INPUT DIVISION FOR BINARY TRANSLATION

BY

GENG DANIEL LIU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Wen-Mei W. Hwu

ABSTRACT

Binary translation is useful in migrating binaries to architectures different from the one they are originally compiled for. The work in this thesis is an optimization of an existing binary translator developed by Chen et al. in 2008. The goal of the binary translator is to allow Android applications with native code compiled for ARM architecture to run on MIPS-based hardware. The ideal time to translate an Android application is when it is being installed. Therefore, the binary translator must execute on a mobile device which has limited compute power. The original binary translator encounters a severe limitation when translating large applications. On those applications, translation takes more than one hour to complete. In the worst case, the translator crashes due to insufficient memory.

We present Input Division, an optimization technique that resolves the aforementioned issues. Input Division improved the original implementation with a more advanced input analysis technique that significantly accelerates output binary generation. As a result, we achieved up to 18.9X speedup in translation time and 48X reduction in memory usage.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Binary translation is a useful technique in many applications such as ISA migration and legacy code migration. In the context of this thesis, the goal of binary translation is to allow Android applications with C/C++ code compiled for ARM platform to run on MIPS platform. When installing an Android Application Package (APK) on a MIPS-based device, the Android application installer invokes the binary translator to convert any ARM-targeted binaries to MIPS-targeted binaries. Since the binary translator must execute on a mobile device, it faces major constraints on both translation time and memory usage. The LLVM-based Binary Translator (LLBT), developed by Chen et al. [1], was originally designed to execute on x86-based servers with abundant CPU power and memory. However, due to the lack of physical resources on cellphones and tablets, running LLBT on mobile devices became infeasible. This thesis describes Input Division, which is an optimization that makes LLBT practical for running on mobile devices.

The remainder of this document will be organized as follows. Chapter 2 will provide some background information on Android application with native code and ARM architecture. It will also describe LLBT at a high level. Chapter 3 will describe Input Division in detail. Chapter 4 will introduce an optimization of Input Division. Chapter 5 will discuss our experimental results. Chapter 6 will conclude the work.

CHAPTER 2

BACKGROUND AND MOTIVATION

In this chapter, we first explain why and how an Android application developer would use C/C++ code. Following that, we discuss some background information on ARM architecture. Finally, we will give an overview of LLBT and discuss the motivation of Input Division.

2.1 APK with Native Code

In general, Android applications are written in Java and they execute in Dalvik, the virtual machine in Google's Android Operating System [2]. Android NDK provides developers the ability to use native functions written in C/C++ as helper functions to the Java programs. Since C/C++ inherently executes faster than Java, application developers typically implement compute-intensive operations such as physics simulation and signal processing in native code [3]. According to a survey conducted by Shen et al., less than 10% of the APKs involve native code [4]. Android NDK compiles native code into shared objects that are packaged into the APK, which will be eventually downloaded by users. When users launch the application, the shared object will be loaded on demand.

2.2 ARM Embedded Application Binary Interface (EABI)

There are many ARM-specific features that make binary translation challenging. In this section, we will only discuss the ones that are pertinent to Input Division. For a comprehensive review, please see the ARM Architecture Reference Manual [5].

2.2.1 Symbol Tables

A shared object typically contains two symbol tables: static and dynamic. The static symbol table contains information on functions and data. The dynamic symbol table is a subset of the static symbol table and it contains the minimum amount of symbols required for dynamic linking. Therefore, it only holds symbols of exported functions and global data. Symbols of internal function, or functions that are not exported, are available from the static symbol table. However, since the static symbol tables are not necessary for program execution, they are often removed, or stripped, in order to minimize the size of a shared object.

2.2.2 Function Call

Although there are many ways to call a function, the most common way is by executing the Branch and Link (BL) instruction in the form of "BL immediate_value". In the example in Figure 1, BL first saves 0x304, the address of the next sequential instruction, in the Link Register (LR) which is a special register dedicated for return address. Then, it stores 0x400 in Program Counter (PC) and the program will continue at the entry of callee_function.

2.2.3 Function Return

There are multiple types of function return instructions and they are summarized in in Figure 2. Type 1 is used in leaf functions, or functions that do not call other functions. In this

```
0x200 <caller_function>
…
0x300: bl 0x400
0x304: add r3, r0, r1
…
0x400 <callee_function>
...
```

Figure 1. Example of Branch and Link

case, the LR register holds the current function's return address because it has not been updated since the entry of the function. Moving the content of LR into PC causes a branch to its caller. The BX instruction in Type 2 is a branch with an option to switch instruction mode. We will discuss instruction modes in section 2.2.4. Type 3 is for non-leaf functions that call other functions via BL instructions. Before calling another function, the current function must preserve its return address by pushing LR onto stack. At the current function's return, it will retrieve LR from stack and store it in PC.

2.2.4 ARM Instruction Modes

ARM ISA has two instruction modes: ARM (32-bit instruction) and Thumb (16-bit instruction). Mode switching happens during a program's execution via instructions such as "BLX" and "BX". BLX is a Branch and Link instruction with an option to switch mode. Besides achieving the semantics of BL, BLX uses the least significant bit (LSB) in the target address to determine the subsequent instruction mode. In the ARM ISA, all functions must start from even byte addresses. This allows the linker to record whether a callee function is an ARM function or a Thumb function by setting the LSB of the target address used by the BLX instruction. The program switches to Thumb mode if the LSB of the target address is 1. Otherwise, it remains in

Return instructions	Semantic
Type 1: mov pc, lr	pc = lr
Type 2: bx lr	pc = lr check lr[0] for mode switch
Type 3: pop {..., pc}	pop lr from stack and store it in pc

Figure 2. Three types of return instructions

ARM mode. Afterwards, the target address is cast to an even number by clearing its LSB and stored in PC. In Figure 3, in order to call the Thumb function at 0x200, r0 must be the callee's address with its LSB set to 1.

As for return, the return address is also set in a similar way. If the calling function is in Thumb mode, the processor sets the least significant bit of the return address to 1 before it is moved into the LR register. Therefore, if a BX instruction sees an odd return address, the processor switches into Thumb mode and assumes that it is returning into a Thumb function. However, if the calling function is in ARM mode, the return address remains as an even number. In Figure 3, at the end of the Thumb function, "BX LR" returns to 0x108. Since the value in LR is 0x108 which is an even number, BX will switch instruction mode back to ARM. In general, we do not know the instruction mode of a code block at translation time because we cannot always statically determine the operands' value for BLX and BX.

2.2.5 Special Symbols

If a shared object is compiled with debug option, its symbol table will contain special symbols, namely $a, $t and $d which stand for ARM code, Thumb code and Data respectively. Each special symbol entry contains an address and its symbol type as shown in Figure 4.

ARM function	Thumb function
… 0x104: blx r0 //r0 = 0x201 0x108: mov r5, r0 ...	0x200 <thumb_function> 0x200: add r4, r1, r2 0x202: mov r5, r0 … 0x400: bx lr

Figure 3. Example of BLX and Bx

Symbol address	Symbol type
0x200	$a
0x300	$d
0x308	$t

Figure 4. Example of special symbols in a symbol table

The corresponding code is shown in Figure 5. ARM code starts from 0x200 until 0x300 which is indicated by the second entry in Figure 4. There are eight bytes of data starting from 0x300. Lastly, Thumb instructions start at 0x308. Note that Thumb instruction addresses increments by two because each instruction is two-byte long.

