Design an Optimized Compiler to Enhance Performance of Android Applications

Marwa Dahdouh
PhD Student
Dept. Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo, Syria

Mouhamad Ayman Naal
Associate Professor
Dept. Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo, Syria

Souheil Khawatmi
Associate Professor
Dept. Systems and Computer Networks
Faculty of Informatics Engineering
University of Aleppo, Syria

Amer Bouchi
Assist Professor
Dept. Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo, Syria

ABSTRACT

This paper presents a detailed study of the mechanism to design a compiler of Smali language to generate optimized Android applications. Smali language; which includes the dex bytecode; is the assembly language under Android OS, it is generated from the Java source code. The phases of designing the target compiler are described and the structure of files that are the input and output of the compiler are explained.

The structure of the input files of ART (Android RunTime) compiler is explained, with the focus on the dex file (Dalvik EXecutable) and its corresponding Smali language file, that includes dex bytecode code. The proposed compiler, Which is called MySMALI compiler, generates optimized Smali code by replacing some blocks in the Smali code by other blocks more efficient in performance and equivalent in behavior with original blocks. Reverse Engineering techniques are used to decompile and verify the correctness of the generated optimized APKs. As result, an optimized compiler is designed and the experimental evaluation shows that the compiler is able to save from 4.8% to 12.9% of the overall execution time in various application scenarios. This ratio of improvements increases up with the size and complexity of the optimized code.

General Terms

Reverse Engineering, Compiler optimization, Android Application Performance.

Keywords

Design Compiler, lex & yacc, Lexical analyses, Smali language, bytecode optimization, APK Decompiler,

1. INTRODUCTION

Over the past two decades, many important changes and challenges have taken place in mobile devices, which have limited processing power, memory, and battery life. So the optimization of mobile applications for better performance is considered a critical task.

Android is an operating system designed for mobile devices and acquired by Google in 2005 [1]. It's now widespread, and lots of applications are developed each day. First it ran DVM (Dalvik Virtual Machine) and had different types of compilers. From Android 2.2, JIT (Just-In-Time) Compiler was added to improve the execution time, it translates the source code into native code during runtime. JIT compilers were of two types, MB-JIT (Method-Based Just In Time) and TB-JIT (Trace-Based Just In Time). Second, ART environment was introduced in Android 4.4.

ART supported multiple compilers based on AOTC (Ahead Of Time) compilation [2]. AOTCs are divided into two categories: standalone-mode and mixed-mode. A standalone-mode AOTC compiles whole application into native codes, while a mixed-mode AOTC compiles only hot methods of code [3]. The Icing framework was released in 2011, it is based on MB_JIT compiler. In 2012 hybrid JIT framework of Android proposed the combination of MB-JIT and TB-JIT techniques to achieve a great performance [4]. ART is the default execution environment instead of Dalvik from Android 5.0, Lollipop version [5]. ART supports two backend compilers, which were a quick compiler and an optimizing compiler [6].

In this article new compiler is designed for Smali (Assembly Android) language. Smali is an ASM-like language used to represent dex bytecode instructions as a human readable language [7]. The goal of the proposed compiler is to generate optimized Smali code. The execution time of dex bytecode instructions was measured in different working environments. The results were analyzed and compared, and instructions with the smaller execution time were defined to be adopted in the MySMALI compiler.

The rest of the paper is organized as follows. Section 2 defines the files of Android compiler. Next, in Section 3 an overview of this approach steps is given and explained, the practical phases of designing a MySMALI compiler are described in section 4. Section 5 illustrates the input and output files of MySMALI compiler. While the proposed algorithms are explained in Section 6. Finally, experimental results of MySMALI compiler are discussed and analyzed in Section 7 and conclusion is given in Section 8.

2. ANDROID COMPILER FILES

Android OS uses many files during the compilation, the input and output files of Android compilation process vary according to the executable environment of Android.

1. **(.class) file**: This file contains JVM (Java Virtual Machine) instructions or Java bytecode and the symbols table, as well as other ancillary information [8].
2. **(.dex) file**: The dex file format is made up of several sections, which are used to hold a set of class definitions and their associated adjunct data [9].

Parsing the dex file is a complex task as it involves running bytecode and selecting relevant data [10].

3. **(.odex) file**: The odex (Optimized DEX) file is generated by dexopt tool in DVM to optimize Dalvik bytecode instructions. In ART environment, oat file is generated instead of it [11].
4. **(.apk) file**: APK (Application PacKage) file is used to distribute applications in Android [12] [13].
5. **(.smali) file**: Smali code is based on Jasmin syntax and usually saved in text format, is a human readable representation for dex bytecode [14].
6. **(ELF) file**: ELF (Executable and Linking Format) file is created by the assembler and link editor. ELF is object file format participates in building and running a program. The object files are binary representations of programs.
7. **(.oat) file**: OAT (Of-Ahead Time) is generated by dex2oat tool in ART [15]. This ELF file includes the main information of dex file and native code.

The compiler of Android OS is developed upon many stages. Table 1, summarizes the compiler development.

