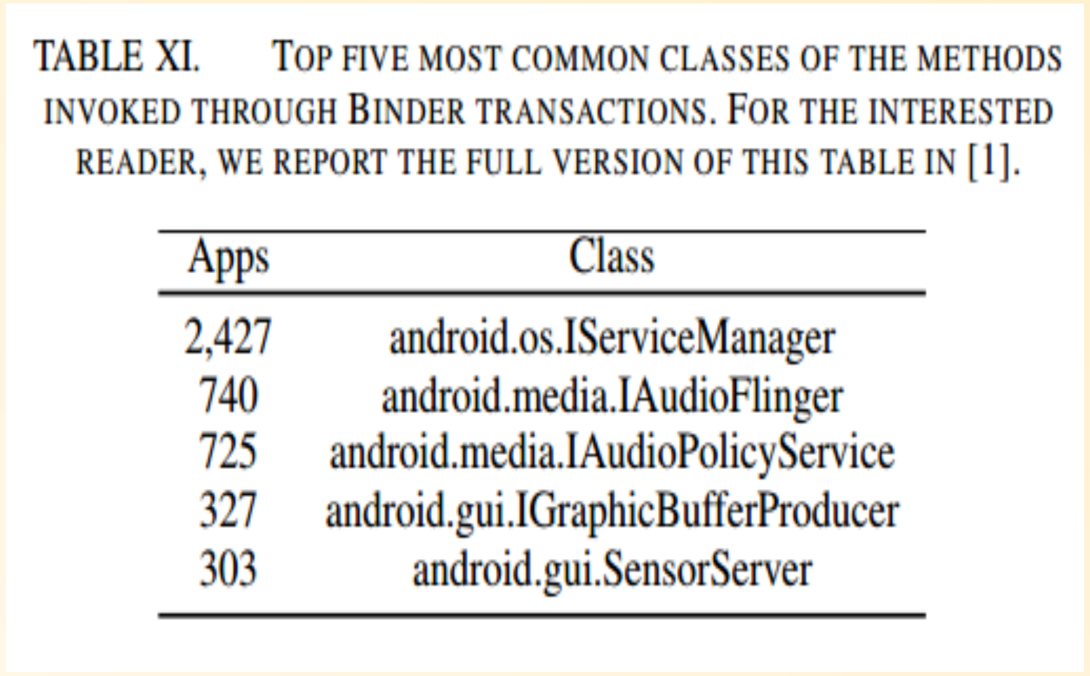


TABLE XI. Top five most common classes of the methods invoked through Binder transactions. For the interested reader, we report the full version of this table in [1].

Apps	Class
2,427	android.os.IServiceManager
740	android.media.IAudioFlinger
725	android.media.IAudioPolicyService
327	android.gui.IGraphicBufferProducer
303	android.gui.SensorServer

Usage of External Libraries

16.6% (24,942) of the apps that reached native code, no standard library was used by a great number of apps.

Several custom libraries were used by more than 7.5% of the apps that executed native code.

