Journal of Information Assurance and Security.
ISSN 1554-1010 Volume 15 (2020) pp. 026-034

Reverse Engineering for potential Malware detection: Android APK Smali to Java

Girish Sharma[1], Mehul Mahrishi[2] , Kamal Kant Hiran[3] and Dr. Ruchi Doshi[4]

[1]Department of Computer Science & Engineering,
Swami Keshvanand Institute of Technology, Management & Gramothan, Jaipur -302017 India
girish@skit.ac.in

[2]Department of Computer Science & Engineering,
Swami Keshvanand Institute of Technology, Management & Gramothan, Jaipur -302017 India
kamalhiran@gmail.com

[3]Department of Computer Science,
Sir Padampat Singhania University, Udaipur, India

Abstract: **Emerge of Smartphone technology has changed the way of communication and processing the data. These smart phones can perform peculiar thing which was only limited to calling and texting previously. This work presents the reverse engineering of the Android application which is the one of the most prominent Smartphone technology based on the Linux kernel. Since it is very difficult to analyze the applications by using intermediate codes like smali, jimple or bytecode, this approach can be useful for the reseachers who work on control and data flow analysis of apps.**
The objective of this work is twofold. One is to identify the components specified by the developer using the Android application's Manifest file and also those class files which have not been specified in the Manifest file . The second objective is to reverse engineer all the components and classes i.e. to convert them in respective Java code.
Keywords: Reverse Engineering, Android, Smali, Jimple, control flow, data flow

I. Introduction

Since the inception of android based systems, is one of the ubiquitous technologies which are used for sharing the resources [1]. At the same time these systems can be exploited by the malicious program writers to exploit the data or resources intentionally. As these systems are increasing exponentially the malicious writers misuse the private data of the user to fulfill their malicious activities. Another big possibility is that one application may use another apps data and could do malicious things [2].
Lot of techniques has been developed for the analysis of an android application which can provide the flow analysis of the application [3]. For example SOOT based framework "Flowdroid" is one of the tool which can be used for static taint analysis [4] which may provide good precision and recall. Flowdroid uses Jimple intermediate code to perform the analysis of the applications [5].

II. Android: Impact, Influence and Motivation

Many tools and technologies have been built to analyze the Android applications to make the world of users more and more secure. The question behind it is, "Is there any ideal tool and methodology which could provide the entire flow of the application?" In this line, framework like SOOT [6] provide analysis for the Java programs by implementing the framework for different types of applications like concurrent applications, point-to analysis. This framework has been used for static and dynamic analysis for intra and inter-procedural features. Flowdroid, [5] which is based on SOOT framework, is used for static taint analysis which may provide good precision and recall. But it is not easy to customize the framework like SOOT. As the applications are being increased exponentially, the possibility of penetration in the users' private data and resources also has increased. This possibility may occur substantially when an application is given a lot of permission than it requires. Apart from this another possibility is when one app requires the data from another app also increases malicious activities to be done through collusion. Another side is also there, and also when the coding standards are not good then also there is a possibility of information leakage [7] [8]. One of the technique to make the malicious code hidden from user or analyst is to use obfuscation. Even the anti malware systems are not able to detect the code if the code is highly obfuscated [9]. Many techniques have been implemented to make the anti malware systems more and more robust to detect malware like Androguard [10] [11].

A. Brief About Android

Android is one the highly growing and developing operating systems and framework for the development of applications. It is based on Linux operating system developed and maintained by Open Handset Alliance which is led by Google. An Android application does not have a single entry point as in Java applications which have an entry point. All these communications is done between or among the activities, services or receivers known as the component, via the intents. Android different components, an entry point for an application, is explained in the Fig- 1.
Activity shows a user interface for interaction which could interact with any of the components. A service is a component which runs in the background without user intervention. A broadcast receiver broadcast's the message to the system. Basically, there are four types of components for an android application which serves the different purpose for the application and have its life cycle from the beginning to end means its creation to destroy.