2.3 LLBT Overview

At a high level, LLBT consists of three phases: binary parsing (frontend), IR processing, and code generation (backend). LLBT frontend leverages GNU binary utilities such as objdump and readelf to disassemble the input shared object and convert it into LLBT internal representation (IR) [6]. LLBT analyzes the IRs in several phases to extract information such as dynamic symbols and control flow. Moreover, LLBT needs to recognize and convert the

```
0x200: push {r4, lr} //start of ARM code
0x204: add r4, r0, r1
…
0x2fc: pop {r4, lr}
0x300: .word 0xffff9984 //pc-rel data
0x304: .word 0xffffab68 //pc-rel data
0x308: push {lr} //start of Thumb code
0x30a: add r3, 1
0x30c: mov r4, 0
...
```

Figure 5. Example of a mixture of instruction modes and data

platform-dependent features of ARM such as shifter operands, PC-relative data and 16-bit Thumb instructions into platform-neutral implementations [1]. As output, LLBT generates LLVM assembly code that emulates the input shared object. As shown in Table 1, LLBT creates a variable for every ARM register and creates a data array for ARM stack. Finally, we utilize LLVM-MIPS backend to generate a MIPS shared object which will be packaged with the APK [7].

During IR processing, LLBT needs to retain exported function names and global variables from the input shared object. For instance, an exported function from the input shared object is translated to an output function with the same name. This ensures that the Java program from the APK can reach the expected functions at runtime. Unfortunately, as previously mentioned, symbols on internal functions are often removed from the symbol table. In other words, LLBT is unaware of where each internal function starts and ends. The original implementation of LLBT consolidates internal functions to a single output function named unexported_text_section. Although correctness can be achieved, this implementation is prone to long translation time and high memory usage.

Table 1. Register Mapping

ARM registers	LLVM variables
r0-r12	ARM_{r0-r12}
SP	ARM_SP
LR	ARM_LR
PC	ARM_PC

2.4 Shortcomings of LLBT

Consolidating internal functions results in a larger output function. The effect of the increased function sizes is insignificant for small shared objects, but becomes a major bottleneck as the input shared object gets larger. Figure 6 shows the profiling results on 14 APKs. On average, over 90% of the code from a shared object is from internal functions. For an APK with 1000 functions, if we combine all of its internal functions into one large function, we could potentially have a "function" that consists of 900 original functions.

In practice, the size of unexported_text_section indeed increases dramatically with the size of the input shared object. The most time-consuming and memory-intensive phase of the binary translation process is LLVM backend compilation. Both LLVM optimizer and instruction selector involve super-linear algorithms. Therefore, under the original implementation,

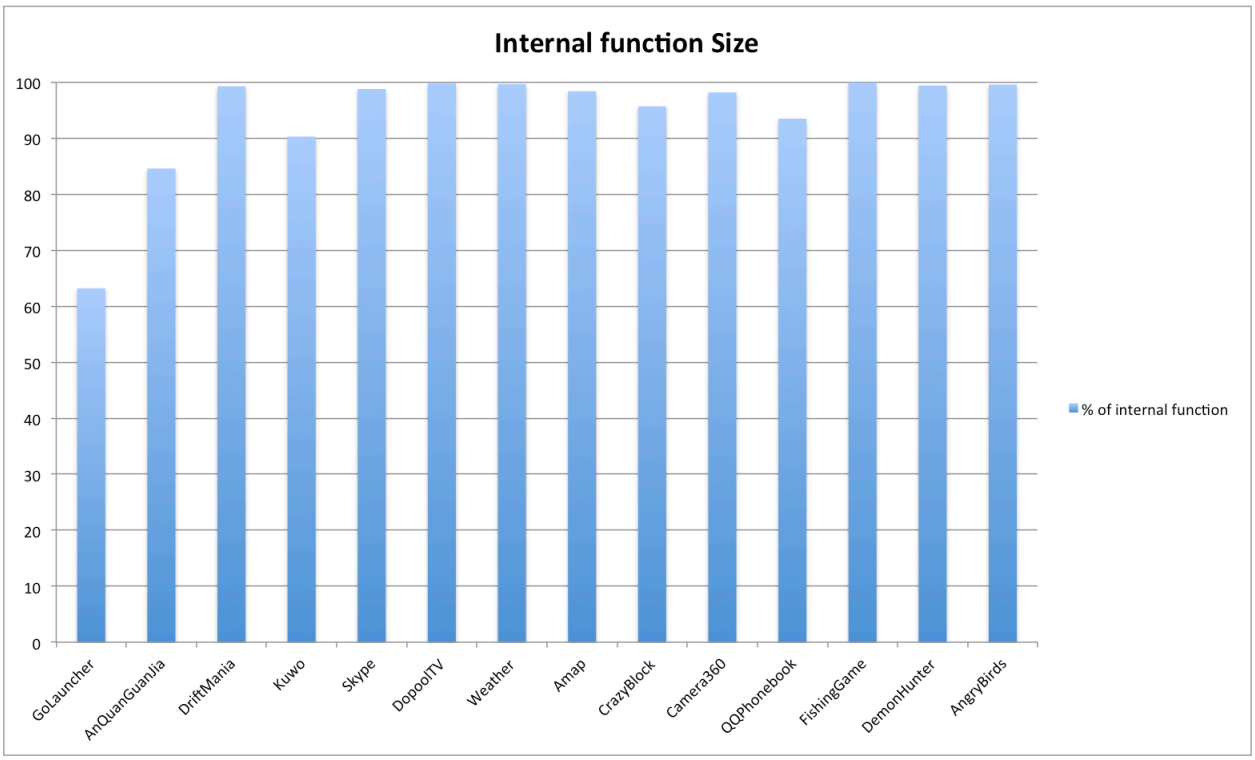

Figure 6. Percentage of APK functions that are internal functions

translation time does not scale well with input size. Table 2 shows the translation time and memory usage of four APKs. Large shared objects took over 20 minutes on an Intel i7 processor with 8 GB of RAM. Some APKs took even longer and eventually ran out memory. To make matters worse, since the processor on mobile devices is less powerful than that of a server, a 3-4X further slowdown is expected. Moreover, RAM is limited to 512 MB to 1GB. Any APKs that require more than 1GB of RAM to translate will cause thrashing between the flash memory and DRAM. Therefore, a user might need to wait for an hour or longer to translate an APK if it can be translated at all.

Table 2. Translation time and memory usage of 4 APKs

APK	Translation Time	Memory Usage
AngryBirds	25 min	7.2 GB
Camera360	20 min	3.2 GB
FishingGame	30+ min	crashed
Weather	40+ min	crashed

CHAPTER 3

INPUT DIVISION

In order to eliminate the bottleneck of compiling the large unexported_text_section, we need to identify internal functions and translate them into independent output functions. The goal of input division can be summarized as follows:

1. *Identify as many internal function entry points as possible by static analysis of the input shared object.*
2. *Create an output function for each internal function detected in step 1.*
3. *Ensure the output functions are still callable by all their original callers*
4. *Ensure correct control flow within the output functions.*
5. *Ensure that an output function can return to its caller.*
6. *Create a mechanism to guarantee 2, 3 and 4 in the case of an incorrect input division.*
7. *Ensure that the output behavior does not deviate from the original semantics.*

We will show later that there are many hazards that prevent Input Division from always correctly identifying all internal functions. Therefore, it is very important to have a mechanism (Goal 6) that allows the translated code to function correctly even if Input Division misses an internal function or incorrectly partitions an original internal function into multiple functions.