Table 1. Evolution of Android compiler

year	Compiler	Environment	Advantages
2008	JIT Interpreter	Dalvik	simple
---	MB-JIT	Dalvik	better than JIT interpreter
---	TB-JIT	Dalvik	speed, performance default environment
2011	MB- AOT	Icing framework with Dalvik	Three times faster than past
2012	Hybrid JIT	Dalvik	optimized code
2013	AOT or JIT	ART or Dalvik	dex2oat is defaul
2014	AOT	ART(Android 5.0)	faster by %80
2015	quick and Portable	ART fuzzing framework released	quick is default dex-to-dex dexFuzz
	merging JIT and techniques	T2R framework	improvement the execution
		static bytecode optimization	
	optimized backend	energy optimization framework	improve the energy processor
2017	optimized backend	compiler-based app instrumentation	application protection

Many approaches were accomplished on optimization issues for enhancing the Android applications performance, however the techniques of production compilers still are challenging [16]. The efforts proceed to combine the improvement of compilers with other techniques. Such as, in 2015, T2R framework has released, which improved the execution performance of applications by 10.5-16.2% and decreasing the size of the code cache by 4.6-28.5% [17]. Source-level Energy-optimization framework was proposed in 2016, saved from 6.4% to 50.2% of the CPU energy consumption [18]. In 2017, ARTist (Android Runtime Instrumentation and Security Toolkit) framework introduced as a compiler-based application for security solutions [19].

Table 2, shows the comparison between compilers in the two runtime platforms [20].

Table 2. Comparison between Android compilers

Advantages	AOT (ART)	JIT (Dalvik)
performance	app occupies a smaller memory footprint, faster execution and shorter launch time	extra memory for JIT code cache, slower execution
battery life	frees the CPU, so leads to a longer battery life	JIT compilation is CPU bound
installation time	dex bytecode is translated during the app installation	The translation is done during the app execution
storage footprint	keep a larger storage	keep a smaller storage

3. GENERATION THE OPTIMIZED APKS

All steps for generating optimized APKs, from writing a Java source code to decompilation, are illustrated in Fig 1.

First, Java program is written and its corresponding APK file is built using Android Studio v3.5. Second, APK file is decompiled using APK_Easy Tool v1.55. APK_Easy Tool v1.55 is a reverse engineering tool, latest version [21]. During the decompiling process of APK [22], many Smali files and folders are extracted inside the decompiled Smali folder. Such as AndroidManifest.xml file that informs the system about the application components. Third, Smali files are opened, analyzed, modified and saved. In this approach, the modification isn't done manually in a random form. Indeed, MySMALI compiler applies the required modification in Smali files based on the proposed replacement technique. Fourth, repackaging of Smali code and folders is done by reverse engineering tool. APK_Easy tool is reused again to recompile and reassemble the modified Smali source files. The optimized APK file is generated to enhance the performance of Android applications.

After the recompiling, APK_Easy tool allow two options: ZipAlign and sign. Android doesn't allow installation APK file of an application that isn't signed [23]. As result, all optimized APKs be digitally signed with a special certificate. APKs can be installed on any mobile device to calculate the execution time of an application instructions.

Three different files are generated by MySMALI compiler. These files are as the following:

OptZCompiled_output.smali is the optimized Smali file. This file is renamed as the input file (sourcefile_Name.smali) and replaced instead of the original file into the decompiled Smali folder. OptZCompiled_output is reassembled with other decompiled folders using the reverse engineering tool to generate the optimized APK file.

LogOutputfile.txt file file contains all the information stored in the base symbol table and the modified symbol table information resulting from the implementation of the replacement algorithm.

CompiledOutputfile.smali file includes the source code of the input file. This file is generated after all phases of analysis (lexical, syntax, semantic) have finished by MySMALI.

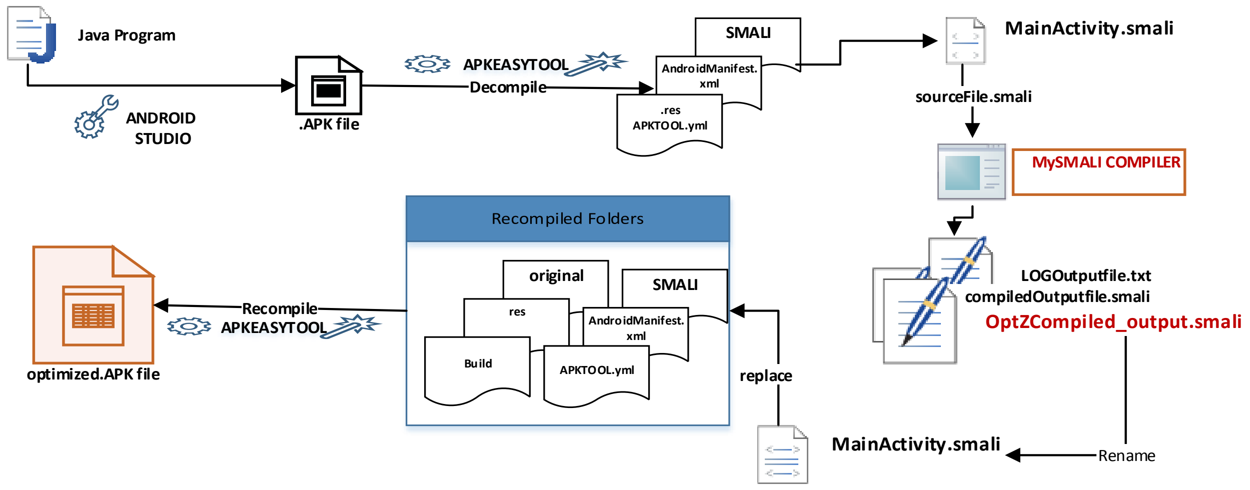

Fig 1: Steps of generating an optimized APK file

4. COMPILER DESIGN

The optimized compiler steps are described in Fig 2 [24].

