AN ENCRYPTION SCHEME TO MITIGATE BINDER EXPLOITS

DISSERTATION

submitted by

YADU K CB.EN.P2CYS13025

in partial fulfillment for the award of the degree

of

MASTER OF TECHNOLOGY

IN

CYBER SECURITY

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

July 2015

AN ENCRYPTION SCHEME TO MITIGATE BINDER EXPLOITS

DISSERTATION

submitted by

YADU K CB.EN.P2CYS13025

in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY

IN

CYBER SECURITY

Under the guidance of

Prof. Prabhaker Mateti
Associate Professor
Computer Science and Engineering
Wright State University, USA

TIFAC-CORE IN CYBER SECURITY
AMRITA SCHOOL OF ENGINEERING
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112

July 2015

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING,COIMBATORE-641112

BONAFIDE CERTIFICATE

This is to certify that this dissertation entitled **"AN ENCRYPTION SCHEME TO MITIGATE BINDER EXPLOITS"** submitted by **YADU K, Reg.No: CB.EN.P2.CYS13025** in partial fulfillment of the requirements for the award of the **Degree of Master of Technology** in **CYBER SECURITY** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering.

Dr. Prabhaker Mateti
(Supervisor)

Dr. M. Sethumadhavan
(Professor and Head)

This dissertation was evaluated by us on .....

INTERNAL EXAMINER

EXTERNAL EXAMINER

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING,COIMBATORE-641112

TIFAC-CORE IN CYBER SECURITY

DECLARATION

I, YADU K, (Reg.No: CB.EN.P2.CYS13025) hereby declare that this dissertation entitled **"AN ENCRYPTION SCHEME TO MITIGATE BINDER EXPLOITS"** is a record of the original work done by me under the guidance of **Prof. Prabhaker Mateti**, Associate Professor, Wright State University, and this work has not formed the basis for the award of any degree / diploma / associateship / fellowship or a similar award, to any candidate in any University, to the best of my knowledge.

Place: Coimbatore

Date: Signature of the Student

COUNTERSIGNED

Dr. M. Sethumadhavan

Professor and Head, TIFAC-CORE in Cyber Security

ACKNOWLEDGEMENTS

At the very outset, I would like to give the first honors to **Amma, Mata Amritanandamayi Devi** who gave me the wisdom and knowledge to complete this dissertation project under her shelter in this esteemed institution.

I express my gratitude to my guide, **Prof. Prabhaker Mateti**, Associate Professor, Computer Science and Engineering, Wright State University, USA, for his valuable suggestion and timely feedback during the course of this dissertation.

I would like to thank **Dr M. Sethumadhavan**, Professor and Head of the TIFAC-CORE in Cyber Security, for giving me useful suggestions and his constant encouragement and guidance throughout the progress of this dissertation.

I express my special thanks to my colleagues who were with me from the starting of the dissertation, for the interesting discussions and the ideas they shared. Thanks also go to my friends for sharing so many wonderful moments. They helped me not only enjoy my life, but also enjoy some of the hardness of this dissertation.

In particular, I would like to thank **Mr. Praveen K**, Assistant Professor, TIFAC-CORE in Cyber Security, Amrita Vishwa Vidyapeetham , Coimbatore and **Mrs. Jevitha K.P**, Assistant Professor, Department of Computer Science, Amrita Vishwa Vidyapeetham , Coimbatore, for providing me with the advice, infrastructure and all the other faculties of TIFAC-CORE in Cyber Security and all other people who have helped me in many ways for the successful completion of this dissertation.

Contents

List of Figures

List of Tables

Abstract

Android is the most popular mobile operating system. It is based on Linux kernel. The popularity and its open source nature makes it a honeycomb for attackers. Binder mechanism handles the entire inter and intra process commination in Android. The Android Binder is coded in Java, C++ and C. Scala is a programming language which is considered as modernized Java. The features supported and consiceness makes it more efficient than Java. In this project, the Binder's Java components are translated into Scala.

Since, all the sensitive application data flows through Binder, subverting it can expose all the communications. Man-in-the-Binder is one such exploit that can subvert the Binder mechanism. This exploit is studied in detail and an encryption scheme is proposed to thwart this attack.

Keywords: Android, Binder, Inter Process Communication, Man-in- the-Binder.

Chapter 1

Introduction

Android is the most widely used mobile OS. With increased exposure comes increased risk. Poorly or maliciously designed applications can compromise the phone and network. While Android defines a base set of permissions to protect phone resources and core applications, it does not define what a secure phone is, relying on the applications to act together securely. Enforcing security through applications have several limitations as applications have only limited privilages. Hence it is required to enforce security at the kernel level. The Android Kernel is mainly written in C, some native applications and libraries in C++ and the rest of the components are written in Java.

Interprocess communication in Android is accomplished by a mechanism known as Binder, that spans over the entire Android Framework. Whether the communication is within the same or between two different applications, it occurs through the Binder. Binder IPC facilitates access to a remote object as if it is a local one. To hide the underlying complex communication mechanism from the applications, the Binder Framework provides an API for the client and server.

Scala is a language which can be considered as next generation Java. It is more concise, less error prone and provides several more features compared to Java. Scala compiles programs to Java Virtual Machine(JVM) byte code. Hence, it is completely inter-operable with Java.

1.1 Problem Statement

Android source code is not well documented. One objective of the project is to document the Android Binder mechanism. Several expolits on Binder are studied and the possible mitigations are suggested. Among such attacks, the Man-in-the-Binder attack is studied in detail.

Man-in-the-Binder is an exploit that subverts the Binder. It was described by [Artenstein and Revivo (2014)]. It is based on the vulnerability that the application data is sent as plaintext between the application and Binder driver. This transaction is intercepted to get confidential application information.

The next task was to identify the Android Binder components in Java and to translate them to Scala.

1.2 Contributions Made

In this project a detailed documentation of Android Binder mechanism is made. The Java components of Binder are translated into Scala. The analysis of the same is done to ensure the equivalence in working. The Man-in-the-Binder attack was studied in detail and a mitigation through encryption is suggested.

1.3 Organization

Chapter 2 describes the background work associated with this project. Chapter 3 is about the JNI and Java Class Loading. A detailed desription of the Binder mechanism is made in chapter 4. Chapter 5 describes the proposed system. Chapter 6 explains how we ensured the equivalence of the Java and the translated Scala files. The various Binder attacks are explained in chapter 7. The Man-in-the-Binder attack is explained in detail, and proposed an encyption scheme to prevent this. Chapter 9 is about the evaluation or the impact of the proposed systems on the Android ROM. Chapter 10 concludes the entire work.

Chapter 2

Background

2.1 Status of Android Security

Android is the most popular mobile OS, with about 78% dominance [International Data Corporation (2015)]. Hence, it is attractive to attackers. Several improvements were made in platform security by Google in 2014 [Google (2014)].

The security enhancements include hardware protected cryptography, SElinux based Mandatory Access Control(MAC), and full disk encryption. From the developers perspective, improved tools were provided to detect and report securiry vulnerabilities. The severity of a vulnerability is classified as critical, high, moderate and low. The critical vulnerabilities are those that are capable of performing remote execution with permission of system. The ability to remotely access data with Android permission corresponds to high severity. Privilege escalation, ADB escalation to root and accessing sensitive data without root privilege are some attacks that comes under moderate potential vulnerability. Unauthorized access to data, Denial of Service that can be stopped by normal user by restarting the system or removing the application, comes under the low level vulnerabilities.

All the apps, prior to getting published in Google Play are reviewed by it. There are over 1 million apps in Google Play and hence identifying all potentially harmful ones is infeasible. Any app that can harm the user, their device, or data is termed as Potentially Harmful Applications(PHA). Before applications become available in Google Play, they undergo an application security review process to confirm that they comply with Google Play policies, prohibiting potentially harmful applications. Google uses machine learning to see patterns for PHA. Millions of data

points, asset nodes, and relationship graphs are analysed to build a high-precision security-detection system. The signals and results are continuously monitored and refined to reduce error rate and improve precision. The potentially harmful applications are classified into the following categories: generic PHA, Phishing, Rooting Malicious, Ransomware, Rooting, SMS Fraud, Backdoor, Spyware, Trojan, Harmful Site, Windows Threat, NonAndroid Threat, WAP Fraud and Call Fraud. If an app falling to any of these category are found installed or is tried to be installed, then an error message is produced by the device.

2.2 A Brief Overview of AOSP Source Code

The Android Open Source Project (AOSP) is aimed at supporting the development of the Android mobile platform. The Android platform includes the operating system (OS), middleware and integral mobile applications. The June 2015 verison of AOSP is Lollipop 5.1, with API level 22. The uncompiled source code is about 22GB. The compiled code is about 900MB.

2.3 Scala vs Java

Scala stands for scalable language. The design of Scala started in 2001 by Martin Odersky and was publicly released on 2004 on Java and .NET platform. Scala compiles into Java Virtual Machine byte code. Scala is an integration of object-oriented and functional programming. It is easy to use, concise, safe, fast and universal in nature. It is an object-oriented language in the sense that every value in Scala is an object whose type and behaviour are described by classes.

1. The basic operators, data types and control structures are same for both Scala and Java.
2. Scala has object definitions besides class definitions.
3. The syntax for definition in Scala is `id: type` unlike `type id` in Java.
4. Arrays are like functions in Scala. Hence the i'th element of an array A is indicated as A(i). Scala does not have as special syntax for array types and array accesses.