3.1 Function Entry Discovery

Since internal function symbols are usually removed from the static symbol table, static analysis of the instructions in the input shared object is the only reliable way to extract internal function entry points. Input Division traverses the disassembled instructions of the input shared object and searches for function call instructions. Whenever it reaches a BL instruction, it marks the target address as a function entry. For example, after traversing the code in Figure 7, Input Division returns three function entry addresses: 0x400, 0x600 and 0x800.

```
0x200 <some_function>
…
0x220: bl 0x400
…
0x334: bl 0x600
…
0x35c: bl 0x800
...
```

Figure 7. Entry extraction example

The major drawback of this method is that it cannot handle indirect function calls such as the one in Figure 8. At translation time, we cannot determine the value of r3. Therefore, we cannot always extract the target function's address from indirect function calls. Since our function entry point discovery method cannot provide 100% coverage, we cannot achieve a one-to-one mapping between an input function from the ARM shared object and an output function. It is possible for an output function to contain multiple input functions as illustrated in Figure 9. Moreover, Input Division could incorrectly subdivide an input function. In the example from Figure 10, Input Division breaks internal_function_1 into two functions because function entry analysis returns a false-positive entry at 0x304. We will discuss the reasons behind false-positive detections in section 3.4 and describe our fail-safe mechanism in section 3.5.

```
mov lr pc
mov pc r3
```

Figure 8. Indirect function call example

Input: ARM shared object	Output: LLVM assembly code

0x200 <internal_function_1>
…
0x400 <internal_function_2>
…
0x800 <end of internal_function_2>
	output_function_100 (...) {
	//implementation for ARM code from
	//0x200 to 0x800
	}

Figure 9. Missed function entry

Input: ARM shared object	Output: LLVM assembly code

0x200 <internal_function_1>
…
0x400 <end of internal_function_1>
	output_function_100 (...) {
	//implementation for ARM code from
	//0x200 to 0x300
	}
	output_function_101 (...) {
	//implementation for ARM code from
	//0x304 to 0x400
	}

Figure 10. False-positive entry detection

3.2 Entry Point Information Consolidation

After collecting function entry points from the dynamic symbol table and function call analysis, we need to consolidate the information because the two sets of function entry points usually overlap with each other. After consolidation, we have a list of function entry addresses, but we are still missing function sizes. We calculate the function sizes by sorting the entry addresses and taking the difference between adjacent addresses. The size of the last function is calculated by the difference between its entry address and the end of text section. Since Input Division cannot guarantee full coverage and accuracy, the list of functions and sizes is merely an estimate.

3.3 Control Flow Handling

Control flow handling is a crucial component in LLBT because it directly affects the runtime behavior of the translated shared object. First, we will introduce a base control flow mechanism that handles the original control flow in the shared objects. Following that, we will discuss the modifications and new elements needed by Input Division.

3.3.1 Original Control Flow

There are two categories of instructions that change control flow: direct and indirect branch. The classification is shown in Figure 11. The target address of a direct branch is known at translation time because it is encoded in the instruction's literal offset field. On the other hand, the target addresses of indirect branches are stored in registers and their values cannot always be determined by static analysis.

To translate a direct branch to an ARM address, LLBT needs to find the corresponding location in the output LLVM assembly code. To facilitate branches, LLBT generates a label for every ARM instruction. As shown in Figure 12, the format of LLVM labels is "L_#" where "#" is a unique number for every ARM address.

In the case of a direct branch, LLBT looks up the LLVM label for the target ARM address and generates a branch to the label. In the example from Figure 13, the call to internal_function_B is translated to a branch to L_2000, which is the LLVM label of the entry (0x600) of internal_function_B.

Direct branch	Indirect branch

b 0xADDR
bl 0xADDR
mov pc, 0xADDR
	bx rx
	blx rx
	mov pc, rx
	ldr pc, [rx]
	ldm rx, {pc}
	add pc, rx, #IMM
	add pc, rx, ry
	
	*rx and ry can be any user-mode register

Figure 11. Classification of branches

Input: ARM shared object	Output: LLVM assembly

0x200 <some_function>
0x200: push {lr, r4, r5}
0x204: add r4, r0, r1
0x208: sub sp, sp, 12
…
0x220: pop {lr, r4, r5} //end of function
	some_function (...) {
	L_1000:
	  //LLVM implementation for
	  //0x200: push {lr, r4, r5}
	L_1001:
	  //LLVM implementation for
	  //0x204: add r4, r0, r1
	L_1002:
	  //LLVM implementation for
	  //0x208: sub sp, sp, 12
	…
	L_1008:
	  //LLVM implementation for
	  //0x220: pop {lr, r4, r5}
	}

Figure 12. Example of LLVM labels

To handle indirect branches, the lookup process needs to be delayed to runtime. Therefore, we need to create an extra data structure, i.e., Address Mapping Table (AMT), that stores the mapping between ARM addresses and their corresponding LLVM labels. As shown in Figure 14, an AMT is a switch table with a case for each possible branch target.

Input: ARM assembly	Output: LLVM assembly

0x400 <internal_function_A>
...
0x500: bl 0x600 <internal_function_B>
…
0x600 <internal_function_B>
...
	unexported_text_section(...) {
	…
	L_600: // entry of internal_function_A
	...
	L_1000: //ARM instruction: bl 0x600
	  //set up LR
	  branch label L_2000
	…
	L_2000: //internal_function_B entry
	…
	}

Figure 13. Example of direct branch

Address Mapping Table

switch target_address {
  ARM_address1: LLVM_label_1
  ARM_address2: LLVM_label_2
  ARM_address3: LLVM_label_3
  ...
}

Figure 14. Address Mapping Table

An indirect branch is achieved by first saving the target ARM address in ARM_PC then branching to AMT instead of the actual target. At runtime, AMT uses the value in ARM_PC to select the target LLVM label. In the example shown in Figure 15, before the branch to AMT, 0x600 is stored in ARM_PC. AMT branches to L_2000 which marks the beginning of internal_function_B.

Similarly, function returns are achieved by updating ARM_PC with ARM_LR and branching to AMT. In Figure 16, when internal_function_B returns, the LR holds 0x504 which is

Input: ARM shared object	Output: LLVM assembly

0x400 <internal_function_A>
…
0x4fc: mov lr, pc
0x500: mov pc, r3 //r3 = 0x600
…
0x600 <internal_function_B>
	unexported_text_section(...) {
	…
	L_600: // entry of internal_function_A
	...
	L_1000: //ARM instruction: mov pc, r3
	  ARM_PC = ARM_r3
	  branch label address_mapping_table
	…
	L_2000: //internal_function_B entry
	…
	address_mapping_table:
	  switch ARM_PC {
	  0x400: L_600
	  0x600: L_2000
	  ...
	  }
	}

Figure 15. Example of indirect branch

the address of the instruction immediately after the call to internal_function_B. Since ARM address 0x504 is mapped to LLVM label L_1001, AMT will branch to the LLVM label of the return address.

3.3.2 Control Flow Modifications

In the previous implementation, all internal function calls are implemented by branches. This relies on the assumption that the caller and callee are in the same output function. Since Input Division breaks the single output function (unexported_text_section) into multiple internal functions, the assumption no longer holds and we run in the error shown in Figure 17.