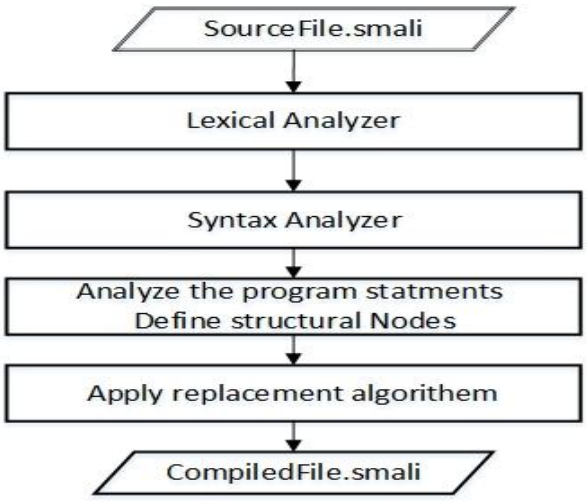

Fig 2: Flowchart of designing process

The optimized compiler is designed by using Lex (LEXical analyzer) and Yacc (Yet Another Compiler-Compiler) tools. EditPlus v5.2 (build 2281) 64-bit and Flex Windows (Lex and Yacc) tool are used. Smali instructions, keywords and identifiers are defined to be used in specification and tokens recognition [25]. Lexical, syntax, semantic analysis and other phases of designing a compiler are applied [26].

At the beginning, an input Smali source file is translated into tokens by Lex. Tokens are utilized to generate a syntax analyzer. Then Smali language grammars are formulated [27]. Structures are extracted from the analyzing process of Smali files. AST (Abstract Syntax Tree) is constructed [28]. At last, the proposed replacement technique is applied depending on the structures. Then a Smali target file of the compiler is produced as output file. The proposed instructional structures allowed the optimized compiler to perform a better analysis and replacement. The process of identifying the appropriate structures for the bytecode instructions is one of the most important stages.

The practical steps of building the optimized compiler can be summarized as follows.

First, Lex file compilation process is performed with the statement: *flex MySmali.l*. A C language file (Lex.yy.c) results from the compilation.

Second, Yacc compilation process is performed with the statement: *Bison –dy MySmali.y*. As a result of the compilation, two files (Y.tab.c, y.tab.h) are generated.

The final stage of building process is accomplished through instruction: *gcc lex.yy.c y.tab.c*. As a result, (MySmali.exe) file is generated. Fig 3 shows these steps.

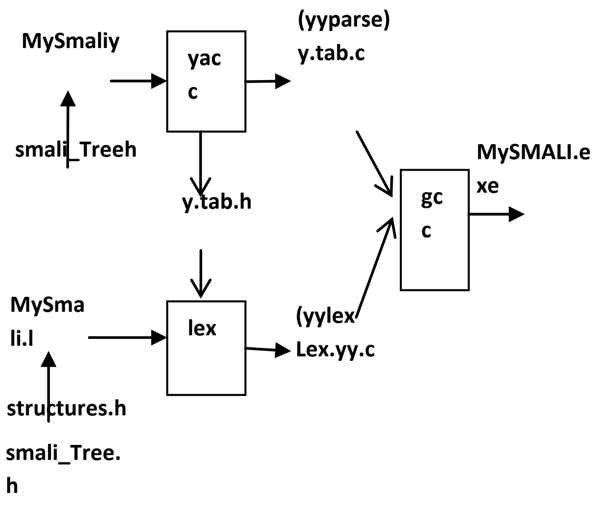

Fig 3: Steps of building MySMALI compiler by Lex/Yacc

5. MYSMALI COMPILER FILES

5.1 Smali.L file

The regular expressions are represented into this file. Regular expressions formally describe the recognized tokens that may be founded in Smali source code. Rules section; the significance part; is declared. Each rule is specified by a pattern and its corresponding action.

Table 3, illustrates some of the proposed regular expressions of the declared patterns.

Table 3. The declared regular expressions

Pattern	Regular Expression
COMMENT	#.*
JAVA_ID	[\[]?[L][_a-zA-Z][_a-zA-Z0-9/$]*;
ID	[_a-zA-Z][_a-zA-Z0-9$]*
Register_ID	(v|p)[0-9]+
CONSTRUCT_INIT	<[_a-zA-Z][_a-zA-Z0-9]*\>
DOUBLEQ_STRING	\"([^\\\"\r\n]|\\[^\r\n])*\"?
HEX_NUMBER	[+-]?0[xX][0-9a-fA-F]+(L|s|t)?
Sml_NUMBER	([0-9]*\.?[0-9]+|[0-9]+\.[0-9]*)([eE][+-]?[0-9]+)?
ARRAY_TYPE	[\[]+[a-zA-Z]+
PSWITCH_LABEL	":pswitch"[_a-zA-Z0-9]+
LABEL	(":cond"|":goto")[_a-zA-Z0-9]+

The user declarations section is includes some macros for defining the location of errors. Such as, YY_USER_INIT macro is evaluated at the first invocation of yylex. YY_USER_ACTION macro is executed each time a rule matches.

5.2 MySmali.y

This file includes a definition section and production rules section [29]. Rules section includes the grammars of Smali language. Grammars are defined to specify how to understand the input source code and what actions to take for each rule. Types of tokens were defined, such as in Fig 4.

```
%token <s_value>

DOT_CLASS  DOT_SUPER  DOT_SOURCE
DOT_IMPLEMENTS DOT_ANNOTATION  DOT_FIELD
DOT_METHOD  DOT_END_METHOD …

%token

SEMICOLON OPEN_BRACK CLOSE_BRACK …

%type <nPtr>

Methods Method_struct Method_prototype Method_Body
direct_method virtual_method variables_List return_type
PSWITCH_LABELS packed_switch_statments  …

%%
```

Fig 4. Types of tokens declaration

Fig 5. displays a piece of source code. The code includes a declaration of a direct method in Smali language. The parser tree of the source code is constructed, such Fig 6.

```
method public constructor <init>()V  # direct methods

locals 0

line 7

invoke-direct {p0},
Landroid/support/v7/app/AppCompatActivity; -> <init> ()V

return-void

end method
```