5. The `void` equivalent in Scala is `unit`.

6. Scala does not have a for-loop. Instead it make use of an expression like:

```
for(var i <- Range) {
      statement (s);
}
```

7. Every class in Scala is a subclass of Scala.AnyRef.

8. Unlike Java, Scala does not require the data type of the variables to defined explicitly. The Scala compiler can infer it from the value assigned to it.

9. Scala is considered to be functional, since every method is considerd as a value.

10. The methods can be passed or received as parameters to functions.

11. Scala allows for the creation of classes without definition, known as abstract classes.

12. Scala has primary and secondary constructors.

13. Scala is more efficient in code reusing. It provides a mixin-class compostion mechanism which allows programmers to reuse all new definitions that are not inherited in the definition of a new class.

14. The Scala compiler compiles the code into JVM class files.

15. Scala supports curried functions, but JVM does not. Hence, functions with multiple parameter sections are eliminated by merging all sections into one. Scala is less verbose than Java. The Scala compiler is smart, it this avoids the developer needing to specify explicitly those things that the compiler can infer.

16. Scala has the ability to take advantage of fuctional programming paradigm and multi-core architecture of current CPUs. Since current CPU development trend is towards adding more cores, rather than increasing CPU cycles, it also favors functional programming paradigm.

17. Scala reduces number of lines from a Java application by making clever use of type inference, treating everything as object, first class functions etc.

18. Scala is designed to express common programming patterns in elegant, concise and type- safe way and makes applying concurrency and parallelism easily.

19. A feature called lazy-evaluation lets deffer computationaly costly operations to be performed only when required.

20. Scala supports operator overloading but Java does not.

21. Though the memory usage is, at a primitive level, the same, Scala's libraries have many powerful methods that can create enormous numbers of objects very easily. The garbage collector in Scala is also more efficient.

2.4 Input Method Editor (IME)

The Man-in-the-Binder attack can be understood more clearly in the context of an example, the standard Input Method Editor (IME). For using IME, an application will register for its callback.The default IME is usually `com.android.inputmethod.latin`. The application will behave as the client and the IME as server. The response from IME contains the key storkes. The reply buffer from IME when parsed contains data in a format as described in Figure[2.1]. The first four bytes of the reply represent the protocol code which are defined in enum binder driver return protocol. `BC REPLY` and `BC TRANSACTION` are the interesting ones here, which indicate the Binder reponse and request respectively [Artenstein and Revivo (2014)]. The Parcel which contains the key stroke information is delivered to an internal interface called `com.android.internal.view.IInputContext`. This interface then sends the data up to the `InputContext` class, for handling the received keyboard inputs. Each time a key is pressed, another callback to `IInputContext` is triggered, and a fresh buffer is sent via Binder. Thus, for a key logging attack, the target is `IInputContext`, the implementation of which is done using AIDL. There is a direct correspondence between the order of the functions in the AIDL file and the function code as it appears in the transaction.

2.5 Unit Testing

To verify the correct working of the Java-to-Scala [Zakrajek (2013)] translated code is a major task. To achieve this, unit testing must be done. The goal of

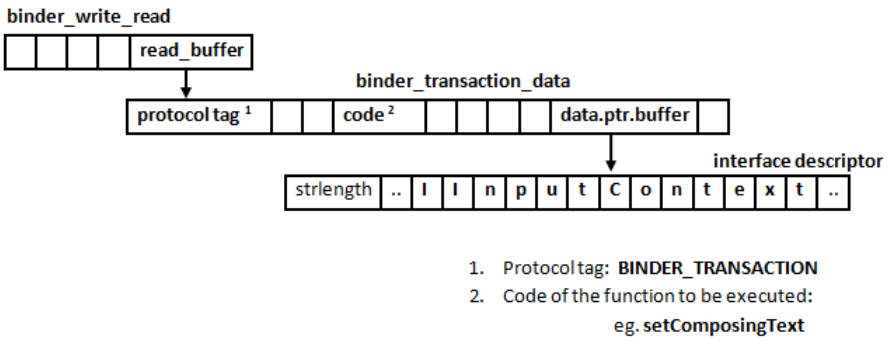

FIGURE 2.1: Input Method Editor

unit testing is to segregate each part of the program and test that the individual parts are working correctly [Osherove (2013)]. This means that for any function or procedure when a set of inputs are given then it should return proper values. It should handle the failures during the course of execution when any invalid input is given. JUnit and ScalaTest are two such unit testing frameworks. Both are available as plugins to the Intellij. They provide support for assertions, exception testing, parameterized tests ets. The test methods are written within a class called Test class. The assert operations are performed within this method.

2.6 Doxygen

Doxygen is a tool used to generate documentation from source code [Doxygen (2013)]. Even if the code is undocumented, doxygen can extract the code structure from them. To help visualize the relations between the code, doxygen will automatically generate dependency graphs and inheritence diagrams. From the documented files in the project, it will automatically generate an HTML documentaion and a reference manual. The control flow diagram for both Java and C++ files can be generated. When it is applied to a project, Doxygen will produce a cross referenced HTML version of the code. This can give a good understanding of the control flow between methods.

2.7 Java to Scala Converter

The `javatoscala` [Zakrajek (2013)] converter aids in the translation of Java files to Scala. The converter make use of the Scalagen Library. It is a Java based parser that performs a modular transformation of the Abstract Syntax Tree to match

Scala idioms(a program expression that is not a built-in feature of that language). The resulting transformed AST is serialized into Scala format. For the creation of AST it makes use of Maven support. Maven is a build automation tool that automates compiling, packaging binary code, do automated tests etc. It will create an XML file that describes the dependencies between modules and components. The required plugins and libraries for Java and Scala are automatically added and compilation and packaging is done.

2.8 Executable and Linkable Format(ELF)

The ELF file has a special format. The structure and the description of the same is given in `/usr/include/linux/elf.h`. The ELF file is divided into different sections. The figure 2.2 shows the format of an ELF file. A Program header table section is present. This table holds the description of each section. The offset to the beginning of this table is present at the ELF header. The sections relevant for the attack are:

- `.symtab` functions and static variables
- `.rel.data` the relocation for static variables for statically linked modules
- `.rel.plt` the list of elements in the PLT (Procedure Linkage Table), which are liable to the relocation during the dynamic linking (if PLT is used)
- `.rel.dyn` the relocation for dynamically linked functions (if PLT is not used)

The `readelf` command will give the header information of the ELF file. `readelf -h libbinder.so` If a program uses the shared library, analysing using `readelf` will give the symbol table(`.synsym`) deatils [Shoumikhin (2010)]. The tables `.rel.dyn` and `.rel.plt` are the tables of relocation for those symbols from .dynsym that need relocation during the linking in general. The table sections `.rel.dyn` and `rel.plt` will show those symbols from .dynsym that need relocation during the linking. By using object dump option of readelf, the assembly code can be obtained for analysing the relocation addresses. All the information about imported and exported functions can be obtained in `.dynsym` section. If the program is coded as position independent code, the calls of the imported functions are performed via Procedure Linkage Table(PLT) and Global Offset Table(GOT). The

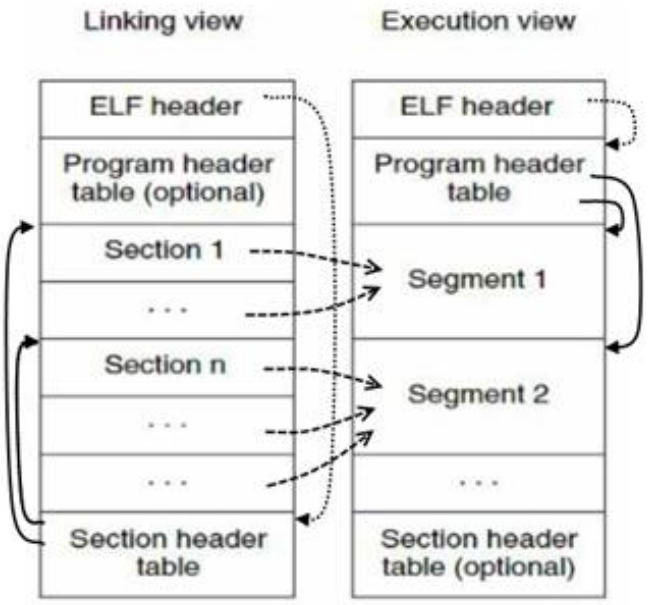

FIGURE 2.2: ELF Header [Shoumikhin (2010)]

relocation will be performed only once for each function and that too for the first instruction of a specific element in PLT. The rel.plt holds the information about such relocation. If the library is compiled without fPIC key, relocations will be performed on the operand of each relative call instruction for each call of some imported function within the code. The .rel.dyn section holds information about such relocations.

2.9 Tiny Encryption Algorithm

The Tiny Encryption Algorithm(TEA) [Alizadeh et al. (2013)] is a lightweight encryption algorithm. The structure of TEA is a Feistel cipher which uses various operations such as ADD, SHIFT, and XOR. The Shannon's twin properties of diffusion and confusion are provided in the TEA, which are necessary for a secure block cipher. The size of data block is 32-bit in this algorithm and the key used is 128-bit.

Chapter 3

JNI and Java Class Loading

3.1 Java Class Loading

Java class loading is used for dynamically loading classes into JVM. In a java application, all classes are loaded using some subclass of `java.lang.classloader`. When a class is loaded, all its references are also loaded [Lindholm et al. (2014)]. There can be unreferenced classes, loaded only when they are referenced. Class loaders in Java are organized into a hierarchy. If a classloader is asked to load a class, it will first check if the class was already loaded. If not loaded, ask parent class loader to load the class. If parent class loader cannot load class, attempt to load it in this class loader. But, in the case of class loaders that support reloading, they wont request the parent to load the class.