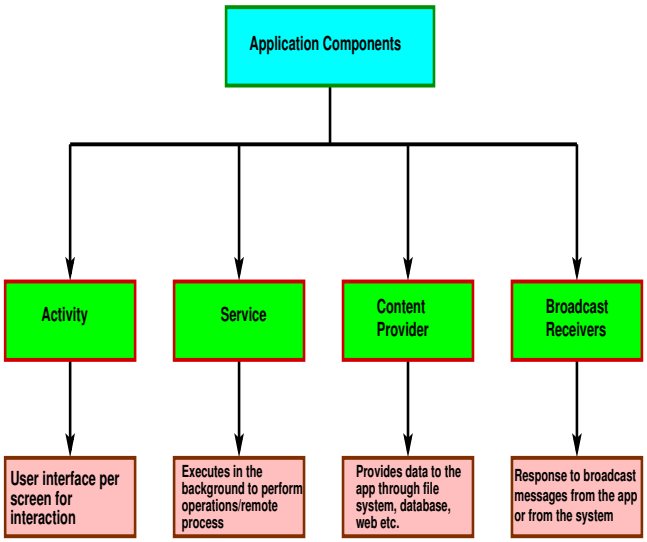

Figure. 1: Android application components

III. Related Work

[12] prepared a novel Eclipse based framework called as VAnDroid which is based on Model Driven Reverse Engineering (MDRE). The model is capable of automatically extracting security-related information from an Android app.
[13] proposed an assembly like language Smali$\hat{+}$, which generates Assembly code for reverse engineering Android applications.
[14] finds the anti pattern at the design and strutural level by using UML modeling. The method finds the 15 different anti-patterns applied on the 29 the mobile apps which divides the ant pattern into four groups. [15] states that malicious code is always different from the syntax of the code due to compiler dependency and therefore decompiled source code can be incorporated for malicious code classification. This not only provides penetration malware analysis but also helps in understanding its nature.
[16] builds an Android vulnerability detection framework which uses a hybrid approach of static and dynamic analysis of Android applications. The framework analyzes the uploaded APK for Information Leaks, Intent Crashes, Http Requests, Exported Android Components, Enabled Backup Mode, and Enabled Debug Mode.
[17] uses an online authentication mechanism to ensure the security mechanism improvement of the APK security. The apk file is encrypted, loaded and executed in the Android system after successful dynamic debugging and decompiling.

IV. Reverse Engineering

Understanding the Dalvik bytecode contained by the apk file is not easy to understand. To make the code user understandable form reverse engineering can be done to make the bytecode in readable form.
Lots of intermediate code has been generated for making the bytecode into a intermediate form that can be used for the analysis. Android applications can be disassembled by using Android-Apktool, dex2jar and Soot tools [18] which generate intermediate code in the form of Smali, Jasmin and Jimple respectively. The disassembly of apk by different tools is shown in the Fig- 2.

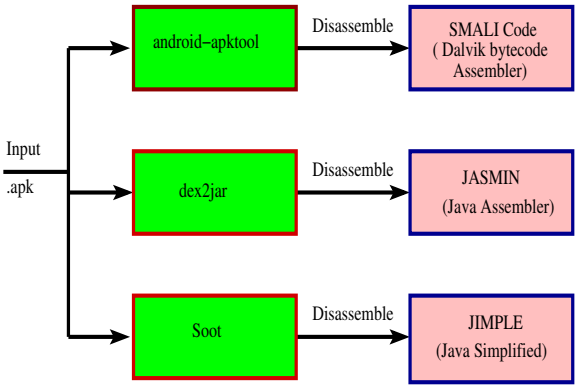

Figure. 2: Dissassembled output by different tools for .apk

1. **Baksmali/Smali**: These are disassembler and assembler which is used by Dalvik for dex format, also used for Virtual Machine implementation for Android Systems. Baksmali performs disassembly of dex.classes into smali form so that developer could make changes easily. Finally, Smali performs the assembly of edited smali code into dex.classes so the android system could understand.

2. **Jasmin**: It is one of the standard instruction formats for Java used for assembling these instructions into .class so that Java RunTime Environment could understand it.

3. **Jimple**: One of the intermediate code for Java code similar to Three Address Code uses only fifteen operations only, so the flow of the code easily be understood by the analyst. This code is especially useful in code optimization.

4. **Grimp**: One of the intermediate code for Java code which is in unstructured form. It is more readable form than the respective Jimple code.

5. **Baf**: It is much like the Java bytecode based on the stack representation. It is less complicated than the bytecode.

A comparative study says that disassembling done by Android-Apktool which generates the smali as an intermediate code preserves the code at the most and depicts the behavior as in the original code [18]. The comparative study of these intermediate codes can be shown in the Table

Table 1: **Program behavior for different intermediate code**

S. No.	Intermediate Code	Program Behavior Preservation
1.	SMALI	97.69
2.	JIMPLE	85.58
3.	JASMIN	81.92

V. Proposed Technique

This Section discusses proposed technique for converting the Smali code into Java code. The approach for this can be explained as:

A. Identifying Components and Classes

The proposed approach for finding components and classes can be explained as:

1. **Convert Classes.dex**: The approach first converts the .apk file into a readable form by using Android apktool which generates an intermediate smali form for classes.dex for the application by using Baksmali.

2. **Component identification**: The approach identifies the components of the application by scanning AndroidManifest.xml.