Fig 5. A direct method in source Smali code

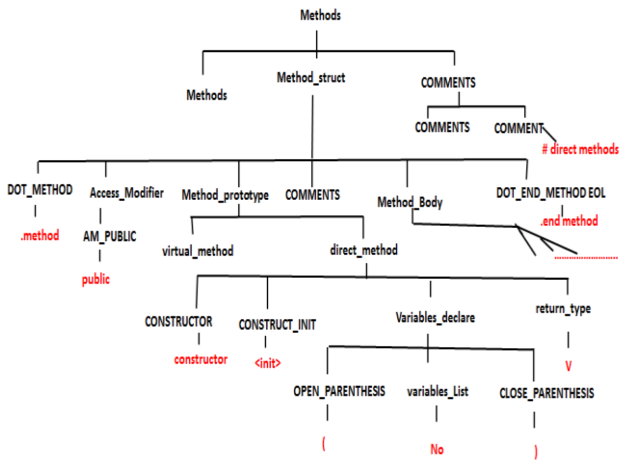

Fig 6. Abstract Syntax Tree of a direct method

Yacc derives a parser tree based on the matching production rules. Table 4 displays some matched rules of a direct method.

For the best results for MySMALI compiler, the production rules are defined so that they do not include any conflicting rules of shift/reduce type or reduce/reduce type.

Table 4. Production rules of method instructions

Rule	Action, corresponding to
methods	: Methods Method_struct COMMENTS {$$=CompNode_define(three_State,3,$1,$2,$3);} \| {$$=NULL;} /*Nothing*/
method_ struct	: DOT_METHOD Access_Modifier Method_prototype COMMENTS Method_Body DOT_END_METHOD EOL
method_ prototype	: direct_method | virtual_method
direct_ method	: CONSTRUCTOR CONSTRUCT_INIT Variables_declare EOL | CONSTRUCTOR CONSTRUCT_INIT Variables_declare return_type EOL | CONSTRUCT_INIT Variables_declare return_type

return_type	: JAVA_ID {$$=stringnNode_define($1);}
	\| primitive_type {$$=$1;}
	| void_type {$$=stringnNode_define($1);}

5.3 Structures.h

This file is included in MySmali.l file and contains the declaration of the suggested structures for Smali language instructions.

During the compiler's analysis of the input file, each node in the symbol table is analyzed and its type is determined in order to build the appropriate structure according to its type. Table 5, shows some structures of nodes.

Table 5. The proposed structures of some instructions

Typedef	Struct declaration	Description

BodyNODE	char *st_name;
	int st_size;
	int lineno;
	char *token_type;
	struct BodyNODE *next;
		Structure of Symbols Table

Methods _struct	char *Dot_Begin;
	char *Register_Num;
	char *Locals_Num;
	int line_Begin; int line_End;
	int index_Begin;
	int index_End;
	struct methods_struct *next;
		Structure of each declared method

IF_struct	char *if_name;
	int Line_Num;
	int index_Begin;
	char *reg_1; char *reg_2;
	char *cond_label;
	struct IFInfo_struct *IFInfo;
	struct List_instruction body;
		Structure of IF statement

goto_struct	char *gotolbl_name;
	struct gotoInfo_struct *gotoInfo;
	struct List_instruction *body;
		Structure of goto statement

Reg_struct	char *Reg_name;
	int Reg_index;
	char *Reg_Kind;
	char *return_type;
	struct Reg_struct *next;
		Register Structure

Loc_struct	char *Loc_name;
	int B_index;
	int E_index;
		Structure of Local

The suggested packed_switch structure consists of three interconnected and overlapping structures. The structures are: switch_struct, table_struct and branch_struct. Fig 7. shows the suggested structure of packed_switch instruction.

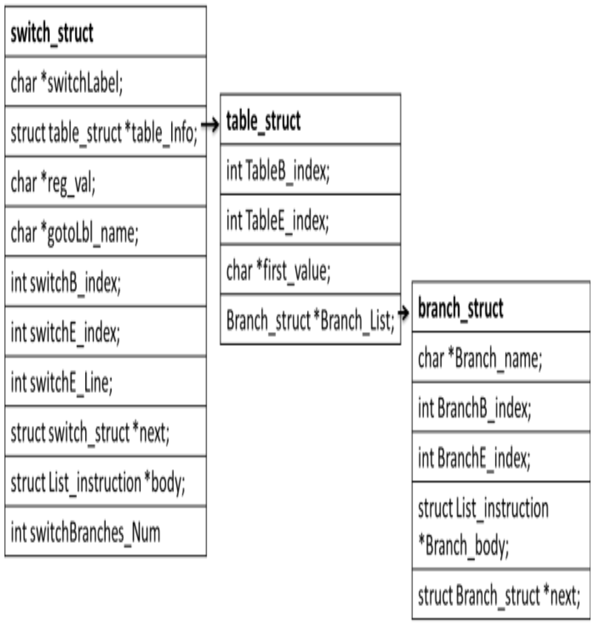

Fig 7. The proposed structure of packed_switch

The switch instruction structure includes several fields to store the basic information that will be used when the replacement algorithm is implemented. Some basic stored information are: the number of last line of the switch instruction block within the input file (switchE_Line), first index in the symbols table in which to start storing the block (switchB_index), last index (switchE_index), total number of branches (switchBranches_Num). This structure also includes a pointer to the table_struct structure (table_Info).