There are three default class loaders used in Java, Bootstrap, Extension and System or Application class loader. Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from `rt.jar` and it is the parent of all class loaders in Java. Bootstrap class loader does not have any parents. Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form `jre/lib/ext` directory or any other directory pointed by `java.ext.dirs` system property. Extension ClassLoader in JVM is implemented by `sun.misc.Launcher.ExtClassLoader`. System or Application class loader and it is responsible for loading application specific classes from `class-path` environment variable, `-classpath` or `-cp command` line option or `Class-Path` attribute of `Manifest` file inside JAR.

3.2 Java Native Interface

Java Native Interface(JNI) allows native code (C, C++ etc) to be used within Java programs [Gordon and Essential (1998)]. We wrote sample code to understand JNI. The Binder API interact with the middleware using JNI. The native methods are declared using `native` keyword. Using javah, headers for files containing native code are created. The `javah` generates C header and source files that are needed to implement native methods. The generated header and source files are used by C programs to reference an object's instance variables from native source code. The `.h` file contains a `struct` definition whose layout parallels the layout of the corresponding class. The fields in the `struct` correspond to instance variables in the class. The name of the header file and the structure declared within it are derived from the name of the class.

Steps in using JNI:

1. Within java code declare native methods using `native` keyword.
   `public native void nativeOne()`

2. The shared library to be created must be loaded before the `native` method is called.
   `System.loadLibrary("NativeLib")`

3. `javah -jni NativeMethods`
   NativeMethods is the name of the Java class containing the `native` methods. If the class passed to `javah` is inside a package, the package name is prepended to both the header file name and the structure name. Underscores (_) are used as name delimiters. By default `javah` creates a header file for each class listed on the command line and puts the files in the current directory. Use the `-stubs` option to create source files. Use the -o option to concatenate the results for all listed classes into a single file.

4. Compile C code and create shared library.
   `g++ $JNI_INCLUDE -shared cImplOne.c -o libNativeLib.so`
   The `jni.h`, can be used to support Java data types, objects, and classes in C and C++. Every native method receives a `JNIEnv` pointer as its first parameter, this pointer provides access to the JNI support functions.

5. Run the java program.

Chapter 4

Binder

This chapter gives a detailed explanation of the Binder mechanism, beyond what appears within the AOSP. Linux has several IPC mechanisms such as signals, pipes, sockets, message queues, semaphores, shared memory etc. But, the use of Binder mechanism has several advantages over the other techniques. All filesystem based or filesystem representable IPC mechanisms, such as pipes, lack a location that is writable by all applications. These mechanisms do not have the capability of locating the services required for the Android. Message queues and pipes cannot be used, since they cannot pass file descriptors.

The Binder IPC is basically a client server mechanism. The client request a service and will wait for the server to respond. The request from the client is actually copied from the client address space to the server's address space.

4.1 Binder Mechanism

Binder spans over the application layer, middleware and kernel. The Binder middlware and API are together known as Binder Framework and the Binder driver is known as Binder Kernel. Each higher layer gives an abstraction of the operations on the lower layer as displyed in Figure 4.1. The objective of the Binder IPC is that it must be possible for a process to access an object of another app, as if it is a local one. Every service in Android, has an intreface part and an implementation of the service. Each interface has two implementations, a proxy on the client side and a stub on the server side. When a client process calls a service, the proxy interface of the service gets invoked. The proxy interface will invoke the Binder

calls to the server. At the server, the stub will be waiting for incoming requests, which will then call the actual implementation on the server. The proxy and stub inetrfaces are implemented using Android Interface Definition Language(AIDL).

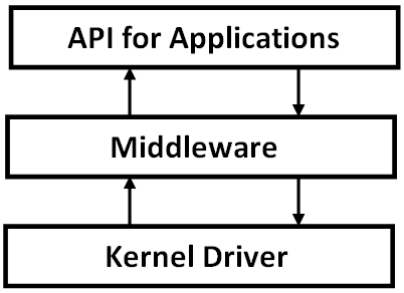

FIGURE 4.1: Binder Layers

The Binder framework API is implemented in Java, the middleware in C++ and the kernel in C. The concept of Binder was derived from OpenBinder, an implementation in Palm OS Cobalt for optimized Inter Process Communication(IPC) [Schreiber (2011)]. In Android, the inter-process and intra-process communication are handled by the Binder. The key implementation goal of Binder is that a process must be capable of accessing a remote object as if it is a local one.

The ServiceManager can use Binder to register as a context manager so as to register or lookup service handles on the go. Serialization of data to be sent is one constraint. Binder transfers data in the form of a structure known as `parcel`. It is marshalled before being sent.

4.2 Binder API

Consider an application S that has a registered itself as a service. Another application A wish to access this service S. The high level communications are shown in Figure 4.2. The `/src` of application S will contain an `.aidl` file, which contains the signatures of the functions that it defines. Once the application is built, the SDK will generate a `.java` file, from the respective `.aidl( for e,g.,: myservice.aidl to myservice.java)`. This `.java` will contain a class named stub. The service will then create an object of the stub and will then define the methods in it. The following code shows how the object of stub(`mBinder`) is created and `mymethod` is the method defined in it.

```
private final myservice.Stub mBinder = new IRemService.Stub() {
```

```
      public int mymethod() {
             return myPid();
       }
}
```

At the client side, it will invoke the `bindService((Intent myservice, Service Connection conn, int flags)`. The argument `ServiceConnection`, is a special method known as a call back method. When `bindService` is called, at the server another method known as `onBind()` is invoked. This method in turn will invoke the `onServiceConnected()` with the `mBinder` object as its argument.

```
public IBinder onBind(Intent intent){
      return mBinder;
}
```

```
IRemoteService mIRemoteService;
private ServiceConnection mCon = new ServiceConnection() {
  public void onServiceConnected(CompName className, IBinder service){
       mRemoteService = IRemoteService.Stub.asInterface(service);
   }
};
```

The service here, is the stub object `mBinder`. We are then casting this Binder object to our copy of stub class object. It is done as follows:

```
mService = IRemoteService.Stub.asInterface(service);
```

With this mService, all the remote methods can be accessed.

4.2.1 Binder at the Middleware

The client, using the `IBinder` object will call the `transact()`. The communication between two processes in Android is known as transaction. The data transmitted is in the form of a structure known as `Parcel`. Hence the parcels needs to be flattened before being sent [Artenstein and Revivo (2014)]. The middleware is responsible for flattening, un-flattening,marshalling, unmarshalling, managing thread pool,

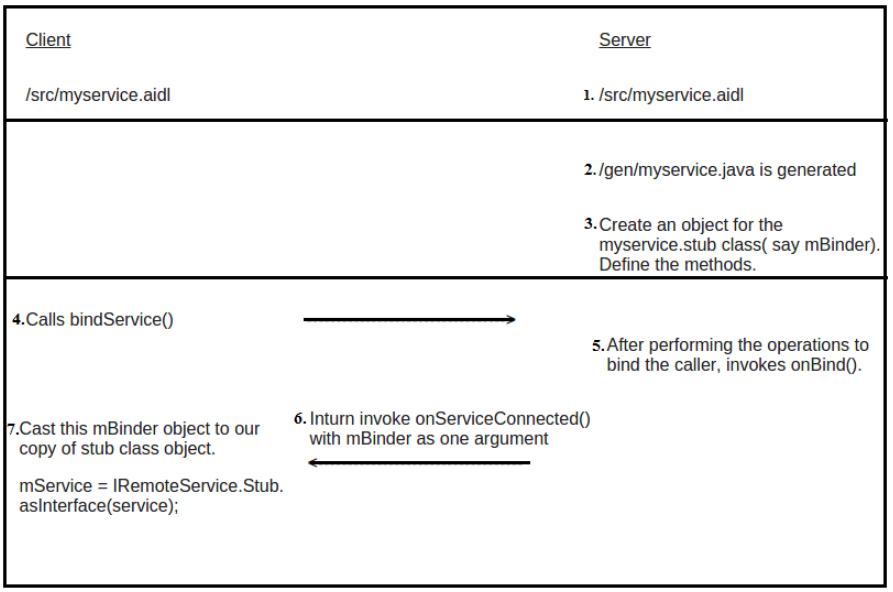

FIGURE 4.2: High level Binder Communication

implementing the communication protocols in kernel etc. The Binder middleware functionalities are achieved using a shared library `libbinder.so`. From a higher level, the Binder IPC is a shared memory mechanism. The client will copy the request to a kernel space, from where the server will obtain it. The server after performing the request, will copy the response back to the same kernel space. The client will then copy it. Since it is not possible for a client process to access the kernel space, the kernel drivers are used. The drivers are implemented in C.

4.2.2 Binder at the Kernel

The various communications between Binder Framework and Kernel are shown in Figure 4.3. Once the Binder Framework creates the parcels to be sent, an `ioct` call is made from within the `transact()`. It is of the form: `ioctl(fd, BINDER_WRITE_READ, &bwt)`, where fd is a file descriptor to `/dev/binder`, `BINDER_WRITE_READ` is a data structure and `&bwt` is a reference to it. The Binder driver will identify the server and will copy the request to the server's address space and wakes up a waiting thread in the server to handle the request. The thread will invoke the `onTransact()`, which will perform the middleware operations. Once the requestesd operation is performed, the result is again marshalled by libbinder, copied to the server address space and then sent to the Binder driver.

The important elements of `binder_write_read` are a write_buffer and a read_buffer. The write_buffer will contain the Binder transaction requests and the read_buffer

will contain Binder transaction replies. The structure of the same is given in Appendix A.