3. **Identify other classes**: Identify those classes which have not been specified in the AndroidManifest.xml by scanning complete dex of the application which has been converted into intermediate smali form.

1) System Architecture for finding the Components/-Classes:

The Fig. 3 shows the engine which takes multiple apk files as the input and identifies all the components and classes.

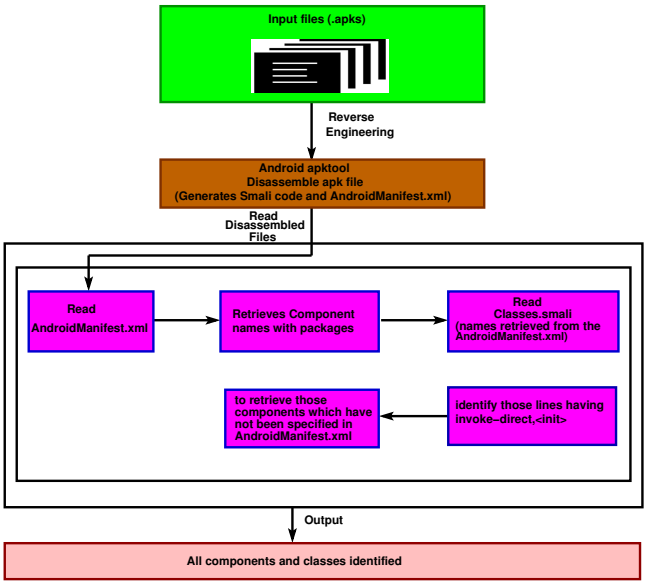

Figure. 3: **Components/Classes Identification**

B. Proposed Approach for Converting classes.smali into Java

This Section V-B discuss proposed the technique for converting the Android applications' classes into respective Java code. The approach for this can be explained in brief as:

1. **Components/Classes**: The section V-A shows identification of components and classes which is stored in the list. The list provides the names of the components to which reverse engineering is to be applied.

2. **Read Smali files**:Identify the location of each Smali file whose names with packages are stored in the list. Read all the Smali file from the dex of the application.

3. **Generate .java files**: Generate the java files for the components identified for the app by scanning and applying reverse engineering to those files.

1) System Architecture for Reverse Engineering Smali Code

The main objective of this work to apply reverse engineering mechanism to Smali classes and to convert them into Java code.
The Fig. 4 shows the engine to convert Smali files of multiple apk files into respective Java files. There is no restriction in the number of input files.

2) Phase I: Identify All The Information Within The Method

This Phase V-B.2 identifies all the information within the method. This can be explained as:

(i) **Read list**: The section V-A shows identification of components and classes which is stored in the

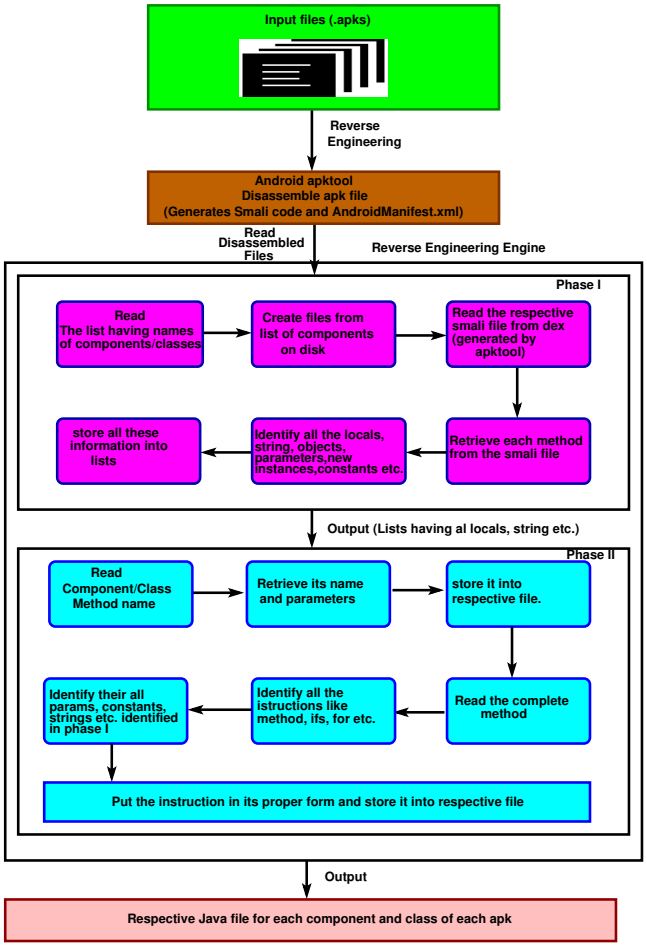

Figure. 4: Architecture: Reverse Engineering Smali

list. The list provides the names of the components/classes to which reverse engineering is to be applied.