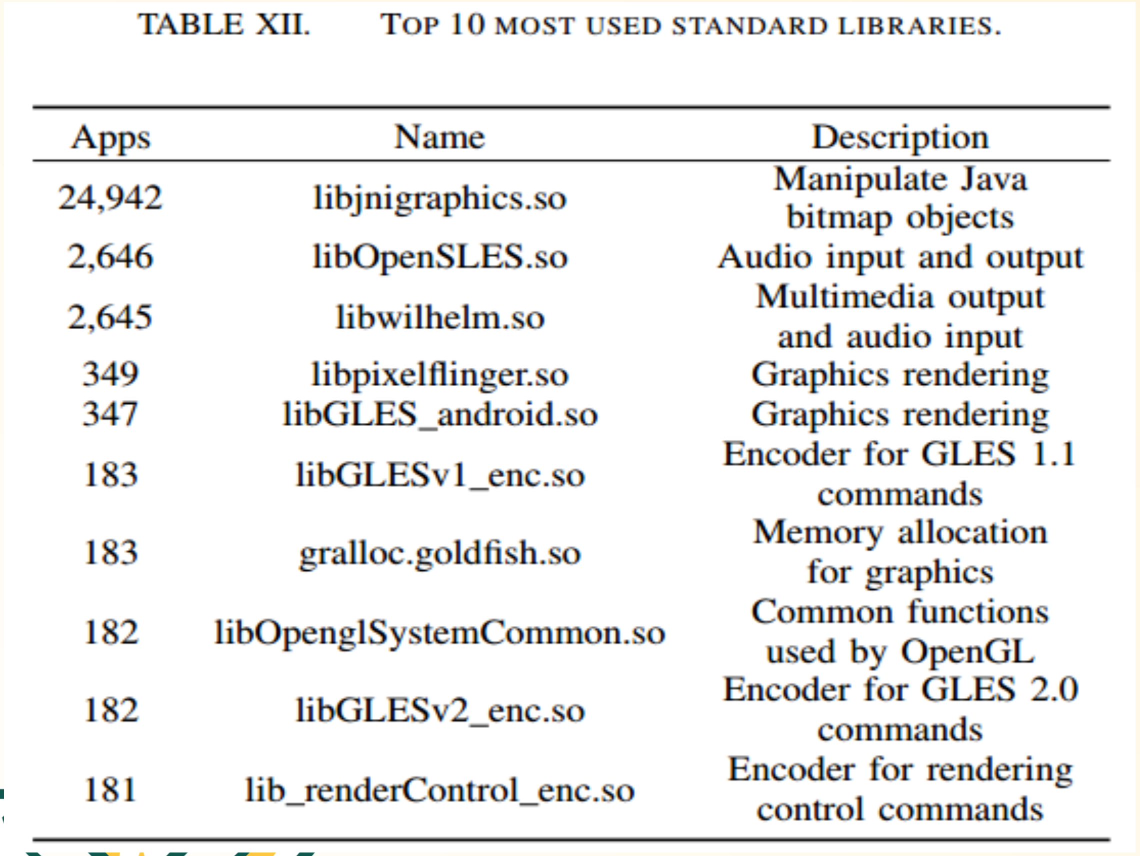


TABLE XII. TOP 10 MOST USED STANDARD LIBRARIES.

Apps	Name	Description
24,942	libjnigraphics.so	Manipulate Java bitmap objects
2,646	libOpenSLES.so	Audio input and output
2,645	libwilhelm.so	Multimedia output and audio input
349	libpixelflinger.so	Graphics rendering
347	libGLES_android.so	Graphics rendering
183	libGLESv1_enc.so	Encoder for GLES 1.1 commands
183	gralloc.goldfish.so	Memory allocation for graphics
182	libOpenglSystemCommon.so	Common functions used by OpenGL
182	libGLESv2_enc.so	Encoder for GLES 2.0 commands
181	lib_renderControl_enc.so	Encoder for rendering control commands