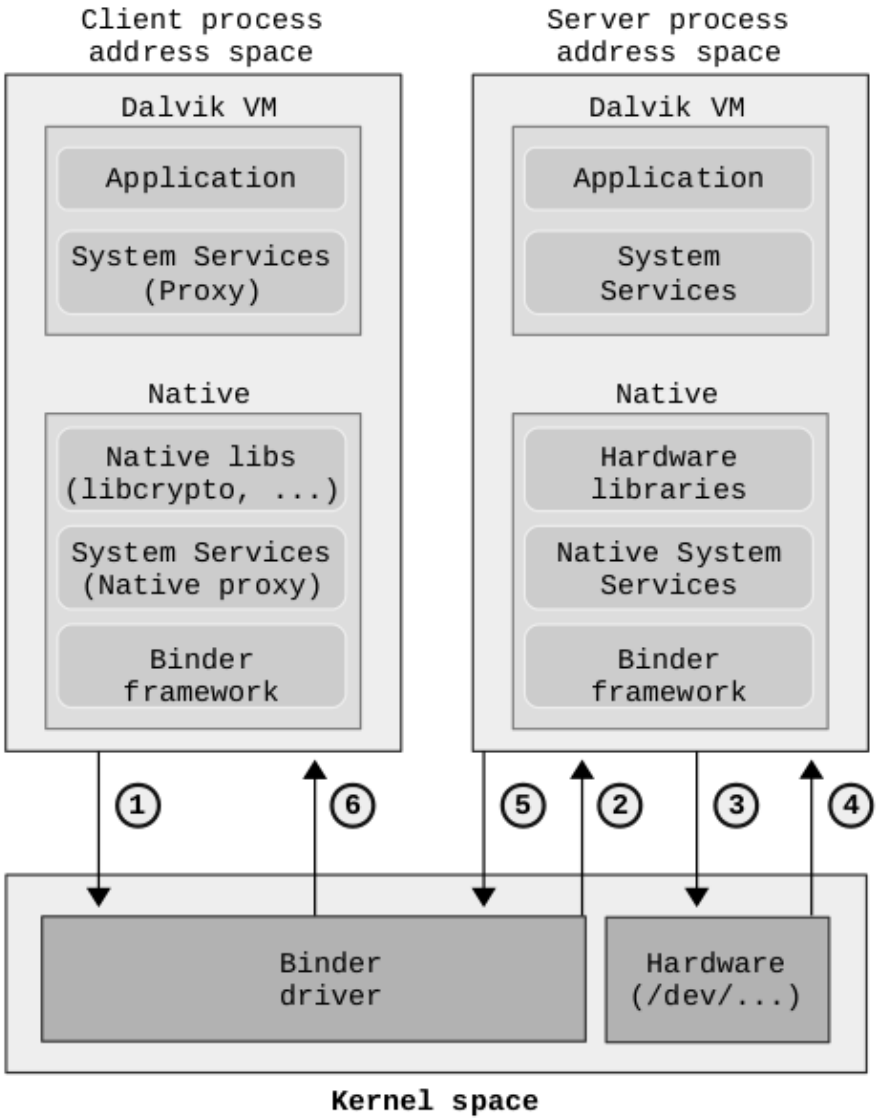

FIGURE 4.3: Overall Binder Communication [Schreiber (2011)]

The `write_buffer` holds an enum representing the transaction data, followed by another data structure `binder_transaction_data`. The `binder_transaction_data` contains the elements such as sender uid and pid, buffers to hold the offset and the actual data, target, code etc. The target is the method handle that should receive the transaction. Code refers to the id of the remote method.

Each process which make use of Binder IPC has a file descriptor to `/dev/binder`. The `binder.c` has a structure `binder_proc` defined within it. The `binder_proc` is a tree structure which contains a sorted list of all binder references allowed for

each attached process. That is, there is a `binder_proc` structure for each attached process.

4.2.3 Kernel Packet Sniffing

The kernel driver continuously sniffs the data being sent in any Binder communication. If the communication is from process B to A , and if B has a Binder reference to A, then kernel will verify whether Binder already belongs to the set of allowed Binders for the targeted process (here A). If not present already, the kernel driver introduce its reference into `refs_by_desc` and `refs_by_node`(both are structures of type rb_root) `rb-trees` in the target `binder_proc` [**?**]. Thus A will receive an invitation to call B. There is one special Binder reference that is inherently allowed by default, the one that refers to the ServiceManager. It is a Linux daemon implemented in `service_manager.c`. From the Binder context, the servicemanager perform two main roles: global binder locator and seeding the invitation relation.

A particular service (say A) registers its binder object at service manager. That is, it passes the binder reference for service manager. This means that servicemanager is invited to call the process A. Later on, when another process P asks a binder reference for A and the ServiceManager replies with this reference, the kernel driver sees this and it introduces that particular binder reference into the respective trees in `binder_proc` of process P. In this way, process P gets also invited to talk to A. Together it causes the same effect as P getting invited to call A. The reference sniffing by Kernel is defined in the structure `binder_transaction_data` in binder.h. The command/reply data being exchanged with a binder object is represented in it. The `data.ptr.buffer` contain the payload and the data.ptr.offsets is an array of offsets that index binder references in the payload.

The user space code prepares the data payload (called parcel), provides these indices to the kernel driver. The parcel is a data structure `flat_binder_object` defined in binder.h. When processing the data payload, the kernel driver looks at those particular offsets and manipulates the references being passed on and eventually introducing them into the respective trees in the target `binder_proc` structure, thus inviting the target process to call these binders later on. The mechanism is same for both command and reply data transfers. The local binders are represented directly by pointers to its userspace and remote binders are represented by handlers. In the kernel driver, there is a system- wide unique representation of each

single binder by `binder_node` structure instance. In the kernel space, local binders of a process are referenced directly by `binder_node` pointers sorted as nodes in r-b tree of that particular`binder_proc`, while remote binders are primarily referenced indirectly via descriptors(handlers) into `refs_by_desc` and `refs_by_node trees` in `binder_proc`.

The Binder objects are uniquely identifiable and hence it can act as shared token. The Binder identifiaction will be known only to the communicating entities. Thus, a Binder can be used as a security access token which can be shared across multiple processes. A callee process can identify its caller process by UID and PID in the Parcel object. Combined with the Android security model, a process can be uniquely identified.

Chapter 5

Binder Attacks

There are various kinds of attacks associated with the Binder. Two such attacks are explained in the following sections: Man-in-the-Binder and Cross-Binder Reference Forgery (XBRF).

5.1 Man-in-The-Binder

The Man-In-The-Binder exploit was described in [Artenstein and Revivo (2014)]. This attack is analogous to the man-in-the-middle attack, where the attacker code stands at the middle of a communication.

The attack was performed using a code injection technique on the shared library `libbinder.so`. The attack can only be done on a rooted device. The attack will first hook a method on the target application component. The method hooked is the point where the Binder Framework communicates with the kernel driver. The exploit is based on the vulnerability that the data sent between an application and Binder driver is in plain text. It is in the form of a structure known as Parcel, which is defined publically in parcel.h. The protocol tag `BC_TRANSACTIONS` and `BC_REPLY` can identify a Binder transacton. Further, the action performed on the parcel data can be identified by comparing the enum and the interface implementation. Hence it is possible for an attacker to identify a particular response of a method in a particular transaction.

The extent of this attack can be better explained by an attack scenario. This attack, makes it possible to establish a system wide keylogger. The applications that

use the standard IME are attacked. The target of the attacker is to obtain the Parcel sent by the IME to the client application. More precisely, the data sent between the IME Binder middleware and the kernel driver is the targeted transaction. The only communicaton between the middleware and kernel takes place through `ioctl` call made by the `talkWithDriver()` of `IPCThreadState`. So, inorder to control the `ioctl` transaction, the `talkWith Driver()` is hooked using a library injection technique. The attacker thus replaced the talkWithDriver() with a malicious code. In the hooked method, before the ioctl call is made, the Parcel data is copied to a file on user space. The Parcel, as explained in the earlier section, can be analysed to obtain the relevant data. For a key logger, the response from IME will have the tag `BC_REPLY`, interface as `com.android.internal.view.IInputContext` and the code for `setComposing Text` is 6 [elinux.org (2014)].

5.2 Performing the MIB Attack

For performing the Man-in-the-Binder (MIB) attack, the `ioctl` system call in `libbinder.so` needs to be hooked. To accomplish this, a library injection technique is used. This is done with the help of a tool called Droid-Injectso [Shao (2014)]. The service whose response is to be obtained is identified and the library injection is done on the respective service's shared object `libbinder.so`. Identifying the location of the shared `libbinder` object and changing the return address of the shared library method to point to the fake `ioctl` call are all done by the Droid-injectso.

The Injectso tool used for library injection makes use of the following information from ELF header, to perform the redirection of the imported function:

- The path to this library in the file system.
- The virtual address at which it is loaded.
- The name of the function to be replaced.
- The address of the substitute function.

The prototype for the redirection function in the C language is as follows:

```
void *elf_hook(char const *library_filename,
```

```
void const *library_address, char const *function_name,
void const *substitution_address);
```

The steps in redirection operation:

- Open the library file.
- The index of the symbol is stored in the .dynsym section, with name as that of the required function.
- In the `.rel.plt` section, search and find the relocation for the symbol with the specified index.
- If the required symbol is found, store its original address for future purposes. Overwrite the original address with that of the substitution function at the address specified in the relocation. This address can be calculated by summing the address at which the library is loaded with the offset value in relocation. Every time the function is invoked by the library, redirection will occur.
- If the symbol serached is not is not found in the `.rel.plt section`, it should be searched in the `rel.dyn` section likewise. In the rel.dyn section the symbol occurence will be multiple times.
- Restore the address of the original function or just NULL if the function with the required name was not found.