(ii) **Read Smali files**: Read each of the Smali file specified by the list.

(iii) **Read method**: Identify the method from the .smali file. The start and end of the method can be easily known since a method starts with the line having `.method` and ends with `.end method`

(iv) **Identify instructions**: Within the method identify all the instructions having const-string, .local, move-result, new-array, const/4, const/high16, new-instance etc. Identify their respective variable or constant information and store them in their respective list which will be used further when the complete method is scaned in Phase II of the engine.

In phase I the engine identifies the following instructions and stores their information into the list.

(i) **const-class**: Provides the name of the classes used within the method which are used to invoke method of other class.

(ii) **const-string**: Provides the Strings with the name of their object.

(iii) **.local**: Provides the local variable used within the method with their names.

(iv) **iput-object**: Provides the information regarding the objects stored in another object.

(v) **iput-object**: Provides the information for an object having the value of another object.

(vi) **move-result-object**: Provides the information for statement or method whose resultant is object.

(vii) **move-result**: Provides the information for statement or method whose resultant is a basic type.

(viii) **new-instance**: Provides the information for the new-instances of the objects.

3) Phase II: Reverse Engineering Smali Code

This Phase V-B.3 generates adjacency list for each the application by scanning complete dex file generated by the apktool.

(i) The engine first reads the Components/Classes as mentioned in the Section V-A. These components are stored in the list. The engine reads and scans them one by one.

(ii) The system creates the files in the disk at the specific location specified for these components which will contain the Java code for respective smali(component) files.

(iii) The engine reads the smali for each of the components and identifies different types of instructions like method calling, if statements, loop statements in the smali code and generates its equivalent Java code.

(iv) The engine reads each and every smali file specified in the manifest and called classes and converts them in java file.

In phase II the engine converts the following methods in the smali code into java code.

(i) **invoke-direct**: If a smali instruction starts with invoke-direct, then it means it is a constructor or a private method. For example if a line in samli code is **“ invoke-direct {v4, p0, v5}, Landroid/content/Intent;-⟩⟨init⟩(Landroid/content /Context;Ljava/lang/Class;)V”**, this means there is a constructor call (⟨init⟩) by creating an object for the class Intent and passing this(p0) and v5 parameter and receiving the result in variable v4.

(ii) **invoke-virtual**: It looks the virtual table of methods associated with the object’s class. For example if a smali instructions line is **“invoke-virtual {v4, v5, v1}, Landroid/content/Intent;-⟩putExtra(Ljava/lang/String;Ljava/lang/String;)Landroid/content/Intent;”**, this means there is function call putExtra through Intent’s object (v4) in which two strings are passed (v5,v1).

(iii) **invoke-super**: It looks the virtual table of the super class associated with the method being called. For example if a smali instructions line is **"invoke-super {p0, p1}, Landroid/app/Activity;-⟩onCreate(Landroid/os/Bundle;)V"**, this means it is calling method onCreate() through object p0(this) and passing the object Bundle(p1).

(iv) **invoke-static**: This is used to invoke static method. For example **" invoke-static {p1, v1, v0}, Landroid/widget/Toast;->makeText(Landroid/content/Context; Ljava/lang/CharSequence;I)Landroid/widget /Toast;"** calls a method makeText() through class Toast passing three parameters Context(p1), CharSequence(v1) and integer value(v0).

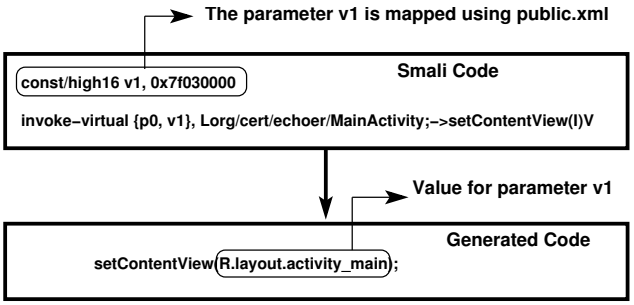

Figure. 5: **Mapping using public.xml**

VI. Test Results

To check the efficacy of the system, various tests on different Android applications have been performed. This section discusses various input application apks used by the system to perform analysis of outputs. It also discusses the test suite Droidbench [19] one of the highly used open source Android applications with Java code for performing static taint analysis and dynamic analysis of the applications by researchers and analyst [20]. Apart from this, the system's performance have been checked with the real world and some developed applications too.