Input: ARM shared object	Output: LLVM assembly

0x400 <internal_function_A>
…
0x4fc: mov lr, pc
0x500: mov pc, r3 //r3 = 0x600
…
0x600 <internal_function_B>
…
0x700: bx lr
	unexported_text_section(...) {
	…
	L_600: // entry of internal_function_A
	...
	L_1000: //ARM instruction: mov pc, r3
	ARM_PC = ARM_r3
	branch label address_mapping_table
	L_1001: // ARM instruction 0x504
	...
	L_2000: //internal_function_B entry
	…
	L_3000: //0x700: bx lr
	ARM_PC = ARM_LR
	branch label address_mapping_table
	...
	address_mapping_table:
	switch ARM_PC {
	0x400: L_600
	0x504: L_1001
	0x600: L_2000
	...
	}
	}

Figure 16. Example of function return

Label L_2000 from internal_function_A is invalid because it is illegal to branch to a label in another function [8]. Therefore, direct branches to other functions need to be achieved by function calls. As shown in Figure 18, the branch to label L_2000 is substituted by a call to internal_function_B. Input division causes a similar issue with indirect branches. In Figure 19, Label L_2000 in internal_function_A's AMT is illegal because the label is defined in another function. To resolve this problem, we created Function Table (FT) which is essentially an address mapping table with global visibility.

Input: ARM shared object	Output: LLVM assembly

0x400 <internal_function_A>
…
0x500: bl 0x600 <internal_function_B>
…
0x600 <internal_function_B>
...
	define void @internal_function_A(...) {
	...
	L_1000: //ARM instruction: bl 0x600
	branch label **L_2000**
	…
	}
	
	define void @internal_function_B(...) {
	L_2000: //internal_function_B entry
	…
	}

Figure 17. An error caused by Input Division

Input: ARM assembly	Output: LLVM assembly

0x400 <internal_function_A>
…
0x500: bl 0x600 <internal_function_B>
…
0x600 <internal_function_B>
...
	define void @internal_function_A(...) {
	...
	L_1000: //ARM instruction: bl 0x600
	call internal_function_B(...)
	…
	}

Figure 18. Direct branch modification

Input: ARM assembly	Output: LLVM assembly

0x400 <internal_function_A>
…
0x4fc: mov lr, pc
0x500: mov pc, r3 //r3 = 0x600
…
0x600 <internal_function_B>
...
	define void @internal_function_A(...) {
	L_600: // entry of internal_function_A
	...
	L_1000: //ARM instruction: mov pc, r3
	ARM_PC = ARM_r3
	branch label address_mapping_table
	…
	address_mapping_table**:**
	switch ARM_PC {
	0x400: L_600
	0x600: L_2000
	//other function entries
	}
	}
	define void @internal_function_B(...) {
	L_2000: //internal_function_B entry
	…
	}

Figure 19. Indirect branch error caused by Input Division

3.3.3 Function Table

Each output function has its own AMT which we will later refer to as the local address mapping table. Each local AMT only contains addresses within its corresponding function. Function Table is a global data structure that establishes the connection among the local AMTs. It contains an entry for every function entry address in the input shared object. Since the purpose of FT is purely for function calls, it does not contain any return addresses.

At runtime, when a target address is not found in the local AMT, the FT is queried. In Figure 20, when function_A calls function_B via an indirect branch, it first checks its local AMT (step 1). In this case, it fails because the target address is outside of the range of function_A (step 2). The address lookup process continues in function_table which contains all possible function

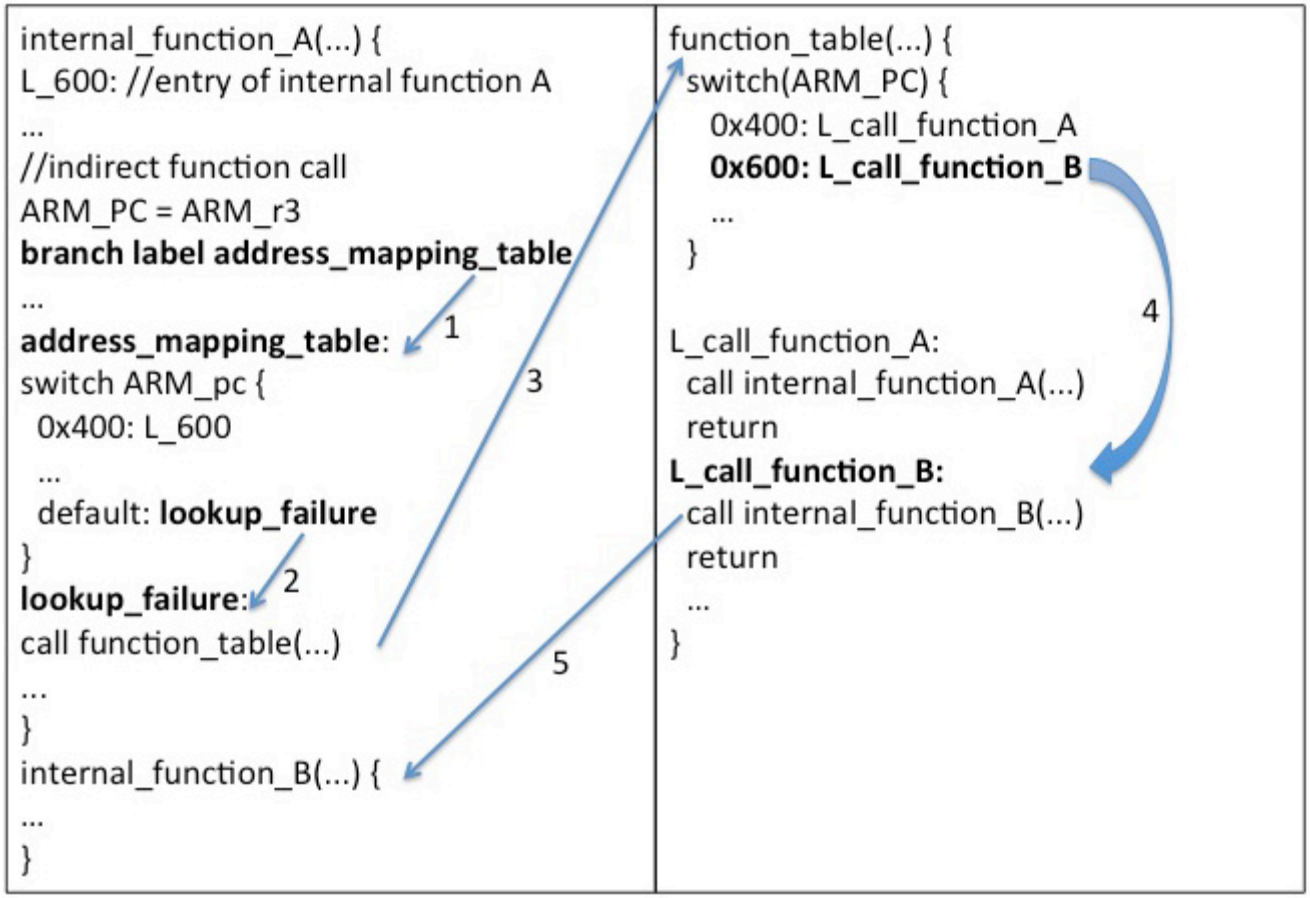

Figure 20. Function table example

entry addresses (step 4). Function_table uses ARM_PC to determine which function to call (step 4) and finally generates a call to internal_function_B (step 5).