5.2.1 Performing Library Injection

To perform the attack, first the modified fake libbinder is created. The libbinder along with the injector code of Droid-injectso is compiled using NDK.

- `/Android/android-ndk-r8e/ndk-build /Android/mib/injector.c`
- `/Android/android-ndk-r8e/ndk-build /Android/mib/hook_ioctl.c`

The compiled objects are then moved to a writable location on Android device(say `/data/local`).

1. `adb push /Android/injector /data/local`

2. `adb push /Android/libbinder.so /data/local`

Then the injector code must be made executable. To make it executable, we require root permission. Hence, this attack will work on rooted devices only. Custom roms are very popular and are by default rooted. Also, with apps like Towel root, rooting can be done in a single click. Thus the requirement of a rooted victim device could be easily obtained. It is done as follows:

1. `adb shell chmod 755 /data/local/injector`

   Identify the `pid` of the target process(here pid is 753). Then the code injection is done with the help of the injector as follows:

2. `adb shell /data/local/injector -p 753`
   `-l /data/local/libhook_ioctl.so`

The modified ioctl and the `libhook_ioctl.so` at `/data/local` are injected to the process with pid 753 with the injector code at `/data/local`. The -p indicate the pid and -l indicates the library object.

The hooked ioctl call is given in Appendix. Once hooked, the parameters to the ioctl call can be obtained. The `ioctl` call as discussed in previous section, contains a protocol identifier, file descriptor and a pointer to the structure `binder_write_read`. The structures are defined in `/linux/drivers/staging/android/binder.h`. The `read_buffer` within this structure will contain the Binder reply. The initial 4 bytes of `read_buffer` contains a protocol tag. The required tag is 1, for Binder reply. The protocaol tags are defined in `enum_binder_return_protocol`. It is followed by the address to another structure `binder_write_read`, which holds a buffer containing the actual Binder reply. Once the pointer to `write_read` is obtained, then assigning to a structure variable of its type will let us access its elements. The arguments to `ioctl` obtained through the attack is shown in Figure **??**.

The value corresponding to read buffer is its contents. The first digit 3 corresponds to protocol tag and it indicates that it is a `BINDER_REPLY`. The remaining is the pointer to the structure `binder_transaction_data`. With these pointers to the write and read buffer, the binder transaction and reply information can be obtained. The `read_buffer` obtained is shown in Figure 5.2.

```
I/hook-ioctl(  265): [+] ioctl is invoked ...
I/hook-ioctl(  265): ** MIB  Attacked file descriptor  9
I/hook-ioctl(  265): ** MIB  Attacked transaction code -1072143871
I/hook-ioctl(  265): ** MIB  Attacked read buffer(hexa) 724b30ab
I/hook-ioctl(  265): ** MIB  Attacked write buffer(hexa) 724b30ab
I/hook-ioctl(  265): ** MIB  Attacked read buffer(lu) 1243061764
I/hook-ioctl(  265): ** MIB  Attacked read size(li) 1917530283
I/hook-ioctl(  265): ** MIB  Attacked read consumed(li) 1917530283
I/hook-ioctl(  265): ** MIB ultimate size of read buffer= 8
```

FIGURE 5.1: The `adb logcat` output printing the ioctl arguments

```
I/hook-ioctl(  266): content = M
I/hook-ioctl(  266): content = O
I/hook-ioctl(  266): content = U
I/hook-ioctl(  266): content = N
I/hook-ioctl(  266): content = T
I/hook-ioctl(  266): content = P
I/hook-ioctl(  266): content = O
I/hook-ioctl(  266): content = I
I/hook-ioctl(  266): content = N
I/hook-ioctl(  266): content = T
I/hook-ioctl(  266): content = =
I/hook-ioctl(  266): content = /
I/hook-ioctl(  266): content = m
I/hook-ioctl(  266): content = n
I/hook-ioctl(  266): content = t
I/hook-ioctl(  266): content = /
I/hook-ioctl(  266): content = a
I/hook-ioctl(  266): content = s
I/hook-ioctl(  266): content = e
I/hook-ioctl(  266): content = c
```

FIGURE 5.2: The Binder parcel data obtained by printing the contents at the address pointed by `read_buffer`

5.3 Cross Binder Reference Forgery (XBRF)

Each application component that uses Binder has a file descriptor to `/dev/Binder`. It contains a tree structure called `binder_proc`, which maintains the list of Binder references that the respective process can make [**?**]. This `binder_proc` structure is maintained by the kernel and is updated by sniffing the Parcels. The kernel driver continuously sniff the data being sent in any Binder communication. If the communication is from process B to A, and if B has a Binder reference to

A, then kernel will verify whether Binder already belongs to the set of allowed binders for the targeted process (here A). If not present already, the kernel driver introduce its reference into `refs_by_desc` and `refs_by_node`(both are structures of type `rb_root`) `rb-trees` in the target `binder_proc`. Thus A will receive an invitation to call B.

5.3.1 Binder_proc

A particular service (say A) registers its Binder object at Service Manager. That is, it passes the binder reference for Service Manager. This means that servicemanager is invited to call the process A. Later on, when another process P asks a binder reference for A and the ServiceManager replies with this reference, the kernel driver sees this and it introduces that particular Binder reference into the respective trees in `binder_proc` of process P. In this way, process P gets also invited to talk to A. Together it causes the same effect as P getting invited to call A. The reference sniffing by kernel is defined in the structure `binder_transaction_data` in binder.h. The command/reply data being exchanged with a Binder object is represented in it. The `data.ptr.buffer` contain the payload and the data.ptr.offsets is an array of offsets that index Binder references in the payload.

The user space code prepares the data payload (called `parcel`), provides these indices to the kernel driver. When processing the data payload, the kernel driver looks at those particular offsets and manipulates the references being passed on and eventually introducing them into the respective trees in the target `binder_proc` structure, thus inviting the target process to call these Binders later on. The mechanism is same for both command and reply data transfers. The local binders are represented directly by pointers to its userspace and remote binders are represented by handlers. In the kernel driver, there is a systemwide unique representation of each single Binder by `binder_node` structure instance. In the kernel space, local Binders of a process are referenced directly by `binder_node` pointers sorted as nodes in r-b tree of that particular`binder_proc`, while remote binders are primarily referenced indirectly via descriptors(handlers) into `refs_by_desc` and `refs_by_node trees` in `binder_proc`.

The Cross Binder Reference Forgery(XBRF) is an exploit in which an application having higher privilege passes directly its file descriptor to a lower privileged app. Thus, the less privileged app will get access to the services permitted to the original application.

Chapter 6

Translating Binder into Scala

The Android framework has a layered structure. Some of its components are written in Java, some in C++ and some in C. The API or the high level components are mainly in Java. The kernel and drivers are written entirely in C. There is a middleware that act as an interface between the application layer and the kernel, which is written in C++.

6.1 Java to Scala

The main objective of the project is to translate the components of the Android Binder in Java to a new language called Scala. The Scala is more consice that Java. Scala has several more features than Java, as explained in an earlier section. Thus, Scala is many ways is considered as the next generation Java. Hence, the Android ROM with the Scala components should be more efficient than the Java vesion.

The translated code has to be analysed for correctness. The variation in the lines of code has to measured. Unit testing and static analysis of the translated code must be done for checking the correctness. It must be ensured that the translated Scala version of Binder work exactly the same way as the Binder in Java. By default, the GNU compiler does not include Scala. Hence, the Scala compiler needs to be added to the build process.

Porting the Android framework components in Java to Scala was one of the major objective of this project. The Scala version must be absolutely equivalent its Java counterpart, in terms of working, throwing exceptions etc.

6.2 Android Framework in Scala

Our project aims at fortifying the Android system by building a new ROM, rather than preventing or detecting malwares. This on going project, thus brings security at the kernel level, as opposed to application layer. Another task associated with this project, is to replace all Java code with Scala code. The security flaws identified are rectified on the go. This includes the work of three Masters' theses completed in July 2015. The source code produced by these theses is on GitHub[1]. The new Amrita ROM will be reIeased in the coming months.

Shetti (2015) has done an MTech thesis that has substantially improved the ideas of Zygote, Morula [Lee et al. (2014)], has deeper ASLR and a few prevention measures for ROP (return oriented programming) attacks. This thesis includes Zygote and Morula re-written in Scala. It also contributes Enhanced Morula freshly written in Scala.

Narayanan (2015) has done an MTech thesis that rewrites the Java portions of Service Manager and Activity Manager.

The Scala version should be completely equvalent to the Java version, even to the extend of generating bugs. This was one common problem faced by all three of us. Unit testing appeared to be a possible solution to this problem, but, was too tedious and also was not sufficient enough to give the assurance of perfect equivalence. Though, the Java components resides at the API layer, most of them had very high dependency on the underlying middleware and kernel. Since, the middleware and kernel are in C++ and C respectively, unit testing required the use of several mock methods. This reduces the assurence of equivalence. Importing all files to the test project was also impossible because of heavy dependency between the files. Hence, unit testing proved to be an inefficient solution to the equivalence problem. Since, the ROM was not yet ready, dynamic analysis by inserting the Scala components to it, was not possible. These constraints left us with just one way of verifying equivalence: Extremely careful code walkthroughs. The deatils of this procedure is explained in the following sections of this chapter.