A. Test Data Set

This work uses Droidbench test suite, Google's Play Store applications and some developed applications for determining the efficacy of the system. Droidbench contains more than 150 applications having data leaks, reflection calls, etc. The system's test has been performed with the Play Store applications like Instagram, Facebook, IRCTC, Whats-app and more applications. The test suite used for system's analysis is shown in Table 2.

The test cases used for the system's analysis identifies the Java code for the respective Smali files.

B. Results for Apk Files

This work shows the system generated for converting the Smali files into Java files. `Echoer.apk` application

Table 2: **Test cases used by the system**

S. No.	Test Suite	Description
1.	Droidbench	More than 120 android applications with different cases.
2.	Google's Play Store	More than 1.6 million android applications.

of Droidbench is used for showing the results in figure 6.

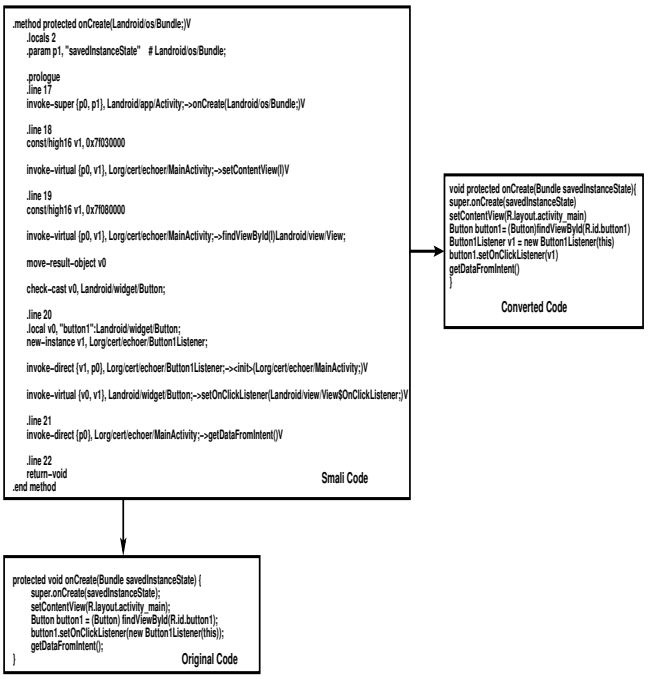

Figure. 6: **Echoer.apk**

In figure the original code, Smali code and converted code is shown. The Smali code is the dex opcodes. The first line of this code is the name of the method `onCreate` having the `Bundle` as a parameter. The name of the parameter is at line 3 showing `.param p1 "savedInstanceState"`. All these parameters for a method are identified by scaning the lines for `.param`.

1) Processing for invoke-virtual Methods

These types of methods are mostly called methods. These methods depicts the normal method call through object for another class method or directly calling methods within that class. The engine process these types of methods as:

(i) **Identify invoke-virtual**: Identify the line having **invoke-virtual** in the Smali code.

(ii) **Parameters**: Identify the parameters of the method which are in . For example here in the Fig. 7 the parameter is **v1**. Here p0 represents the object on which method is called and v1 is the actual parameter of the method **findViewById**.