3.3.4 Function Return

In section 2.2.3, we introduced three types of return instructions. LLBT scans the input binary for the instructions in Figure 2. When LLBT finds a match, it updates ARM_PC and performs a return operation in the output LLVM code to the caller. See the example in Figure 21. Although this implementation is very intuitive, it is not robust in the case of false-positive or false-negative function entry detections by Input Division. Therefore, we choose not to use the

```
function_2 (...) {
//0x400: add r4, r0, r1
…
//0x480: mov pc lr
ARM_PC = ARM_LR
return
...
}
```

Figure 21. Function return example

simple implementation in Figure 21. Rather, we use the implementation described in the following section to address these limitations.

3.4 Fail-Safe Mechanism

The function call/return mechanism from section 3.3 relies on the following assumptions:

1. No-false negative detections: all internal function entries can be detected.
2. No false-positive detections: all function entries detected are valid function entries.

As mentioned in section 3.1, function entry analysis cannot achieve 100% coverage. Therefore, the first assumption does not hold. Moreover, due to the uncertainty in determining instruction mode (ARM vs Thumb), which will be discussed in detail in section 3.5, the second assumption is also invalid. Therefore, we need a more tolerant control flow mechanism that works on incorrectly divided functions.

3.4.1 Function Call

In general, we cannot assume program execution always starts from the beginning of an output function because the output function could potentially contain multiple input functions. At the entry of each output function, we artificially introduce a branch to AMT which will direct control flow to the expected target address. In Figure 22, function_B and function_C are

function_A(...) {
…
// bl 0x400 <function_B>
ARM_PC = 0x400
call function_BC(...)
...
//bl 0x600 <function_C>
ARM_PC = 0x600
call function_BC(...)
…
}
	function_BC(...) {
	branch label address_mapping_table
	L_800: //0x400 <function_B>
	…
	L_1200: //0x600 <function_C>
	…
	address_mapping_table:
	switch ARM_PC {
	0x400: L_800
	0x600: L_1200
	}
	...
	}

Figure 22. Function call modification

grouped into a single output function, i.e., function_BC, because entry analysis failed to detect function_C. Since both calls are based on explicit target addresses, LLBT can determine that their target addresses are within the body of function_BC. Therefore, function calls to B and C on the left become identical. The only distinguishing factor is the value in ARM_PC. The AMT in function_BC uses ARM_PC to determine whether function_A or function_B is called.

3.4.2 Function Return

Before we present the adopted solution, we would like to first discuss the flaws of the simple solution where return instructions from the input ARM binary are translated into LLVM return instructions. This will lead to the error in Figure 23 when there are multiple input functions per output function. The problem is that the number of returns does not match the number calls, resulting in incorrect program execution. To enforce the balance between function calls and returns, we need to conform to the following rules.

1. *If an ARM function call is translated to a LLVM function call, the corresponding ARM return instruction should be translated to a LLVM return statement.*
2. *If an ARM function call is translated to a LLVM branch, the corresponding ARM return instruction should be translated to a LLVM branch.*

ARM	LLVM

0x200 <function_A>
…
0x300: bl 0x400 <function_B>
…
0x400 <function_B>
…
0x500: bl 0x600 <function_C>
…
0x600 <function_C>
…
0x800: bx lr
	function_A (...) {
	…
	//bl 0x400 <function_B>
	ARM_PC = 0x400
	call function_BC(...)
	…
	}
	
	function_BC(...) {
	//entry of function_B
	…
	//bl 0x600 <function_C>
	branch label L_1600
	…
	//entry of function_C
	L_1600:
	…
	
	return
	}

Expected control flow: A--call-->B--call-->C **C--return-->B--return-->A**	Actual control flow: A--call-->B--call-->C **C--return-->A**

Figure 23. A problem with function return

In Figure 23, function_B calls function_C via a LLVM branch. However, when function_C returns, a LLVM return statement is executed. Therefore, we cannot blindly return from a function without considering how the function is called. If the caller and callee reside in the same output function, function call and return are achieved by LLVM branches. In this case, the callee's return address is in the local AMT. Therefore, the decision whether to execute a LLVM return should be made after checking the local AMT. If AMT contains the return address, we simply branch to the LLVM label of the return address. If not, we need to return via a LLVM

return instruction. Note that there are two cases when the local AMT fails to look up the address. If the instruction prior to the branch to AMT was a function call, we need to continue the lookup process in Function Table. If it was a return instruction, AMT lookup must have failed because the return address is in another output function. In this case, we should execute a LLVM return. To distinguish these cases, we created a return flag. For every return instruction, we set the return flag before branching to the local AMT. In Figure 24, when function_C returns, the return address(0x504) is in the same function, so it will be found in the local AMT. Before branching to the return address(L1000), we need to reset return flag so that it will not affect future address lookup. When function_B returns, the return address in a different output function(function_A), we will reach lookup_failure because the address is outside of function_BC. Since return flag has been set, function_B will return to its caller.

When an input function is incorrectly divided into two output functions, we will encounter another error with function returns. In Figure 25, Input Division divides branch_test at 0x400 due to a false-positive entry detection. This forces LLBT to translate the branch at 0x300 to a function call to branch_test_2(...). When we reach the return instruction at 0x600, the intended behavior is to return to the caller of branch_test. In the implementation in Figure 25, however, we return to branch_test_1 because it is the immediate caller of branch_test_2. This is another instance of the problem where the number of function returns does not match the number of function calls. Since we introduced an extra function call for the branch at 0x300, we need to generate an extra function return to compensate. We can leverage the fact that the value in ARM_LR always stores the correct target address. The assumption is safe because LLBT only updates ARM_LR when the original ARM instruction intends to update Link Register. The solution is to insert an ARM_PC update and a branch to local AMT immediately after each

ARM	LLVM

0x200 <function_A>
…
0x300: bl 0x400 <function_B>
…
0x400 <function_B>
…
0x500: bl 0x600 <function_C>
…
0x600 <function_C>
…
0x800: bx lr
	function_A (...) {
	…
	//bl 0x400 <function_B>
	ARM_LR = 0x404
	ARM_PC = 0x400
	call function_B(...)
	…
	}
	
	function_BC(...) {
	//entry of function_B
	…
	//bl 0x600 <function_C>
	ARM_LR = 0x504
	branch label L_1600
	L_1000: //ARM address 0x504
	...
	//entry of function_C
	L_1600:
	…
	//bx lr
	ARM_PC = ARM_LR
	return_flag = 1
	branch label address_mapping_table
	…
	address_mapping_table:
	switch ARM_PC {
	…
	0x504: **return_flag = 0**
	branch label L_1000
	…
	default: lookup_failure
	}
	lookup_failure:
	if (**return_flag**)
	return
	else
	call function_table(...)
	}

Figure 24. Return flag example

function call statement. This way, when the function call returns, the execution will always look up AMT. In Figure 26, we branch to AMT immediately after returning from branch_test_2. Since the return address is in the caller of branch_test, the AMT lookup will fail and we will execute a LLVM return and return to the caller. This also balances out the extra function call at 0x300.