[1] https://github.com/yadukaladharan/Binder_Scala.git

6.3 Overview of Source Code Files

The following tables give the details of the files associated with Android Binder. The comparison between the lines of count in Scala and Java, the file locations, the number of methods defined etc are given in the table.

Java	Scala	nM	path name
287	230	21	`$OSPNM/Binder`
32	31	-	`$OSPNM/IBinder`
1677	995	163	`$OSPNM/Parcel`
45	33	9	`$OSPNM/ParcelableParcel`

TABLE 6.1: Comparion between the lines of count

The Java and Scala columns give source lines of code count (SLOC) of Binder files. The nM column gives the number of methods defined in that file. Note that `OSPNM=frameworks/base/core/java/com/android/internal/os`.

C++	nM	path name
197	19	`$FBLU/Binder.cpp`
274	23	`$FBLU/BpBinder.cpp`
16	-	`$FBLU/IInterface.cpp`
259	11	`$FBLU/ProcessState.cpp`
947	33	`$FBLU/IPCThreadState.cpp`
C	nM	path name
3325	77	`$/drivers/staging/android/binder.c`

TABLE 6.2: SLOC of `.cpp and c` files associated with Binder.

Note that `FBLU=frameworks/base/libs/utils`. The `.cpp` and `.c` files remain unchanged.

6.4 Mechanical Translation

The Java files were translated to Scala with the help of some software tools. These tools, in general, make use of simiar techniques to perform the translation. The converters make use of Scalagen library. The translators, will first generate an Abstract Syntax Tree(AST), from the Java source code. The AST is then parsed and is modified to match the Scala idioms. The Scala idioms are the expressions that are specific to Scala. the AST is then serialized into Scala format. Since, these techniques take each and every expression in the source code and translate

them, this itself gives a good assurence of the equivalence. Each method in the Scala translated file will have a corresponding method in its Java counterpart.

6.4.1 Tools Used

- ScalaCheck: This tool is used for property checking the Scala code. Property checking refers to producing statements about the output by trying hundreads of input. Unlike unit testing, the ScalaCheck will automatically give different inputs, so as to find the conditions that will produce the wrong output. Since, it can work for both Java and Scala, comparing the results produced by ScalaCheck for Java and the respective translated Scala code, is one sort of equivalence verification.

- ScalaStyle: This tool will identify the potential problems with the code such as, there are not too many types declared in a file. In addition to the predefined rules, custom rules also can be applied. Running ScalaStyle with standard predefined rules itself can identify several potential issues.

- Scala Build Tool(SBT): SBT is a build tool for Scala and Java projects. It has the native Scala compiler integrated into it. Automatically identifies the dependencies and provides support for projects with mixed Scala/Java files. It is heavy tool and requires the use of a Makefile to indicate the associated files and their paths. Also, integrating SBT to the build porcess of our Android source, is a complex process.

6.4.2 Makefile

Scala uses a rather heavy "build system", known as `sbt`[2]. But it is fairly simple matter to use the `make` tool used in building, e.g., an AOSP[3] ROM.

6.4.3 No Syntax Errors

We verified that the Scala files are without syntax errors by compiling them with `scalac`. The figure below shows the details of the resulting `.jar` file.

[2] http://www.scala-sbt.org/
[3] https://source.android.com/

```
# Makefile for Scala
# GNU Make has built in "knowledge" of many prog langs, but
# unfortunately not Scala (yet)
# list the extensions of files of interest
.SUFFIXES: .cpp .o .C .java .scala .class
%.class: %.scala
        /usr/bin/sbt clean run $< /home/yadu/Lollipop/out/target/common/obj/
        JAVA_LIBRARIES/android_stubs_current_intermediates/
        classes/android/os/
#sbt compile /home/yadu/Lollipop/frameworks/base/core/java
                                        /android/os/ >
# string var names and their assigned values
FILES = Binder.scala IBinder.scala Parcel.scala ParcelableParcel.scala
OBJFILES = Binder.class IBinder.class Parcel.class ParcelableParcel.class
PROJECT = /frameworks/base/core/java/android/os
$(PROJECT): $(OBJFILES)
        ls -l *.class
test: quickSort.class
        scala  QuickSort        # use actuall class file name base
tar archive: clean
        (cd ..; tar cvvfj ./$(PROJECT).tbz $(PROJECT); ls -l $(PROJECT).tbz)
clean:
        rm -fr *.o *~ *.out $(PROJECT) $(OBJFILES)
# -eof-
```

FIGURE 6.1: Makefile for Scala

```
( jar cvf ./sourcefiles.jar sourcefiles; ls -l sourcefiles.jar)
added manifest
adding: sourcefiles/(in = 0) (out= 0)(stored 0%)
adding: sourcefiles/Parcel.scala(in = 28970) (out= 4829)(deflated 83%)
adding: sourcefiles/IBinder.scala(in = 1278) (out= 466)(deflated 63%)
adding: sourcefiles/ParcelableParcel.scala(in = 1191) (out= 414)(deflated 65%)
adding: sourcefiles/Binder.scala(in = 6577) (out= 1710)(deflated 74%)
-rw-rw-r-- 1 yadu yadu 8442 Jul 22 06:20 sourcefiles.jar
( jar cvf ./sourcefiles.jar sourcefiles; ls -l sourcefiles.jar)
added manifest
adding: sourcefiles/(in = 0) (out= 0)(stored 0%)
adding: sourcefiles/Parcel.scala(in = 28970) (out= 4829)(deflated 83%)
adding: sourcefiles/IBinder.scala(in = 1278) (out= 466)(deflated 63%)
adding: sourcefiles/ParcelableParcel.scala(in = 1191) (out= 414)(deflated 65%)
adding: sourcefiles/Binder.scala(in = 6577) (out= 1710)(deflated 74%)
-rw-rw-r-- 1 yadu yadu 8442 Jul 22 06:20 sourcefiles.jar
```

FIGURE 6.2: The result of `make jar archive`

6.5 A Java Method versus the Scala Method

A systematic comparison was made between every pair of Java and it's corresponding Scala method. Each line of code was carefully read, spending a couple of weeks for this process. Some methods showed an obvious equivalency whereas, many others needed a close line by line observation. Below we illustrate this with a few examples.

6.5.1 Obviously Equivalent

There are several Java methods that got translated into Scala methods that are patently equivalent. All it takes is an understanding of the syntax and semantics of the constructs in both languages.

```
public class Binder implements
    IBinder {
    private static final boolean
    FIND_POTENTIAL_LEAKS = false;
    private static final boolean
    CHECK_PARCEL_SIZE = false;
    static final String TAG = "
    Binder";
    private static String
    sDumpDisabled = null;
    public static final native int
    getCallingPid();
    public static final native int
    getCallingUid();
    public static final UserHandle
    getCallingUserHandle() {
        return new UserHandle(
    UserHandle.getUserId(
    getCallingUid()));
    }
```

FIGURE 6.3: Class definition in Binder.java

```
object Binder {
  private val FIND_POTENTIAL_LEAKS =
     false
  private val CHECK_PARCEL_SIZE =
    false
  val TAG = "Binder"

  private var sDumpDisabled: String
    = null
  def getCallingPid(): Int

  def getCallingUid(): Int

  def getCallingUserHandle():
    UserHandle = {
    new UserHandle(UserHandle.
    getUserId(getCallingUid))
  }
```

FIGURE 6.4: Respective class definition in Binder.scala

The class definitions are similar. Instead of the keyword `class`, Scala uses `object`. By default everyting is public. Hence, only the private elements needs to be specified so, explicitly.

The IBinder.java is an interface to the Binder methods. Hence, it contains only the initialisations and method declarations. The verification hence is not much complex.

In Scala, the type of the variables are inferred automatically. The type is simply given as either `val` for constants and `var` for variables. The return type is specified after the name of the function and `Int`, specifies it to be of integer type. It can also be given as `var`. The `import` statements are also identical. Similar to `interfaces` in Java, `traits` are used to define object types with the signature of the supported methods. The `def` keyword defines a method in Scala. The default access mode is public in Scala.

The `boolean transact()` a simple method that calls the `checkParcel()` and `transactNative()`. The equivalence can be easily determined for such methods, as there are no much complex operations or use of Java idioms.

```
import java.io.FileDescriptor;
public interface IBinder {
int FIRST_CALL_TRANSACTION  = 0
    x00000001;
 int LAST_CALL_TRANSACTION   = 0
    x00ffffff;
 int PING_TRANSACTION        = ('_
    '<<24)|('P'<<16)|('N'<<8)|'G';

 int DUMP_TRANSACTION        = ('_
    '<<24)|('D'<<16)|('M'<<8)|'P';

 int INTERFACE_TRANSACTION   = ('_
    '<<24)|('N'<<16)|('T'<<8)|'F';


 public String
    getInterfaceDescriptor() throws
    RemoteException;

 public boolean pingBinder();
 public boolean isBinderAlive();
 public IInterface
    queryLocalInterface(String
    descriptor);
 public void dump(FileDescriptor fd,
     String[] args) throws
    RemoteException;
 }
```

FIGURE 6.5: Some statements in IBinder.java

```
import java.io.FileDescriptor
trait IBinder{
 var FIRST_CALL_TRANSACTION: Int = 0
    x00000001
 var LAST_CALL_TRANSACTION: Int = 0
    x00ffffff
 var PING_TRANSACTION: Int = ('_' <<
     24) | ('P' << 16) | ('N' << 8)
    | 'G'
 var DUMP_TRANSACTION: Int = ('_' <<
     24) | ('D' << 16) | ('M' << 8)
    | 'P'
 var INTERFACE_TRANSACTION: Int = ('
    _' << 24) | ('N' << 16) | ('T'
    << 8) | 'F'

 def getInterfaceDescriptor():
    String
 def pingBinder(): Boolean
 def isBinderAlive(): Boolean
 def queryLocalInterface(descriptor:
     String): IInterface

 def dump(fd: FileDescriptor, args:
    Array[String]): Unit
 }
```

FIGURE 6.6: Respective statements in IBinder.scala

```
public boolean transact(int code,
    Parcel data, Parcel reply, int
    flags) throws RemoteException {
        Binder.checkParcel(this,
    code, data, "Unreasonably large
    binder buffer");
        return transactNative(code,
    data, reply, flags);
    }
```

FIGURE 6.7: The transact() in Binder.java

```
def transact(code: Int,
      data: Parcel,
      reply: Parcel,
      flags: Int): Boolean = {
    Binder.checkParcel(this, code,
    data, "Unreasonably large binder
     buffer")
    transactNative(code, data, reply
    , flags)
  }
```

FIGURE 6.8: The transact() in Binder.scala

6.5.2 Non-Obvious but Equivalent

For most of the other Scala methods, the equivalence requires a bit of explanation. Below are some examples of such methods.