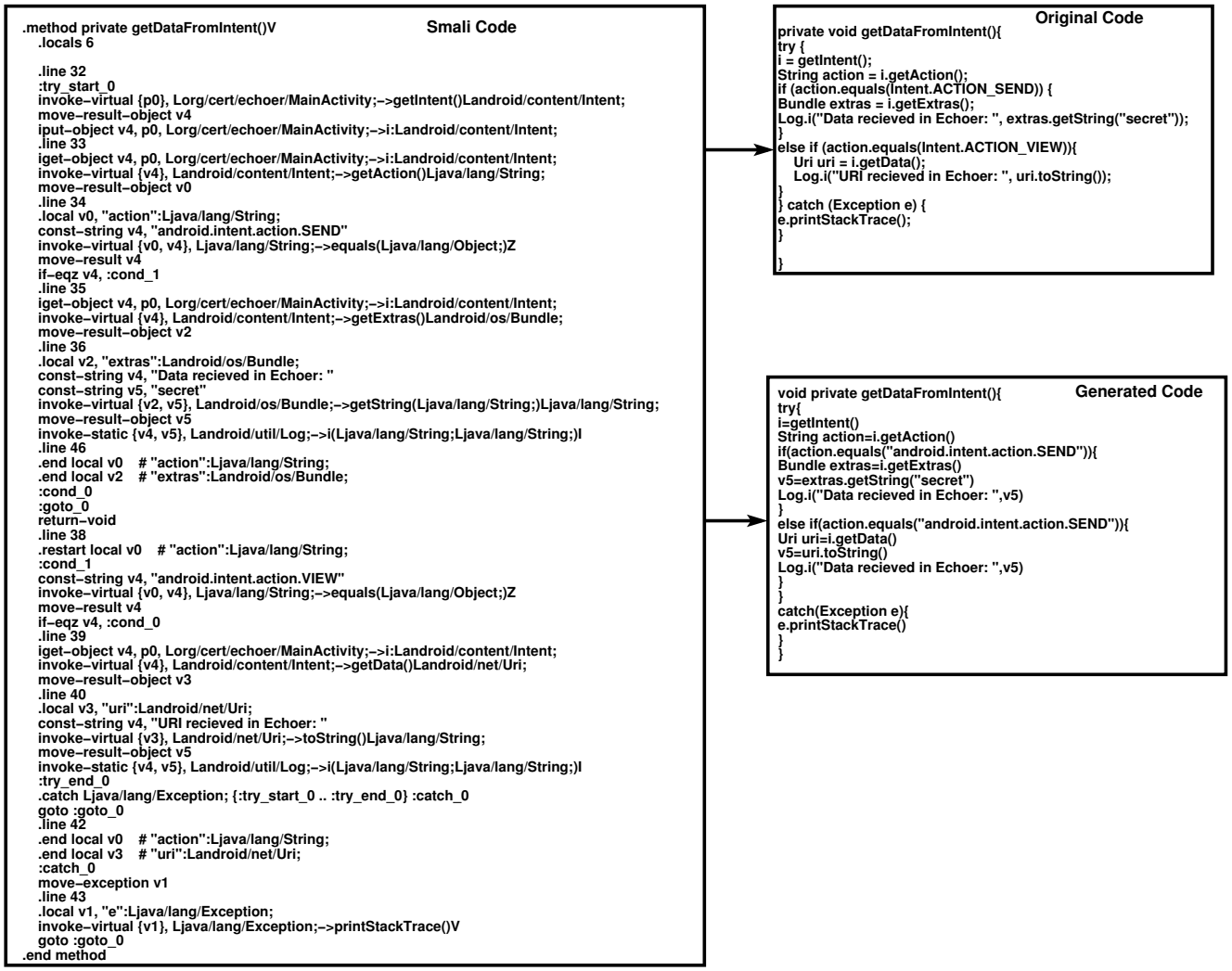

Figure. 7: **Echoer.apk**

(iii) **Object**: Object on which method is called is identified by reading the code in reverse. Here p0 represents the keyword **p0**. The parameter is also identified by reading the code in reverse. Note that all the paramters and object are identified by reading the code in reverse.

(iv) **move-result-object**: If the line following the invoke-virtual is move-result-object means method returns that type of object and stored in the object when returns. Here the result is moved to object v0 which is the object of Button class.

(v) **Casting**: It may also be possible that the result is casted in other type of object. If the move-result-object is followed by **check-cast** than the result is casted into that type of object. Here it is casted in **Button** object.

C. Some Results of Droidbench Applications

The Fig. 8 shows the result the Droidbench's application `StringToCharArray.apk`. The result is for the method onCreate(). The generated code almost resembles the original code. This code contains the `for` loop.

D. public.xml

Android dex contains a folder named with **res**. When an apk is reverse engineered using the **Apktool**. The dex also contains this **res** folder. This folder contains the layout, string values, and many other objects related to the application. An important file is the public.xml which is useful for converting the Smali code into the Java code.
This file consists of various string values in hexadecimal form which can be used for mapping when that particular value comes in the smali code in hexadecimal form. This particular scenario is depicted in Figure 5.

VII. Conclusions and Future Work

This work coverts the smali classes into Java form. For each identified method the system converts it into Java form.

1. all the methods like invoke-super, invoke-static, invoke-virtual are converted into Java form.