The table_struct structure contains information about all branches of packed_switch instruction. The information are: the reference to the list of branches that includes all the instructions to be performed when matching with each branch (Branch_List), first value to compare with when evaluating the first branch of a packed_switch instruction block (first_value), first index in the symbol table where the branch table information is stored within (TableB_index), last index (TableE_index).

The branch_struct structure contains information about each branch. The information stored in this structure is as follows: a name of the branch (Branch_name), the first index in the symbols table in which to start storing the branch (BranchB_index), the last index in which the comparison branch instructions are stored in the table (BranchE_index), the instruction block for each branch to be approved later in the replacement phase within the proposed if structure (Bracnch_body).

5.4 Smali_Tree.h

The suggested structures for nodes types are declared in this header file. These structures are specified during the creation process of the symbols table. Two types of simple nodes are declared (*simple_inode int_value, simple_Snode string_value*). One type of complex node is declared (*compound_node compound_value*). The complex type includes a number of operators of each node and a reference for each operator. Compound instructions have been processed such as, jump statement (goto), conditional statements (if, ifz), selection statement (packed-switch).

For example, If Lexer analyst finds the following packed-switch statement: *packed-switch v6, :pswitch_data_0*.

The characters of this instruction will be divided into the corresponding tokens and type of each token will be determined. Finally information about tokens is stored in the symbols table. The symbols table will include the following information as in Table 6.

Table 6. Symbols table of packed-switch declaration

token name	**token index in symbols table**	**line number in a source file**	**token type**
packed-switch	i= 238	83	T_PACKED _SWITCH
v6	i+1= 239	83	T_Reg_ID
,	i+2= 240	83	COMMA
:pswitch _data_0	i+3= 241	83	T_SWITCH LABEL

6. PROPOSED ALGORITHMS

MySMALI compiler used several algorithms to implement a proposed replacement technique. The replacement technique involved several steps to modify specific instruction blocks structures. So they are replaced by instruction blocks with less execution time and similar behavior. The proposed instruction blocks structure which will be modified are:

- A packed-switch instruction block that consumes a larger execution time will be replaced by its equivalent if_block instruction. If_block instruction consumes less execution time.
- The replacement technique depends on conditional overlapping instructions with a multi-way and two-way if_else structure. The structure of a one-way conditional instruction structure isn't replaceable because it causes a change in instructional behavior. If_block conditional statements are replaced by following conditional instructions (if-ne, if-ge, if-le, if-nez, if-gez, if-lez) that consume less execution time.
- The structure of nested instruction blocks within Smali language, which is equivalent to the repetitive behavior of instructions (for, do-while) in Java language. This structure is suggested to be replaced by the structure of instruction block equivalent to (while) instruction that is defined as consuming less execution time.

After the analysis of Smali code of the input file, the instruction structures have been built and the proposed structures have been identified to replace their instructions in order to reduce execution time. Optimized compiler applies the replacement technique using the following algorithms:

6.1 Proposed algorithm for discovering structures

MySMALI compiler applies a structures discovery algorithm. There are several steps of this algorithm.

Methods structure is constructed and instructions are analyzed in the input file for each method. All instructions for the most efficient structures are analyzed and a global list of each instruction is constructed. The efficient structures are*: goto, if-eq, if-ge, if-le, if-eqz, if-gez, if-lez, registers, Local, packed_switch*. These instructions are considered to have a programmatic effect on MySMALI compiler. So they are proposed to be modified by the replacement technique.

The basic structure of the proposed algorithm is illustrated by a flowchart described in Fig 8.

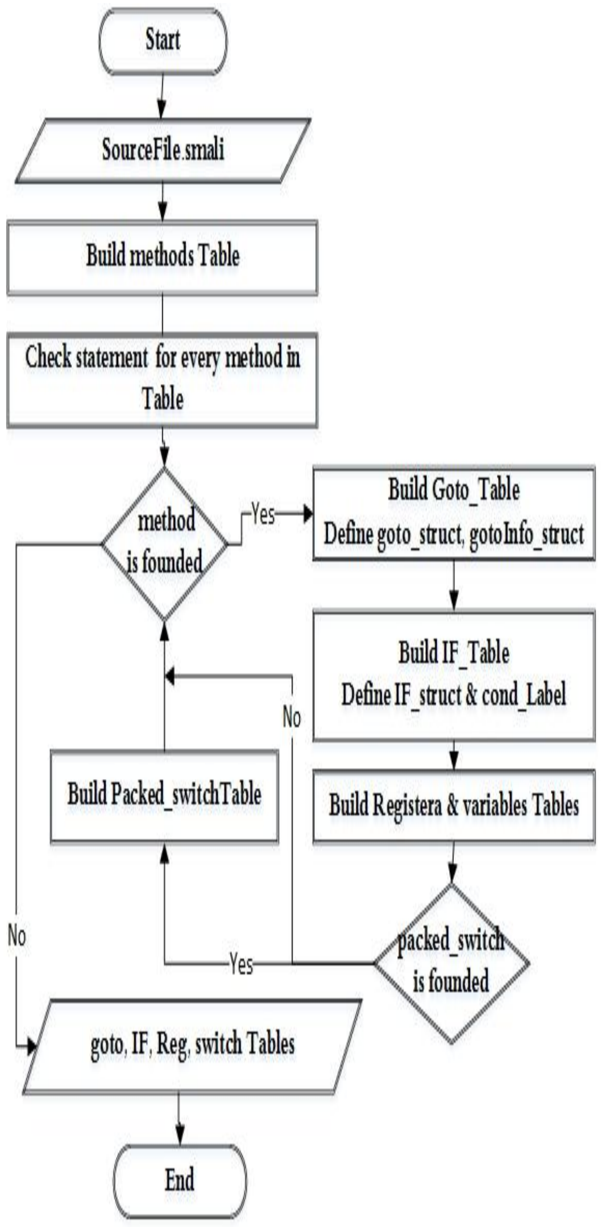
Start
SourceFile.smali
Build methods Table
Check statement for every method in Table
method is founded
Yes
No
Build Goto_Table
Define goto_struct, gotoInfo_struct
Build IF_Table
Define IF_struct & cond_Label
Build Registera & variables Tables
packed_switch is founded
Yes
No
Build Packed_switchTable
goto, IF, Reg, switch Tables
End

Fig 8. Flowchart of structures discovery algorithm

6.2 Proposed replacement algorithm

Fig 9. Shows steps of the replacement algorithm for replacing a packed_switch instruction with the equivalent if_block.