The `execTransact()` is a complex function. Since the datatypes are automatically inferred, the elements of type Parcel will be taken as type `val`, by the converter. The Scala compiler will not remember the inferred type beyond this method. In the context of this method, the values assigned to the Parcel objects does not change. Hence, it identifies the Parcel to be of type `val`. But, the Parcel

```
private boolean execTransact(int
  code, long dataObj, long
  replyObj,
  int flags) {
Parcel data = Parcel.obtain(
  dataObj);
Parcel reply = Parcel.obtain(
  replyObj);
boolean res;
try {  res = onTransact(code, data
  , reply, flags);
  } catch (RemoteException e) {
if ((flags & FLAG_ONEWAY) != 0) {
 Log.w(TAG, "Binder call failed.",
   e);
  } else {reply.setDataPosition(0)
  ;
              reply.writeException
  (e);}
       res = true;
    }  }
 checkParcel(this, code, reply, "
  Unreasonably large binder reply
  buffer");
  reply.recycle();
  data.recycle();
  StrictMode.
  clearGatheredViolations();
  return res; }}
```

FIGURE 6.9: The exectransact() in Binder.java

```
private def execTransact(code: Int,
    dataObj: Int, replyObj: Int,
     flags: Int): Boolean = {
  var data = Parcel.obtain(dataObj
  )
  var reply = Parcel.obtain(
  replyObj)
  var res: Boolean = false
  try { res = onTransact(code,
  data, reply, flags)} catch {
    case e: RemoteException => {
      reply.setDataPosition(0)
      reply.writeException(e)
      res = true  }
    }
  reply.recycle()
  data.recycle()
  res }
```

FIGURE 6.10: The exectransact() in Binder.scala

objects will be modified and hence needs to be of type `var`. The method in Java makes use of a function `StrictMode.clearGatheredViolations()`. The StrictMode is used to alert the accidental network or disk access. Since, it is a specific funcion used only in Android, the converter will fail to identify it and will simply ignore that statement. The `freeze` option in `scala.util.Properties` can achieve a similar functionality in Scala. Similarly, the `Log.e, Log.w` etc, may be ignored by the converter, along with the conditional statement associated with it. For example, the statements `if ((flags & FLAG_ONEWAY) != 0)  Log.w(TAG, "Binder call failed.", e)`, is ignored in the Scala version. Hence, it must be ensured that the correct exceptions are thrown even without these statements.

The below example shows the `dumpAsync()` in Java and its corresponding Scala version.

The above example involves the creation of a new thread. The converter is capable of identifying this and the method `run()` isexecuted in a thread. There is no `static` or `final` attribute in Scala. The file operations and threads, are all similar in Scala and Java.

```
public void dumpAsync(final
    FileDescriptor fd, final String
    [] args) {
        final FileOutputStream fout
    = new FileOutputStream(fd);
        final PrintWriter pw = new
    FastPrintWriter(fout);
        Thread thr = new Thread("
    Binder.dumpAsync") {
            public void run() {
                try { dump(fd, pw,
    args);
                } finally {
                    pw.flush();
                }  }  }; thr.start()
    ; }
```

FIGURE 6.11: The transact() in Binder.java

```
def dumpAsync(fd: FileDescriptor,
    args: Array[String]) {
    val fout = new FileOutputStream(
    fd)
    val pw = new PrintWriter(fout)
    val thr = new Thread("Binder.
    dumpAsync") {
      def run() {
        try {
          dump(fd, pw, args)
        } finally {
          pw.flush()
        } } } thr.start() }
```

FIGURE 6.12: The transact() in Binder.scala

6.5.3 All Pairs of Methods Were Equivalent

In the Android Framework, there is no usage of functional programming or complex Java synatx. Hence, not many Scala idioms were required to be be used. All these methods were under the above mentioned categories. With a considerable time spent on understanding the Scala syntax and by reviewing the translated code, the equivalence was ensured.

6.6 Establishing Equivalence

An exhaustive testing or use of formal methods to mathematically prove the equivalence is required. Ensuring code equivalence by line by line code walk throughs is done here, though it is not accepted as a standard software engineering technique. Since the components we are translating are very large in terms of the number of lines, number of files, the dependency on the underlying layers of the framework etc, the standard techniques are difficult to be applied. Also, in the absence of the Android ROM that would have incorporated the components we built, the dynamic testing is also infeasible. Nevertheless, Nevertheless, we have put in a great effort and time to manually verify and ensure that the Scala code developed by us is bug-for-bug is quivalent to the original Java code.

Chapter 7

The Encryption Scheme

The Binder mechanism handles the entire IPC in Android. Hence, subverting the Binder can expose all critical application data to an attacker. As explained in the previous section, there are a number of ways to do this[Jia et al. (2015)]. Encrypting the application data is a good solution to thwart these attacks. Since, the number of applications and services are very high, getting each communicating entities into a key agreement is not an easy solution. Hence in this thesis, we suggest a technique to encrypt the Binder communications through a shared secret key mechanism. The attackers are usually interested in the Parcel data that is received as `BC_REPLY` from a service.

The Binder data from the client is first converted into Parcels. The Parcel has a complex structure and is given in Appendix A. The data from the client to the server will be present in the `write_buffer` and the response from the server to the client will be present at the `read_buffer`. Both these buffers are defined within the structure `binder_write_read`. Thus, the client will fill the `write_buffer` of Parcel object `data` with arguments to the remote method and will then invoke the method `transact()`. The signature of `transact()` is as follows: `transact(int code, Parcel data, Parcel reply, int flags)`. The code indicate the type of communication. For Binder transactions, the code corresponds to BC_TRANSACTION. The `Parcel data` contains the buffer containing request or data sent from client to the server and the `reply` will contain data obtained as a response to a request. For communications from client to server, the `reply` will be `null` and `data` will be consumed. For Binder reply, both these buffers will be non empty.The `read_buffer` is a raw buffer.

7.1 The Encryption Scheme to Prevent Man-in-the-Binder

The proposed encryption scheme is as follows. Once the server finishes performing the requested operation over the client data, the reponse is written to the Parcel `reply` object. It then calls the `transact()`, followed by the `nativeTransact()` to invoke the midddleware methods. Before the `nativeTransact()` is called, the `reply` is checked whether it is null or not. If not `null`, then it indicate that it is a Binder response data. Hence, it must be encrypted. The `reply` is a raw buffer and its contents are added as 32bit unsigned long inetger elements. The `reply` is parsed as 32 bit blocks. Each 32bit block is encrypted using the Tiny Encryption Algorithm(TEA). The TEA is a light weight encryption technique and makes use of a 32 bit key. The key is kept hardcoded in our implementation. The encrypted blocks are written back to the `reply` buffer. The `nativeTransact()` is then invoked with `reply` as one of its argument. The signatures for `nativeTransact()` and `transact` are identical. The Parcels are then flattened and sent to the Binder driver through an `ioctl`. The Binder driver will then call the `onTransact()` on the client. This method is responsible for unmarshalling, unflattening etc. The decryption of the Parcel `reply` occurs here. The `reply` is first checked whether it is null or not. If not null, then the buffer is parsed as 32 bit blocks and decrypted. The decrypted data is written back to the buffer. A pointer to the starting of this buffer is returned by the `onTransact()`, which the client uses to obtain the reply. For parsing the `reply` buffer as 32bit blocks, the `reply.setDataPosition` is made use of. After each block, the position of the buffer is incremented by 4 bytes.

7.1.1 Key Selection

The key is of size 128bits. For generating the random key strings, the `/dev/random` can be used. It allows to specify the size of keys to be generated. The keys must be generated by the kernel. The key for a particular transaction can be either added to the respective communicating application's address space or can be returned through system calls.

Chapter 8

Evaluation

The Java components associated with the Binder mechanism are translated to Scala. There has been a significant reduction in the lines of code. The Scala is more efficient in terms of size and performance, when compared to Java. Hence, the Amrita ROM wih the Scala components should be more enhanced compared to its Java counterpart. The below table shows the lines of count compasion.