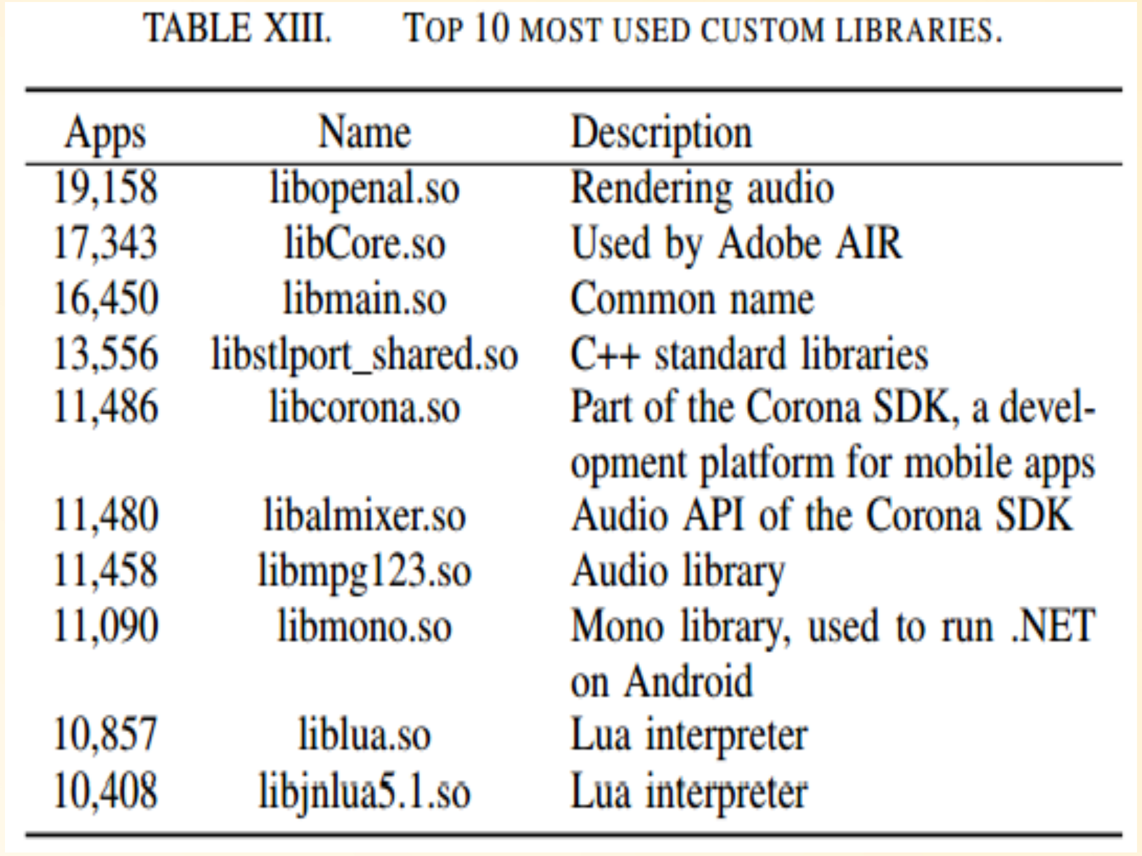


TABLE XIII. TOP 10 MOST USED CUSTOM LIBRARIES.

Apps	Name	Description
19,158	libopenal.so	Rendering audio
17,343	libCore.so	Used by Adobe AIR
16,450	libmain.so	Common name
13,556	libstlport_shared.so	C++ standard libraries
11,486	libcorona.so	Part of the Corona SDK, a development platform for mobile apps
11,480	libalmixer.so	Audio API of the Corona SDK
11,458	libmpg123.so	Audio library
11,090	libmono.so	Mono library, used to run .NET on Android
10,857	liblua.so	Lua interpreter
10,408	libjnlua5.1.so	Lua interpreter

Security Policy Generation

- One of the main step to limit the possible damage that native code can do is to isolate it from the Java code using the native code sandboxing mechanisms.
- Here we propose to use the dynamic analysis system to generate security policies which means the normal behavior of the applications.
- This dynamic analysis has two modes:

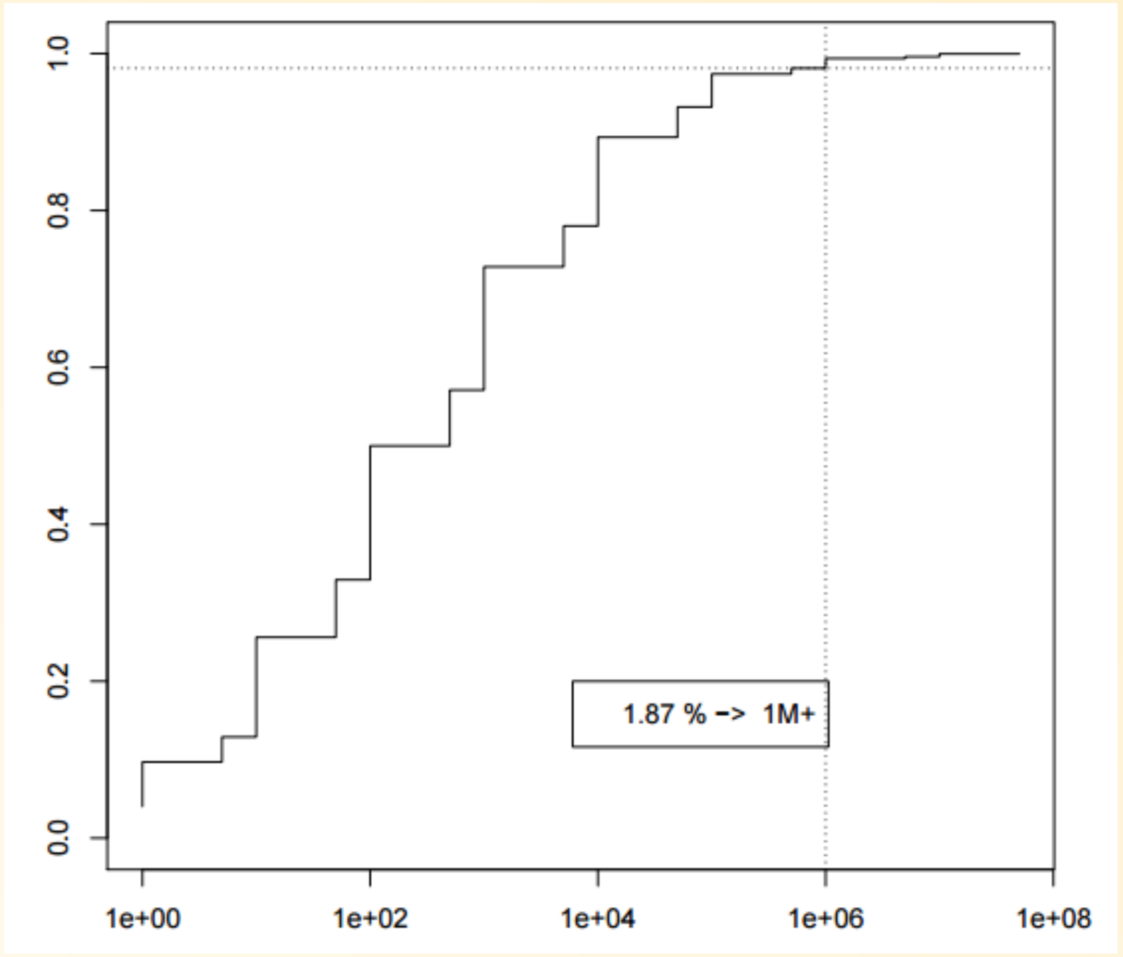
1. Permissive mode:

   In this mode the system would log and report the usage of unusual behavior.
2. Enforcing mode:

   The system would block the execution of unusual behavior and stop the application.

Impact of Security Policies

- To understand the impact of implementation they analyzed the popularity (lower number of installations) of the apps whose behavior seen during the dynamic analysis would be blocked.
- Among the applications for which the policy would block at least one behavior that has been executed at runtime, 1.87% (51) of them have more than 1 million installations.



Impact of Security Policies contd..

- They identified three types of suspicious activities among these apps.

1. **Ptrace:**

280 apps used ptrace. 276 of these only call ptrace to trace itself without checking the result. Developers do this on purpose because app cannot be traced by another process.

2. **Modifying Java code:**

Identified 7 apps that modify the Java section of the app from native code. All these apps perform this action from the library libAPKProtect.so. It harder for reverse engineering tools to decompile the app.

3. **Fork and inotify:**

57 apps were identified that create a child process in native code and use inotify to monitor the apps' directory, in order to identify when they are uninstalled

Limitations

1. The policies that the tool generate might not be complete they might block more applications when adopted at large-scale, and the performance overhead of isolating native code could be higher, using a more-sophisticated instrumentation tool could possibly improve the amount of native code behavior.

   Deploying the automatically generated policies in a native sandbox with reporting mode would help to observe the behaviors that the policies would block.
2. Another limitation is that the authors approach restricts access to permissions from native code, but it still allows the native code to invoke (some) Java methods. This would drastically reduce the possibility of introducing malicious behaviors.
3. The authors are not completely certain that there are no malicious apps in the dataset depending on how the malware works.
4. The tracing system slows down the execution of the apps by around 10 times. There were only small subset of apps run and analyzed i.e 177 apps.

Related Work

Large Measurement Studies:

- Viennot et al. did a large measurement study on 1,100,000 applications crawled from the Google Play app store. They measured the frequency with which Android applications make use of native code components.
- Lindorfer et al: They analyzed 1,000,000 apps, of which 40% are malware. Authors used Andrubis, a publicly-available analysis system for Android apps that combines static and dynamic analysis.

Application Analysis Systems:

- Several systems have already been used in this paper for analysis.

Protection Systems:

- Fedler et al: proposed a system where a root t exploits by preventing apps from giving execution permission for custom executable files and by introducing a permission related to the use of the System class.

Native Code Isolation:

- There are lot of systems in order to isolating the native code Klinkoff et al. [26] focus on the isolation of .NET applications, whereas Robusta [33] focuses on the isolation of native code used by Java applications

Conclusion

- Developers are allowed to mix Java code and native code enables developers to fully harness the computing power of mobile devices but this feature does more harm than doing good.
- Native code sandboxing is the e correct approach to properly limit its potentially malicious side-effects.
- This paper demonstrates an approach to automatically generate an effective and practical native code sandboxing policy.

Thank you