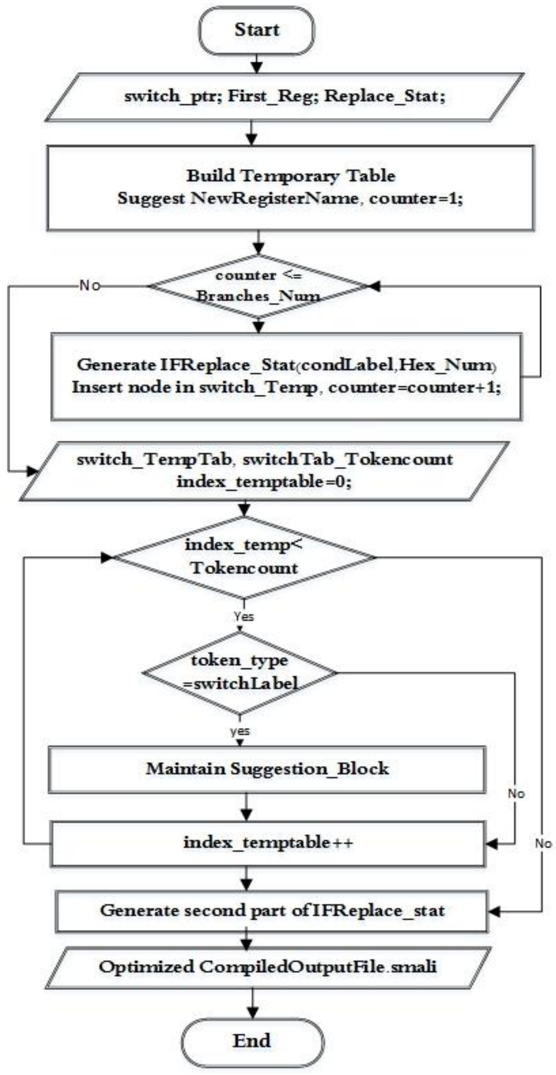
Start
switch_ptr; First_Reg; Replace_Stat;
Build Temporary Table
Suggest NewRegisterName, counter=1;
counter <= Branches_Num
No
Generate IFReplace_Stat(condLabel,Hex_Num)
Insert node in switch_Temp, counter=counter+1;
switch_TempTab, switchTab_Tokencount
index_temptable=0;
index_temp< Tokencount
Yes
token_type =switchLabel
yes
Maintain Suggestion_Block
No
index_temptable++
No
Generate second part of IFReplace_stat
Optimized CompiledOutputFile.smali
End

Fig 9. Flowchart of replacement algorithm

The replacement algorithm is applied and proposed replaceable instruction blocks are replaced by the equivalent blocks of behavior. The steps of packed_switch replacement algorithm can be summarized as follows.

Step1: A temporary symbols table is constructed and the desired modification is applied.

Step2: Replaced if_block is built to include all the basic information that is supported in the replacement process. The basic information are: names of new registers, hexadecimal values that are compared with each branch, goto labels statement and condition labels of if statements, etc.

Step3: The second part of the replaced if_block is generated instead of the switch statement. This part contains all the instruction blocks to be executed when each condition of the if_statement is valid. So, final form of the proposed replacement code has been completed.

7. DISCUSSION AND ANALYSIS EXPERIMENTAL RESULTS

7.1 MySMALI compiler

Different Smali files were generated to study each individual instructions and instructions blocks. Two types of source files were distinguished within this study. Simple file includes individual instructions and compound file includes structural blocks of instructions.

The test process was applied to 139 simple source files, and 100 compound files. All input files result from unpacking APK files so they are correct and error free. Assuming MySMALI compiler detects an error, the compilation process will be stopped immediately. Then first grammatical error will be located, and an alert message will be printed indicating the line number and column that included the error. Table 7, displays the percent ratio of correction.

Table 7. Results of MySMALI compiler

type of source file	number of tested files	percent of correction
simple file	139	%100
compound file	100	%100

As result, experiment proved the following:

- Some source code instructions such conditional statements (if-eq, if-lt, if-gt) and IFz (if-eqz, if-ltz, if-gtz) consume more execution time. These instructions should not always be replaced by instructions with less execution time consumption (if-ne, if-ge, if-le, if-nez, if-gez, if-lez). Replacing process of one-way conditional statement changes the original code behavior.
- The common-purpose instruction block is distributed within the source code of Smali language file in non-sequential lines of code. Collecting process of all single blocks instructions sequentially led to grammar errors in Smali language and was not useful. Errors prevent the reassembling process of files and prevent the optimized APKs generation. So MySMALI compiler did not compile the instruction blocks in successive lines.
- Verification and validation of APK files proved that the code behavior after applying the replacement algorithm is equivalent to the behavior required from the original code. The Android applications that are reassembling their files and folders using APKTool is more than preserves its original behavior as has been proven [30].