Java	Scala	path name
287	230	`$OSPNM/Binder`
32	31	`$OSPNM/IBinder`
1677	995	`$OSPNM/Parcel`
45	33	`$OSPNM/ParcelableParcel`

TABLE 8.1: Lines of count comparison between Java and Scala files

The Binder in Android responsible for entire inter process communication. Hence, captivating it can bring the entire application data under the attacker's control. Thus, understanding the Binder mechanism in details and the various attacks on it is very important. The proposed encryption scheme can prevent the Man-in-the-Binder attack. The new mechanism aims at encrypting the application data sent through the Binder mehanism. Since, all the communications, whether inter-app or intra-app, takes place through Binder, the overhead it wll inccur on performance will be high. Hence, to reduce this overhead a Light Weight Cryptographic(LWC) [Eisenbarth et al. (2007)] technique has been used. The LWC used is Tiny Encryption Algorithm(TEA). The response from every service is encrypted. Since, a unique key is assigned to each service, the other types of attacks such as XBRF, can also be prevented. Any type of library injection attack can only obtain the cipher text.

The ROM where we will include the encryption of `Binder parcels` is yet to be released.

Chapter 9

Conclusion

The Android Binder is the core mechanism that takes care of all communications between the processes in Android. The Android source code is written in C , C++ and Java. The Scala is known as modernized Java. It has several more features than Java, is more concise, less error prone and more efficient. Hence, translating the Java components of Binder to Scala should improve the efficiency and security of the Binder mechanism and hence the Android system.

The Binder IPC is an elaborate mechanism, which has been well documented in this thesis. Subverting the Binder can cause all the sensitive app information to be exposed to the attacker. There are various attacks on Binder, of which one potentially very harmful attack is the Man-in-the-Binder. The paper explains a real implementation of the attack on an Android device and an encryption mechanism to mitigate it. The encryption scheme used is a light weight mechanism and hence the overhead will be minimum.

Chapter 10

Publication

1. The paper titled, **"An Encryption Technique to Thwart Android Binder Exploits"** [Yadu Kaladharan, Prabhaker Mateti and Jevitha K.P (2015)], has been accepted in *International Symposium on Intelligent Systems Technologies and Applications (ISTA' 15)*. The paper will be published by Springer in Advances in Intelligent Systems and Computing Series.

Chapter 11

Appendix

11.1 Structures associated with Binder transactions

System call ioctl:
The communication with the kernel is accomplished by an `ioctl` system call. The `ioctl` call is made from within the `transact()`. It is of the form: `ioctl(fd, BINDER_WRITE_READ, &bwt)`.

The `fd` in `ioctl` call is a file descriptor to `/dev/binder`, `BINDER _WRITE_READ` is a data structure and `&bwt` is a reference to it. The structure of `binder_ write _read` is as follows:

```
struct binder\_write\_read {
  signed long write\_size;
  signed long write\_consumed;
  unsigned long write\_buffer;
  signed long read\_size;
  signed long read\_consumed;
  unsigned long read\_buffer;
};
```

Binder Transaction data The `write_buffer` holds an enum representing the transaction data. It is followed by another data structure `binder_transaction_data`. Given below is its form.

```
struct binder\_transaction\_data {
 union {
   size\_t handle;
   void *ptr;
            } target;
```

```
 void *cookie;
 unsigned int code;
 unsigned int flags;
 pid\_t sender\_pid;
 uid\_t sender\_euid;
 size\_t data\_size;
 size\_t offsets\_size;
 union {
   struct {
          const void \_\_user *buffer;
          const void \_\_user *offsets;
                } ptr;
 uint8\_t buf[8];} data;
};
```

The target is the method handle that should receive the transaction. Code refers to the id of the remote method.The driver then copies the data to the server's address space and wakes up a waiting thread in the server to handle the request. The thread will invoke the `onTransact()`, which will perform the middleware operations. Once the requested operation is performed, the result is again marshalled by libbinder, copied to the server address space and then sent to the Binder driver.

11.2 The hooked `ioctl` is as given below.

```
#include <unistd.h>
#include <sys/types.h>
#include <android/log.h>

#include "../../../libhook/hook.h"
//added by Yadu for parcel related structures.
#include "binder.h"
#define LOGI(...) ((void)__android_log_print(ANDROID_LOG_INFO, "hook-ioctl",
    __VA_ARGS__))
#define LOGE(...) ((void)__android_log_print(ANDROID_LOG_ERROR, "hook-ioctl",
    __VA_ARGS__))

void _init(char *args)
{
    LOGI("[+] lib loaded ...");
}

int (*orig_ioctl)(int, int, ...);

int hooked_ioctl(int fd, int cmd, void *data)
{
   int x;
   LOGI("[+] ioctl is invoked ...");
   LOGI("****** Yadu 1 Attacked file descriptor  %d  -", fd);
   LOGI("****** Yadu 2 Attacked transaction code %d  -", cmd);
   struct binder_write_read bwr;
   struct binder_transaction_data *btd;
```

```
   LOGI("****** Yadu 3  Attacked read buffer(hexa) %x ", ((struct
    binder_write_read*)data)->read_buffer);
   LOGI("****** Yadu 4 Attacked write buffer(hexa) %x ", ((struct
    binder_write_read*)data)->write_buffer);

        x= (*orig_ioctl)(fd, cmd, data);
        LOGI("****** Yadu ultimate 1  Attacked read buffer(lu) %lu ", ((struct
    binder_write_read*)data)->read_buffer);
        unsi
        //LOGI("****** Yadu ultimate 1  Attacked read buffer(I64u) %I64u ", ((
    struct binder_write_read*)data)->read_buffer);
        LOGI("****** Yadu ultimate 1  Attacked read size(li) %lu ", ((struct
    binder_write_read*)data)->read_size);
        LOGI("****** Yadu ultimate 1  Attacked read consumed(li) %lu ", ((struct
    binder_write_read*)data)->read_consumed);
        LOGI("****** Yadu ultimate size of read buffer= %d ", (sizeof((struct
    binder_write_read*)data)->read_buffer));
        //LOGI("****** Yadu ultimate 2  Attacked read buffer(hexa) %x ", *data->
    read_buffer);
        slong=((struct binder_write_read*)data)->read_size;
        LOGI("$$$$$$$$BINDER RESPONSE %ld$$$$$$$$",slong);
           unsigned char *p = (unsigned char*)&((struct binder_write_read*)data)
    ->read_buffer;
        while(i<slong)
                 {LOGI("content = %c", *p);
                p++;
                i++;}

        return x;
}

void so_entry(char *p)
{
    char *sym = "ioctl";

    // servicemanager does not use /system/lib/libbinder.so
    // therefore, if you want to hook ioctl of servicemanager
    // please change module_path to /system/bin/servicemanager
    char *module_path = "system/lib/libbinder.so";
    ///system/lib/libbinder.so
    orig_ioctl = do_hook(module_path, hooked_ioctl, sym);

    if ( orig_ioctl == 0 )
    {
        LOGE("[-] hook %s failed", sym);
        return ;
    }

    LOGI("[+] orignal ioctl: 0x%x", orig_ioctl);
}
```

Bibliography

Alizadeh, M., Hassan, W. H., Zamani, M., Karamizadeh, S., and Ghazizadeh, E. 2013. Implementation and evaluation of lightweight encryption algorithms suitable for RFID. *Next Generation Information Technology 4*, 1, 65.

Artenstein, N. and Revivo, I. 2014. Man in the Binder: He Who Controls IPC, Controls the Droid. https://www.blackhat.com/docs/eu-14/materials/eu-14-/Artenstein-/Man-In-The-Binder-/He-Who-Controls-/IPC-Controls-The-Droid-wp.pdf.

Doxygen. 2013. Doxygen. http://www.stack.nl/~dimitri/doxygen/.

Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., and Uhsadel, L. 2007. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers* 6, 522–533.

elinux.org. 2014. Android Binder Driver. *eLinux.org*. http://elinux.org/Android_Binder.

Google. 2014. Android Security 2014 in Review. https://static.googleusercontent.com/media/source.android.com/en//devices/tech/security/reports/Google_Android_Security_2014_Report_Final.pdf.

Gordon, R. and Essential, J. 1998. Java native interface. *Prentince Hall PTR*.

International Data Corporation. 2015. Smartphone OS Market Share. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

Jia, P., He, X., Liu, L., Gu, B., and Fang, Y. 2015. A Framework for Privacy Information Protection on Android. In *Computing, Networking and Communications (ICNC), 2015 International Conference on*. IEEE, 1127–1131.

Lee, B., Lu, L., Wang, T., Kim, T., and Lee, W. 2014. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 424–439.

Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. 2014. *The Java virtual machine specification*. Pearson Education.

Narayanan, S. 2015. SMScala: Secure Android ServiceManager Component in Scala. M.S. thesis, Amrita Vishwa Vidyapeetham, Ettimadai, Tamil Nadu 641112, India. Advisor: Prabhaker Mateti.

Osherove, R. 2013. The Art Of Unit Testing. http://artofunittesting.com/.

Schreiber, T. 2011. Android Binder. M.S. thesis, Ruhr University, Bochum, Germany, http://www.ruhr-uni-bochum.de/. http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf.

Shao, Y. R. 2014. Droid-injectso. https://github.com/windflyer/droid_injectso.

Shetti, P. 2015. Enhancing the Security of Zygote/Morula in Android Lollipop. M.S. thesis, Amrita Vishwa Vidyapeetham, Ettimadai, Tamil Nadu 641112, India. Advisor: Prabhaker Mateti.

Shoumikhin, A. 2010. Redirecting functions in shared ELF libraries. http://www.codeproject.com/Articles/70302/Redirecting-functions-in-shared-ELF-libraries.

Yadu Kaladharan, Prabhaker Mateti and Jevitha K.P. 2015. An encryption technique to thwart android binder exploits. In *The International Symposium on Intelligent Systems Technologies and Applications (ISTA15)*. Springer.

Zakrajek, L. 2013. Java to Scala Converter. http://javatoscala.com/.