Note that this process is also triggered after "regular" function returns where program execution is supposed to continue at the instruction after the function call. In Figure 27, when branch_test_1 returns to the caller of branch_test, the intended execution is to continue at the next instruction at 0x114. Since ARM_LR holds 0x114 at this moment, the local AMT will branch to L_1001 which is the LLVM label for ARM address 0x114.

ARM	LLVM

0x200 <branch_test>
...
0x300: b TARGET
0x304: mov r0, r4
…
0x400: //false-positive function entry
...
0x500: add r4, r0, r1 <TARGET>
…
0x600: bx lr //return
	branch_test_1(...) {
	…
	//0x300: b TARGET
	ARM_PC = 0x500
	call branch_test_2(...)
	//0x304: mov r0, r4
	ARM_r0 = ARM_r4
	…
	}
	branch_test_2(...) {
	…
	//0x600: bx lr
	ARM_PC = ARM_LR
	branch label address_mapping_table
	…
	}

Figure 25. Function return error

ARM	LLVM

0x200 <branch_test>
...
0x300: b TARGET
0x304: mov r0, r4
…
0x400: //false-positive function entry
...
0x500: add r4, r0, r1 <TARGET>
…
0x600: bx lr //return
	branch_test_1(...) {
	…
	//0x300: b TARGET
	ARM_PC = 0x500
	call branch_test_2(...)
	ARM_PC = ARM_LR
	branch label address_mapping_table
	//0x304: mov r0, r4
	ARM_r0 = ARM_r4
	…
	}
	branch_test_2(...) {
	…
	//0x600: bx lr
	ARM_PC = ARM_LR
	branch label address_mapping_table
	…
	}

Figure 26. Fix to function return error

ARM	LLVM

0x100 <branch_test_caller>
…
0x110: bl 0x200 <branch_test>
0x114: mov r5, r0
...
	branch_test_caller (...) {
	…
	L_1000: //ARM 0x110: bl 0x200
	ARM_PC = 0x200
	ARM_LR = 0x114
	call branch_test_1(...)
	ARM_PC = ARM_LR
	branch label address_mapping_table
	L_1001: //ARM 0x114: mov r5, r0
	ARM_r5 = ARM_r0
	…
	address_mapping_table:
	switch ARM_PC {
	...
	0x114: L_1001
	…
	}
	}

Figure 27. A regular function return routine

3.4.3 Fall-Through Functions

In the LLVM assembly code generated by LLBT, there exist "fall-through" functions which do not return at the end. The intended execution is to continue to the next function. There are two possible reasons for this. First, some highly optimized assembly code has fall-through functions. In the example from Figure 28, function_A and function_B have a large overlap, i.e., instructions 3 to 64. Since they only differ on instruction 1 and 2, it is more space-efficient to make them share the same code region. Another reason for fall-through function is false-positive entry detections from Input Division. As we will discuss in section 3.4.1, it is possible that Input Division breaks an input function into two output functions. Therefore, there is no return instruction at the end of function 1 because the intended execution is to continue to the first instruction in function 2.

The solution to this problem is very straightforward. If the end of an output function does not have a return instruction, we artificially introduce a call to the beginning of the next function. In Figure 29, at the end of function_A, LLBT updates ARM_PC with the entry address of function_B and generates a call to function_B. Note that there will be a new return statement after call function_B(). The call-return sequence will be slightly different from the original code.

callees	caller
0x400 <funciton_A> 0x400: instruction 1 0x404: instruction 2 0x408 <function_B> 0x408: instruction 3 0x40c: instruction 4 … 0x4fc: instruction 64 0x500: bx lr //return	0x200: bl 0x400 … 0x240: bl 0x408

Figure 28. Fall-through function example

```
function_A(..){
//instruction 1
//instruction 2
store 0x408 %ARM_PC
call function_B(...)
}
```

Figure 29. Fall-through function handling

Originally, the execution starts from a caller, enters function_A, continues to function_B, and eventually returns to the caller. After the translation, the execution starts from the same caller, enters function_A, calls function_B, returns from function_B to function_A, and returns from function_A through the new return instruction.

3.5 Incorrect Division

Previously, we mentioned that input partitioning could generate false entry points. First, we will talk about the cause of incorrect division as well as the performance penalty of it. Then, we will describe how to prevent false-positive detections.

3.5.1 ARM-Thumb Ambiguity

Due to indirect branches, it is difficult to determine whether a code region is ARM or Thumb without special symbols. Conservatively, LLBT frontend generates a set of IRs for both ARM and Thumb. Input division traverses both ARM and Thumb IR sets to search for Branch and Link instructions. In general, only one of them is valid and will be used at runtime. However, since Input Division needs to scan them statically, it must assume that both are possible. When an ARM instruction is disassembled as Thumb instruction, or vice versa, it is entirely possible that a non-branch-and-link instruction is incorrectly generated into a Branch and Link.

In most cases, when an instruction is incorrectly interpreted as a BL instruction, the target address is based on garbage bit patterns and often corresponds to an address outside the .text section that contains all the valid target addresses for that branch. This helps us to screen away some false-positive entries. That is, we check if the target address is valid by comparing it to the address range of .text section. This filters out the majority of the false-positive detections, but it cannot guarantee to eliminate all of them. It is possible that the target address from an incorrectly disassembled instruction seems valid because it falls within the .text section. In this case, input partition will register the address as a valid function entry point and will subdivide a function.

3.5.2 Cost of Incorrect Division

When a function is incorrectly partitioned into multiple functions, the problem becomes similar to fall-through functions. As we described previously, the inserted function call branches will ensure program execution continues from one function to the next. Therefore, under normal circumstances, we can still achieve the expected runtime execution even when there are incorrect divisions. However, a problem arises when input partition divides a function into two parts that have frequent branches to each other.

When the thumb function on the right-hand side of Figure 30 is disassembled as ARM, it contains a bl instruction with target address in branch_test. Therefore, input division thinks 0x380 is a valid function entry and breaks branch_test into two output functions shown in Figure 31. Since PART_ONE and PART_TWO are not in the same output function, any branches between these regions have to be achieved by a function call. PART_ONE and PART_TWO will keep on calling each other until we reach the return statement at the end of branch_test. During the process, stack keeps growing. Moreover, it is likely that the program runs out of stack before

0x200 <branch_test> ... 0x2fc: sub r0, r1, r2 <PART_ONE> 0x300: b PART_TWO … 0x400: add r4, r0, r1 <PART_TWO> … 0x500: b PART_ONE … //return	//A Thumb function disassembled as ARM … bl 0x380 ...

Figure 30. Example of incorrect division

branch_test_1(...) {
//starting at 0x200
...
//0x300: b PART_TWO
ARM_PC = 0x400
call branch_test_2(...)
...
}
	branch_test_2(...) {
	//starting at 0x380
	…
	//0x500: b PART_ONE
	ARM_PC = 0x2fc
	call branch_test_1(...)
	…
	}

Figure 31. Output of incorrect division

PART_TWO returns. This violates Goal 7 of input division because the input program does not intend to recursively allocate stack frames. As a result, we decided not to allow Input Division to subdivide functions.