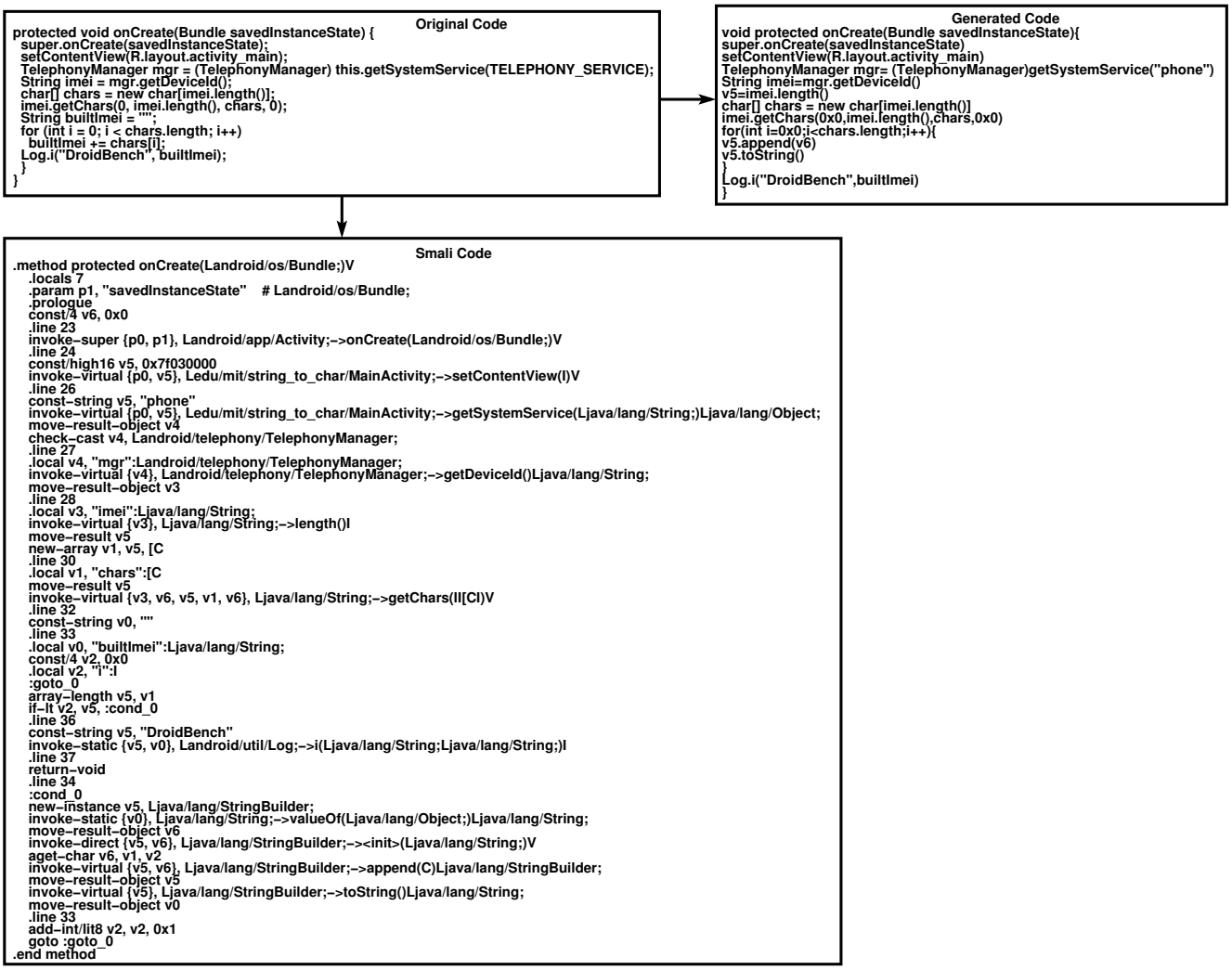

Figure. 8: **StringToCharArray1.apk onCreate() Method**

2. all the statements like if, for etc. are converted into their respective Java form

3. all the try-catch blocks are identified and converted into their respective Java form.

The system the does the things at its best level but a best system can have some flows and these flows can be used in future by analyst to make the system more reliable. Some of the interesting research directions generated by this work can be explained in the following points.

1. The system generated can be used for control and data flow analysis of the applications easily. Since the Java code can be easily read and it can provide better results than the previous static analysis technique using the intermediate code.

2. The analysis of the applications works very fine for multiple apps. For example the testing for multiple applications of test suite like Droidbench was accomplished and it worked well. But when the number of applications are very high or if they are very big applications like Facebook. The system might not work which is due to memory overrun not due to logic. In essence the system can be further optimized so that it should not fail for any number of applications an for any size applications.

3. Further investigations can be applied to the system, by using the different data mining algorithms like decision tree, k-means, apriori etc. to classify the application as a benign or suspicious by statically analysing them.

References

[1] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[2] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1):131–170, 1996.

[3] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[4] Alexandre Bartel, John Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *Software Engineering, IEEE Transactions on*, 40(6):617–632, 2014.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[6] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark*, 2008.

[7] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104. IEEE, 2012.

[8] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. Springer, 2012.

[9] Chunfu Jia, Zhi Wang, Kai Lu, Xinhai Liu, and Xin Liu. Directed hidden-code extractor for environment-sensitive malwares. *Physics Procedia*, 24:1621–1627, 2012.

[10] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Manoj Singh Gaur, Marco Conti, and Muttukrishnan Rajarajan. Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 414–421. IEEE, 2014.