7.2 Optimized Smali file

After MySMALI compiler generated optimized Smali files, the corresponding optimized APKs are generated by APK_Easy Tool. Experiments have shown that all APK files have been successfully executed on a real mobile device and do not contain any bugs. Table 8, illustrates comparison of Smali generated code between MySMALI compiler and other compilers.

Table 8. Comparison generated Smali code between MySMALI compiler and other compilers

Smali code corresponding (to)		
Android Studio (v3.1.1 , v 3.5)	APK_Easy tool 5.5	MySmaliCompile

packed-witch v6, :pswitch_data_de .line 25 const/16 v6, 0xf goto :goto_30 .line 23 :pswitch_2c const/16 v6, 0xa goto :goto_30 .line 21 :pswitch_2f const/4 v6, 0x5 .line 26 :goto_30 #...second part :pswitch_data_de .packed-switch 0xa :pswitch_2f :pswitch_2c .end packed-switch	packed-switch v6, :pswitch_data_0 .line 25 const/16 v6, 0xf goto :goto_1 .line 23 :pswitch_0 const/16 v6, 0xa goto :goto_1 .line 21 :pswitch_1 const/4 v6, 0x5 .line 26 :goto_1 #...second part :pswitch_data_0 .packed-switch 0xa :pswitch_1 :pswitch_0 .end packed-switch	const/16 v0 , 0xa if-eq v6 , v0, :cond_3 const/16 v0 , 0xB if-eq v6 , v0, :cond_4 .line 25 const/16 v6 , 0xf goto :goto_1 .line 23 :cond_4 const/16 v6 , 0xa goto :goto_1 .line 21 :cond_3 const/4 v6 , 0x5 .line 26 :goto_1

Reverse engineering of APK files (decompile and recompile) is done using following tools:

- APK_Easy tool v1.541 (2018-09-16), APKTool 2.3.4.
- APK_Easy tool v1.55 (2019-05-09), APKTool 2.4.0, latest version.

A case study is conducted to test packed_switch statement using MySMALI compiler. Several applications are implemented that included different states of switch instruction. Cases of packed_switch instruction are studied according to a number of branches compared with them (two, three, four or five branches). Some applications also included switch instruction structure that is repeated a number of times. Applications containing switch instructions are implemented in different study cases and execution times are measured with an average of 1000 repetitions per execution. The results are displayed in Table 9.

Measurement of execution times is performed on a real mobile device with the following specifications: (Galaxy Grand Prime+, Android 6.0.1 , API 23). To ensure that applications are executed within an ART environment. minSdkVersion minimum SDK (Software Development Kit) version is defined to be executed on 21, compileSdkVersion is set to 26. The Android applications are implemented on a device with the following specifications: (CPU: Intel Core i5, System: Windows 7 Ultimate 64- bit, RAM: 8GB).

Comparison of original APK execution times with optimized APK implementation times showed that optimized files consume less execution time. Experience proved the efficiency of optimized MySMALI compiler and that the optimized compiler achieved desired results.

Table 9. Comparison execution times between original APKs and optimized APKs

statement cases	Averages of execution times of APKs (nano_second)			
packed-switch	Before	After	profit time	percent optimization
2 branches	1513.527	1440.473	73.05	% 4.826
3 branches	1547.394	1457.994	89.4	% 5.777
4 branches	1568.761	1465.71	103.1	% 6.568
5 branches	1611.225	1453.249	158	% 9.804
repetition 2_switch	1655.38	1522.922	132.5	% 8.001
repetition 3_switch	1674.601	1458.173	216.4	% 12.924

Fig 10, shows a comparison of the average rate of original and optimized APK files execution times and how the execution time for applications has been improved.

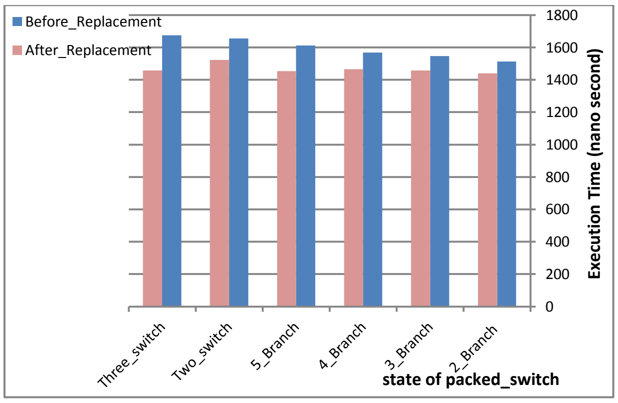

Fig 10: Comparison of execution times for applications before and after using replacement technology

8. CONCLUSION

In this paper, an optimized Smali compiler is designed. This research improves application performance by generating Smali optimized files and corresponding optimized APK files.

The research consists of 1) building a lexical analyzer that recognizes all vocabulary of Smali language; 2) parsing several Smali files to formulate all grammars; 3) building a syntax analyzer that includes all rules formulated for Smali language; 4) developing algorithms to apply the proposed replacement technique; 5) designing an optimized MySMALI compiler that replaces dex bytecode instructions with instructions equivalent to behavior and consumes less execution time.

The original Smali files are generated using the reverse engineering tool to decompile APK files. MySMALI compiler generates optimized Smali files by applying the proposed replacement technology. Optimized Smali files have been reassembled with decompiled folders using APK_Easy tool to generated optimized APKs. The performance of original APK files are compared with optimized APKs performance. Comparison proved that optimized APK files do the same behavior with better performance. The results demonstrate the efficiency and accuracy of the optimized compiler and that MySMALI compiler has improved the performance of Android applications.