3.5.3 False-Positive Prevention

To prevent function subdivision, we need to improve the quality of entry extraction. The root cause of the problem is the ambiguity of instruction mode. We can leverage the information held in the dynamic symbol table. Besides providing a list of function entries, the dynamic symbol table also gives us a pool of code blocks on which we can safely conduct function entry analysis. The addresses listed in the dynamic symbol table indirectly reflect whether a function is ARM mode or Thumb mode. Since instruction addresses are half-word aligned, the LSB is

always 0. The dynamic symbol table uses the LSB to indicate the mode of a function. In Figure 32, function_2 is a Thumb function because the LSB of its address is 1. Note that the actual address of function_2 is 0x500. The function addresses and their corresponding sizes in the dynamic symbol table provide us a list of code blocks with their instruction modes. To prevent false-positive entry detections, we restrict the function entry analysis to these code regions only. This approach will extract all the internal functions that are directly called by exported functions, but it will miss ones that are only called from internal functions.

In Figure 33, internal_function_2 will be detected when we scan external_function_1. However, internal_function_3 will be missed if it is only called from an internal function. The experiments in Table 3 compare the number of discovered function entries before and after the adjustment on entry extraction. On average, the number of function entries is reduced by 9%. Therefore, the adjustment is a reasonable compromise because it ensures correct runtime behavior.

Address	Size	Name
0x400	0x100	external_function_1
0x501	0x200	external_function_2
0x700	0x250	external_function_3

Figure 32. Example of a dynamic symbol table

```
0x400 <external_function_1>
…
0x420: bl 0x400
…
0x1400 <internal_function_2>
…
0x1440: bl 0x1600
…
0x1600 <internal_function_3>
...
```

Figure 33. Example of restricted function entry extraction

Table 3. Effect of entry extraction adjustment

APK	# of functions before	# of functions after	% missed entries
Kuwo	106	92	13.3
Skype	302	288	4.6
Weather	305	236	22.6
Amap	813	759	6.6
CrazyBlock	411	372	9.5
QQPhoneBook	694	678	2.3
DemonHunter	1331	1319	0.9
AngryBirds	2255	1964	12.9

CHAPTER 4

OPTIMIZATION

Up to this point, the output of LLBT is a single LLVM assembly file. Before Input Division, the file contains a large output function (unexported_text_section) that holds more than 90% of the shared object. Input Division breaks the function into many output functions resulting in a significant speedup. A further improvement is to divide the output into multiple files. As shown in Chapter 5, dividing the output file not only improves compilation speed, but also significantly reduces memory footprint. The output file is divided at the function level. For example, the output from Figure 34 is divided into three files in Figure 35.

This approach would generate many output files if there are many small functions. As a further improvement, we establish a minimum file size. The code generation phase is a while loop that iterates over all the output functions. Instead of creating a new file per output function, we only create a new file if the previous file has already exceeded the minimum threshold. Under this implementation, the large functions will be isolated into different files and small ones are still in the same file.

output.ll

define void @function_1(...) {
//implementation for function_1
}
define void @function_2(...) {
//implementation for function_1
}
define void @function_3(...) {
//implementation for function_1
}

Figure 34. Single output file

output1.ll

//global declarations
define void @function_1(...) {
//implementation for function_1
}

output2.ll

//global declarations
define void @function_2(...) {
//implementation for function_2
}

output3.ll

//global declarations
define void @function_3(...) {
//implementation for function_3
}

Figure 35. Multiple output file

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we evaluate Input Division and Optimized Input Division in terms of translation time and memory usage. The benchmarks we used are chosen from the list of most popular APKs in Android Marketplace. We selected 14 APKs that have a wide range of shared object sizes that give us different input characteristics. We translated these 14 APKs whose shared object sizes range from 20KB to 2MB on a machine with Intel i7 CPU at 2.0GHz and 8GB of RAM. We measured the results with three versions of LLBT. The baseline version of LLBT consolidates all internal functions into a single output function. The Input Division version produces multiple output functions in a single output file. The Optimized Input Division version divides the output into multiple files.

5.1 Translation Time

Table 4 shows the translation time of 14 APKs with three versions of LLBT. Note that Weather and FishingGame crashed during translation by Baseline LLBT because the process ran out of memory. Figure 36 normalizes the results from Input Division and Baseline version to that of Optimized Input Division. The speedup on shared objects over 500KB is on average over 10X. In the best case, Input/Output Division became an enabling technique because some APKs such as Weather and FishingGame cannot be translated by Baseline LLBT. Optimized Input Division also achieved a slight speedup over Input Division. In summary, Optimized Input Division achieves an average speedup of 6.9X over Baseline version and 1.4X over Input Division.

Table 4. Translation time of 14 APKs

APK	Binary size(KB)	Baseline (seconds)	Input division (seconds)	Optimized input division(seconds)
GoLauncher	20.5	3	6	2
AnQuanGuanJia	32	12	10	7
DriftMania	108	235	32	26
Kuwo	128	3	7	5
Skype	216	18	11	9
DopoolTV	270	857	77	65
Weather	385	N/A	135	102
Amap	532	390	176	138
CrazyBlock	536	209	33	26
Camera360	580	1173	73	62
QQPhonebook	680	104	37	30
FishingGame	801	N/A	254	187
DemonHunter	1300	771	108	87
AngryBirds	2040	1495	161	124

5.2 Memory Usage

Table 5 shows the peak memory usage of the translation process and Figure 37 normalizes the results from Input Division and Baseline to that of Optimized Input Division. Optimized Input Division reduced memory usage by 20X on average over Baseline and 6X over Input Division.

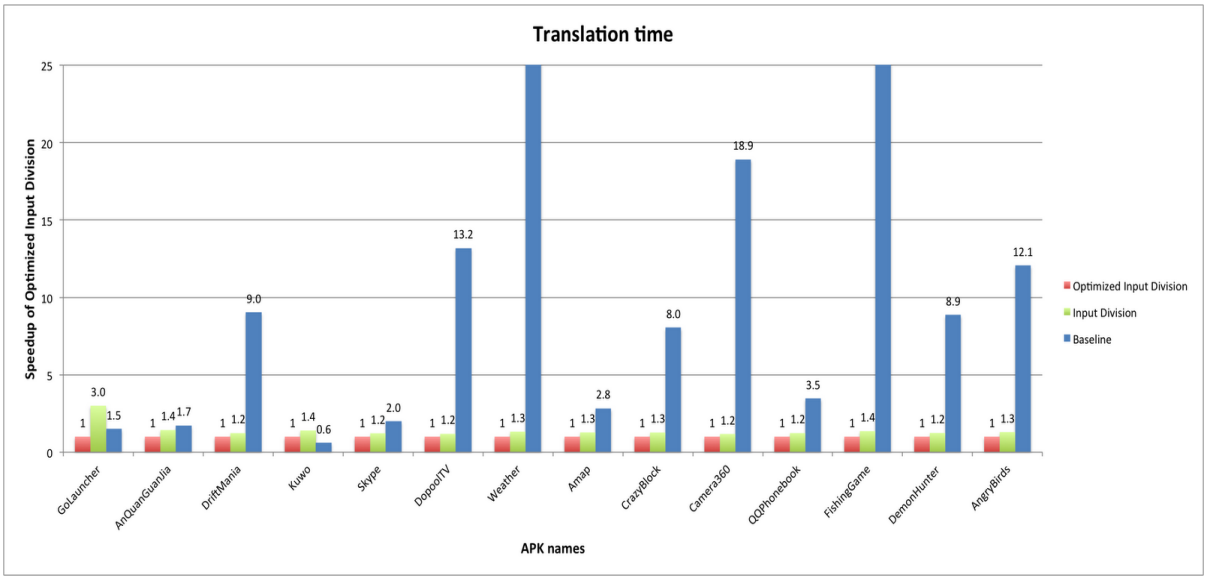

Figure 36. Speedup of Optimized Input Division

Table 5. Peak memory usage

APK	Binary size(KB)	Baseline (MB)	Input division (MB)	Optimized input division(MB)
GoLauncher	20.5	124	82	52
AnQuanGuanJia	32	254	154	88
DriftMania	108	1205	413	146
Kuwo	128	41	127	52
Skype	216	776	186	47
DopoolTV	270	2936	1301	268
Weather	385	N/A	1502	443
Amap	532	7475	2117	683
CrazyBlock	536	2232	500	56
Camera360	580	3268	1207	94
QQPhonebook	680	2228	536	80
FishingGame	801	N/A	4202	396
DemonHunter	1300	6780	1524	141
AngryBirds	2040	7263	2096	186