[11] Qi Xi, Tianyang Zhou, Qingxian Wang, and Yongjun Zeng. An api deobfuscation method combining dynamic and static techniques. In *Mechatronic Sciences, Electric Engineering and Computer (MEC), Proceedings 2013 International Conference on*, pages 2133–2138. IEEE, 2013.

[12] Atefeh Nirumand, Bahman Zamani, and Behrouz Tork Ladani. Vandroid: A framework for vulnerability analysis of android applications using a model-driven reverse engineering technique. *Software: Practice and Experience*, 49(1):70–99, 2019.

[13] Marwa Ziadia, Jaouhar Fattahi, Mohamed Mejri, and Emil Pricop. Smali: an operational semantics for low-level code generated from reverse engineering android applications+. *Information*, 11(3):130, 2020.

[14] Eman K Elsayed, Kamal A ElDahshan, Enas E El-Sharawy, and Naglaa E Ghannam. Reverse engineering approach for improving the quality of mobile applications. *PeerJ Computer Science*, 5:e212, 2019.

[15] Roni Mateless, Daniel Rejabek, Oded Margalit, and Robert Moskovitch. Decompiled apk based malicious code classification. *Future Generation Computer Systems*, 2020.

[16] Amr Amin, Amgad Eldessouki, Menna Tullah Magdy, Nouran Abdeen, Hanan Hindy, and Islam Hegazy. Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information*, 10(10):326, 2019.

[17] DONG Zhenjiang, WANG Wei, LI Hui, ZHANG Yateng, ZHANG Hongrui, and ZHAO Hanyu. Sesoa: Security enhancement system with online authentication for android apk. *ZTE Communications*, 14(S0):44–50, 2019.

[18] Yauhen Arnatovich, Hee Beng Kuan Tan, Sun Ding, Kaiping Liu, and Lwin Khin Shar. Empirical comparison of intermediate representations for android applications. In *SEKE*, pages 205–210, 2014.

[19] Nguyen Tan Cam, Pham Van Hau, and Tuan Nguyen. Android security analysis based on inter-application relationships. In *Information Science and Applications (ICISA) 2016*, pages 689–700. Springer, 2016.

[20] Xingmin Cui, Da Yu, Patrick Chan, Lucas CK Hui, Siu-Ming Yiu, and Sihan Qing. Cochecker: Detecting capability and sensitive data leaks from component chains in android. In *Information Security and Privacy*, pages 446–453. Springer, 2014.

Table 3: **Dalvik opcodes used in the system**

S. No.	Opcode	Description
1.	move vi, vj	Places contents of register vj into register vi (range256).
2.	move/from16 vi, vj	Places contents of register vj into register vi.(vj range64 K, vi 256).
3.	move-object vi,vj	Places object reference vj into vi.
4.	move-object/from16 vi,vj	Places object reference vj into vi.(vj range 64 k, vi256)
5.	const/16 vi,#ConstValue	Puts the literal value (16 bits) into the specified register vi(16 bits).
6.	const-class vi, ID	Puts the object of the class specified by the ID intothe register(8 bit).
7.	const-class vi, StringID	Puts the String reference specified by the StringIDinto the register(8 bit).
8.	iput-object vi,vj,ID	Puts the object referenced by vj specified by ID intothe vi.
9.	move-result-object vi	Result of the invoke-... to be put into the specifiedregister vi.
10.	const/4 vi,#ConstValue	Puts the literal value (4 bits) into the specified registervi(4 bits).
11.	const/high16 vi,#ConstValue	Puts the literal value (signed int 16 bits) into thespecified register vi(8 bits).
12.	new-instance vi, ID	Puts the object created specified by ID into the register vi(8 bits).
13.	iget-object vi, vj, ID	Gets the object referenced by vj at the offset specified by ID into the vi.
14.	invoke-direct{arguments/parameters},methodID	Used to call direct methods which cannot be overridden.For example: constructor.
15.	invoke-virtual{arguments/parameters},methodID	Used to call normal method. For example non-staticmethods, non-final methods.
16.	invoke-super{arguments/parameters},methodID	Used to call normal method (virtual) of closest superclass.
17.	invoke-static{arguments/parameters},methodID	Used to call static methods.
18.	invoke-interface{arguments/parameters},methodID	Used to call interface method.
19	invoke-virtual/range {vi..vj},methodID	Used to call virtual method (virtual table). The parameters shows the range of register passed as anargument.

A. Smali Instructions

The table3 contains smali instructions which were used to generate the system and the analysis of the code. Different Dalvik opcodes with descriptions which are used by apktool (Baksmali) for generating the intermediate code are aforementioned.