As for the future work, MySMALI compiler may be apply in different fields of reverse engineering for optimizing Smali code. Another future work may focus on using the proposed algorithms on static and dynamic analysis techniques of APKs. The mechanism that is used in this research can also be used to build a new tool. The suggested new tool allows modifying the performance of Android applications and hacking their information.

9. REFERENCES

[1] Ms. Debosmita Sen Purkayastha, Mr. Nitin Singhla, June 2013, "Android Optimization: A Survey, International Journal of Computer Science and Mobile Computing", IJCSMC, Vol. 2, Issue. 6, pg.46 – 52.

[2] Jeehong Kim, Inhyeok Kim, Changwoo Min, Hyung Kook Jun, Soo Hyung Lee, Won-Tae Kim, and Young Ik Eom, 2015, "Static Dalvik Bytecode Optimization for Android Applications", Institute for Information & Communications Technology Promotion (IITP), ETRI Journal.

[3] Chih-Sheng Wang, Guillermo A. Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih, 2011, "A Method-Based Ahead-of-Time Compiler for Android Applications", ACM 978-1-4503-0713.

[4] Chung-Min Kao, Wei-Chung Hsu, Yeh-Ching Chung, Guillermo A. Pérez, 2012, "A Hybrid Just-In-Time Compiler for Android, Comparing JIT Types and the Result of Cooperation", ACM.

[5] Abhishek Vasisht Bhaskar, 2016, "Automated code extraction from packed android applications", Syracuse University.

[6] CENSUS S.A, 2015, "Fuzzing Objects d'ART Digging Into the New Android L Runtime Internals".

[7] Aleksandr Pilgun, Olga Gadyatskaya, Stanislav Dashevskyi, and all, 2018, "Fine-grained Code Coverage Measurement in Automated Black-box Android Testing", University of Luxembourg, Luxembourg, arXiv:1812.10729v1 [cs.CR].

[8] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, 2014, "The Java Virtual Machine Specification". Addison.Wesley, Java SE 8 Ed. 600p.

[9] Google Inc.dex, 2007, "Dalvik Executable Format".

[10] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, Dinghao Wu, , 2016, "Adaptive Unpacking of Android Apps", The Hong Kong Polytechnic University, et al.

[11] Alexandre Bartel, Jacques Klein, Yves Le Traon, Martin Monperrus, 2012, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot", ACM, ISBN 978-1-4503-1490.

[12] Pooja Singh, Dr. Santosh Singh, Pankaj Tiwari, 2015, "Discovering Persuaded Risk of Permission in Android Applications for Malicious Application Detection", AMET University.

[13] Quan Qian, Jing Cai, Mengbo Xie, Rui Zhang, 2016, "Malicious Behavior Analysis for Android Applications", International Journal of Network Security, Vol.18, No.1, PP.182-192.

[14] Patrick Schulz, Daniel Plohmann, 2012, "Code Protection in Android", Institute of Computer Science, university of Communication and Distributed Systems.

[15] Geonbae Na, Jongsu Lim, Kyoungmin Kim, and Jeong Hyun Yi, "Comparative Analysis of Mobile App Reverse Engineering Methods on Dalvik and ART", Journal of Internet Services and Information Security, 2016.

[16] Yang Wenbo, Zhang Yuanyuan, Li Juanru, Shu Junliang, Li Bodong, Hu Wenjun, Gu Dawu, 2015, "AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware", National Science and Technology Major Projects of China.

[17] Yi-Ping You, Jian-Ru Chen, 2015, "A static region-based compiler for the Dalvik virtual machine", National Chiao Tung University.

[18] Xueliang Li, John P. Gallagher, 2016, "A Source-level Energy Optimization Framework for Mobile Applications", arXiv:1608.05248v1 (cs.SE).

[19] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, Sebastian Weisgerber, 2017, "ARTist:The Android Runtime Instrumentation and Security Toolkit", IEEE.

[20] Paul Sabanal, 2015, "Hiding Behind ART", IBM Security Systems, IBM X Force.

[21] https://forum.xda-developers.com/android/software-hacking/tool-apk-easy-tool-v1-02-windows-gui-t3333960. [Accessed 14/5/ 2019].

[22] Dewashish Upadhyay, et al, 2016, "Detecting Malicious Behavior of Android Applications", IJSTE, International Journal of Science Technology & Engineering , Volume 2, Issue 10, ISSN (online): 2349-784X.

[23] Chit La Pyae Myo Hein, 2014, "Permission Based Malware Protection Model for Android Application", International Conference on Advances in Engineering and Technology.

[24] Evgeniy Ilyushin, Dmitry Namiot, 2016,"On source-to-source compilers", International Journal of Open Information Technologies ISSN: 2307-8162.

[25] Malaga, Spain, 2018, "Model Checking Software", 25th International Symposium, SPIN.

[26] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, 2007, "Compilers principles, Techniques, & Tools", Second Edition, QA76.76.C65A37.

[27] https://github.com/JetBrains/android/tree/master/smali. [Accessed 14/2/ 2019].

[28] Lu CHEN, Xing LIU, Yuan-yuan MA, Cong-cong SHI and Ni-ge LI, 2016, "Research on Static Analysis Technology of Android Application Security Defects", International Conference on Electrical Engineering and Automation, ISBN: 978-1-60595-407-3.

[29] Annal Ezhil Selvi S, J . Persis Jessintha, 2018, "Compiler Design Concepts, Worked out Examples and MCQs for NET/SET",https://www.researchgate.net/publication/316560022.

[30] Yauhen Leanidavich Arnatovicha, et al, 2018, "Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation", IEEE Aceess, Digital Object Identifier 10.1109/ACCESS.2018.2808340.