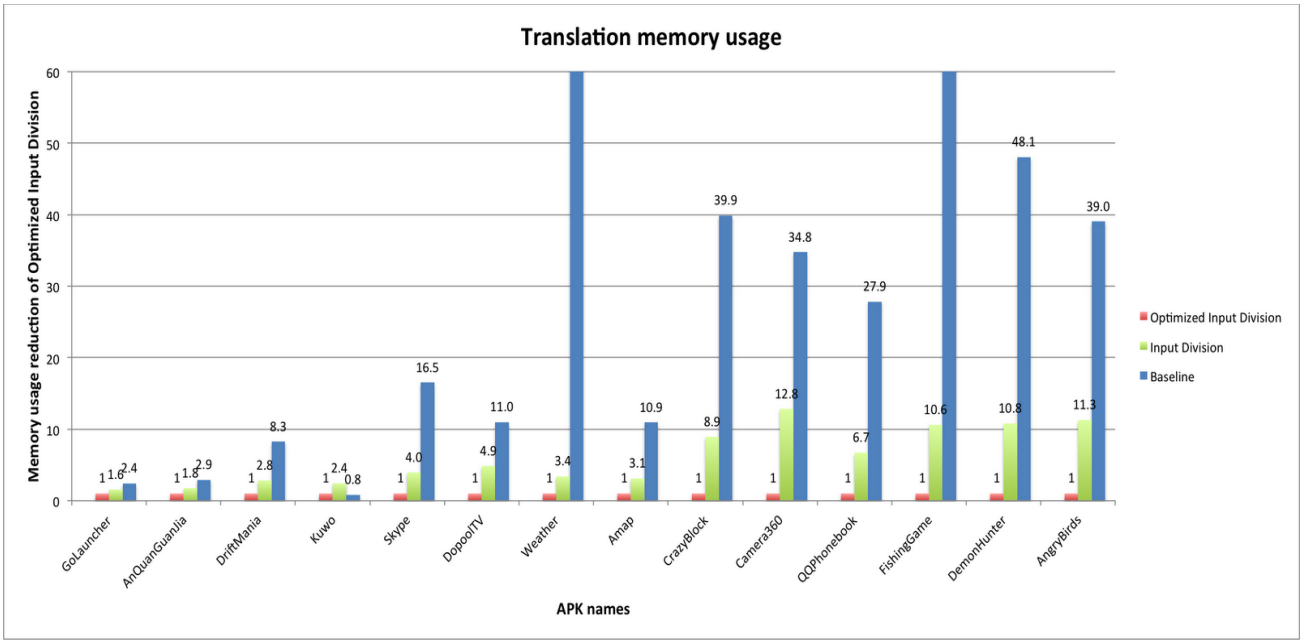

Figure 37. Memory usage reduction

CHAPTER 6

RELATED WORK

Kruegel et al. encountered a similar problem in their static disassembler for obfuscated binaries [9]. Obfuscated binaries are the ones that have been transformed to make them harder to disassemble. These transformations make it difficult to reverse-engineer the machine code instructions from a binary while preserving the original program's functionality. The goal of these transformations, a.k.a. obfuscation, is usually to protect proprietary information in software products. It can also be used to hide malicious content in a seemingly normal program.

In their paper, Kruegel et al. described techniques they employed to efficiently disassemble obfuscated binaries. The first step of their work is to identify function entry points. The function entry analysis presented in this thesis is one of the methods. It would be ideal to scan function call instructions and extract target addresses. However, the required information is not available at this step because it is the disassembler's job to translate bit patterns into function call instructions. Moreover, an obfuscator can redirect calls to a central function that transfers control flow to appropriate targets. Therefore, extracting function entries by scanning function call instructions is not a feasible solution in their application. As a result, they used a heuristic to locate function entries. They search the binary for typical byte sequences that implement function prologs. For example, opcodes that allocate stack space to save callee-saved registers are usually a good indicator of a function entry address.

The technique from this paper is useful to LLBT because it will enhance the coverage of our function entry analysis. However, it may impose stress on compilation time, especially when

we are unsure of the instruction mode of a byte sequence. Without careful handling, this may also produce false-positive detections similar to ones discussed in section 3.5.

CHAPTER 7

CONCLUSION

In general, Input Division reduces translation time and memory usage. The effect of Input Division on smaller shared objects is less noticeable because small shared objects have very few internal functions. Therefore, there is little opportunity for reduction in translation time and memory usage. Moreover, the overhead of processing multiple function entries offsets the improvement attained by Input Division. This results in less overall improvement. For large shared objects, Input Division provides up to 18.9 X improvement. In two test cases, the translation does not even work without input division. Optimized Input Division provides further improvement in both translation time and memory usage. It appears that the backend code generation and optimization tools have super-linear execution time and memory usage over the size of the input files. Although we do not have access to the implementation details, experimental results show that keeping the LLVM files small is critical in achieving fast translation and small memory usage.

In general, larger shared objects take longer to translate. However, shared object size is not the only factor. Translation time and memory usage are also dependent on control flow complexity. For example, Amap is smaller than CrazyBlock, but takes longer and more memory to translate.

In summary, Optimized Input Division achieved up to 18.9X speedup and 48X memory usage reduction over Baseline LLBT on the 14 APKs we tested. It also made translation possible for two APKs that previously required too much memory.

REFERENCES

[1] J.Y. Chen, W. Yang, J. Hung, C. Su and W.C. Hsu, "A Static Binary Translator for Efficient Migration of ARM based Applications," in *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded System*, 2008.

[2] "Dalvik Virtual Machine," Google Inc., (accessed March 2012), [Online], Available: http://code.google.com/p/dalvik/

[3] "Android Native Development Kit," (2012), Google Inc., [Online], Available: http://developer.android.com/sdk/ndk/overview.html

[4] B.Y. Shen, "Binary Translation for Native Code Inside Android Applications," (accessed march 2012), [Online], Available: http://people.cs.nctu.edu.tw/~byshen/

[5] "ARM Architecture Reference Manual," (2011), ARM, [Online], Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

[6] "GNU Binary Utilities," (2011), GNU, [Online], Available: http://www.gnu.org/software/binutils/

[7] "LLVM API Documentation", (2012), LLVM, [Online], Available: http://llvm.org/docs/doxygen/html/dir_98f17b3216e00ac06ad45315bb3cdc97.html

[8] "LLVM Assembly Language Reference Manual," (2012), LLVM, [ONLINE], Available: http://llvm.org/docs/LangRef.html

[9] C. Kruegel, W. Robertson, F. Valeur and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of USENIX Security,* 2004.