Formal Semantics Extraction from Natural Language Specifications for ARM

Viet Anh Vu

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology

December, 2018

Master's Thesis

Formal Semantics Extraction from Natural Language Specifications for ARM

1610432 Viet Anh Vu

Supervisor :	Mizuhito Ogawa
Main Examiner :	Mizuhito Ogawa
Examiners :	Kazuhiro Ogata
	Nguyen Minh Le
	Keita Yokoyama

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
[Information Science]
November, 2018

Abstract

Recently, Malware Analysis has been received much attention not only in industry but also in the academic community. Modern malware frequently applies obfuscation techniques (e.g., indirect jump, overlapping instruction) to conceal its behaviors and protect itself against antivirus software, which regularly uses lightweight detection methods like bit-based fingerprints. Since famous commercial disassemblers such as IDA Pro and Capstone are easily fooled by these obfuscations, analyzing and detecting the obfuscated malware are not straightforward. For dealing with these problems, some malware analysis approaches based on Control Flow Graph (CFG) have been proposed (e.g., VxClass at Google). To effectively reconstruct the CFG, a technique called Dynamic Symbolic Execution (DSE) (known as concolic testing) has been widely applied. It is the combination of symbolic execution and testing to automatically explore all feasible program execution paths and determine the destination of obfuscation code like indirect jumps (by using a binary emulator). In DSE, the feasibility of a path constraint is checked by testing with a satisfiable instance using a Theorem Prover. Currently, we have developed BE-PUM (Binary Emulation for PUshdown Model), a binary analyzer concentrating on malware for Intel x86 architecture. Learning from its experiences, BE-PUM can be extended to other platforms. By considering IoT Malware, ARM is our first target.

ARM is a family of computer processors, acts as the architecture behind billions of devices, especially IoT devices. The explosive growth of IoT devices leads to the rapid increase of IoT malware. With a huge number of connected devices worldwide, IoT malware can infect quickly from a device to others among the network. After the infection, their collaboration might cause dangerous large-scale attacks (e.g., BotNet). As our observation, even IoT malware is lightweight, it still contains obfuscated code. To overcome this problem, DSE should be applied to efficiently reconstruct the Control Flow Graph of IoT malware. After that, the generated CFG can be used to correctly trace behaviors of malware then proceed detection and classification tasks. However, ARM architecture consists of various series such as Cortex-M, Cortex-A, and Cortex R. Due to the huge number of instructions for each series, manual implementation of the Dynamic Symbolic Execution for ARM requires a lot of engineering efforts. As the result, a method to semi-automatically extract the semantics of ARM is essential and meaningful.

Based on our observation from the official ARM developer website, the ARM instructions have been written by natural language in English with some specific information such as mnemonic, operation specifications, and flags-update descriptions. Through our investigation, these descriptions frequently use some particular phrases, which can be extracted by using some Natural Language Processing techniques. Therefore, it is feasible to systematically extract formal semantics of ARM instructions from these documents. After that, the binary emulator and path conditions can be also generated, which will be used in the Dynamic Symbolic Executor for ARM. By semi-automatically generating from Natural Language Specifications, this procedure saves a lot of human efforts. However, the lack of pseudo-code in the description of operations is a big challenge, and the various type of flags-update description also makes some difficulties to analyze.

This thesis proposes an efficient method to systematically extract the formal semantics of ARM instructions from their natural language specifications over six Cortex series: Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, and Cortex-M33. Although ARM is based on RISC architecture and the number of instructions is rather small, a large number of variations exist under Cortex-A, Cortex-M, and Cortex-R. Thus, automatic extraction of the formal semantics of rather simple instructions reduces the human effort for tool development, such as the dynamic symbolic execution. We focus on 6 variations, M0, M0+, M3, M4, M7, and M33 of ARM Cortex-M, aiming to cover IoT malware. By preparing 205 semantics interpretation rules (and additional 23 syntax normalization rules) on phrases, we have automatically extracted the formal semantics of 692 instructions among 1039 collected natural language specifications for 6 variations, and 662 instructions have passed the automated conformance testing. With 35 manually implemented default methods, a dynamic symbolic execution tool for ARM Cortex-M variations has started to work. Because our proposed method is a generalized approach, it can be quickly extended to other platforms without a lot of modifications.

Keywords: Semantics Extraction, IoT Malware, ARM Cortex-M, Dynamic Symbolic Execution, Natural Language Processing.

Acknowledgment

First and foremost, I wish to express my sincere gratitude and respect to my supervisor, Professor Mizuhito Ogawa for his continuous supports and kindly guidance during my study at Japan Advanced Institute of Science and Technology. He has inspired me to become a scientific researcher, as well as given me invaluable knowledge of how to deal with problems and how to think critically. While I was completing this thesis, even at very late night, he still spent several hours to discuss and gave me lots of comments. I am very thankful about that.

Besides, I would like to say my special thanks to my second supervisor, Associate Professor Nao Hirokawa for his useful advice and sharp comments for my research. He also gave me many suggestions for the slides and contents of my presentation, which actually helps me a lot to improve my work.

I would like to express my appreciation to my friends, Dr. Vu Xuan Tung, Mr. Nguyen Lam Hoang Yen, Ms. Vuong Thi Hai Yen, Mr. Trac Quang Thinh, and Ms. Yoon Myet Thwe. Thank you for sharing wonderful moments, interesting ideas, and useful experiences not only in research but also in daily life. I will never forget our Kimono lab, absolutely. My thank also goes to Mr. Le Khanh Trinh, for helping me a lot since the very first day I came to JAIST.

Last but not least, my family is an indispensable part of my life. There are no proper words to express how much important they are. Their big love and encouragement motivated me to study and keep moving forward. I would like to give my heartfelt thanks to my dear father, mother, and sister. Without their support, it would be impossible for me to complete this work.

Contents

This dissertation was prepared according to the curriculum for the collaborative education program organized by Japan Advanced Institute of Science and Technology and University of Engineering and Technology, Vietnam National University.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Malware (shortened of Malicious Software) is a program intending to infiltrate the system without the acceptance of owners, them harm or disable computer systems. By the purpose of how it damages the system, it can be characterized into many types such as worms, virus, trojans, and spyware. The best way to protect the systems against the damages of malware is proactive detection. However, it is not an easy problem because modern malware usually uses many obfuscation techniques [1] in order to conceal its behaviors and protect itself against antivirus software. Therefore, the problems of how to detect and classify malware have been received a lot of attention in the community. There are many existing approaches to do that including static analysis, dynamic analysis, model checking based, and machine learning approach. In almost cases, disassembling binary files is the very first requirement. However, current famous disassemblers (e.g, IDA Pro, Capstone) are easily fooled by obfuscation techniques. To overcome this problem, some model checking based approaches [2] [3] [4] have been applied to first obtains the abstract model of the binary file, then proceed some further methods to detect malware based on the generated model. A typical model has been used is Control Flow Graph (CFG), which is a directed graph to represent the program execution process. Currently, we have developed BE-PUM (Binary Emulation for PUshdownModel), a binary analyzer focusing on x86 malware. It uses the Dynamic Symbolic Execution technique to reconstruct the CFG of malware, then enables us to trace its correct behaviors. By learning from the experiences, BE-PUM can be extended to other architectures, and the first target is ARM.

ARM is a processor family, which is the backbone of billions of devices in the world, especially the IoT devices. Recently, the number of IoT device has raised rapidly. According to a prediction of Statista [1], the number of IoT devices connected worldwide during the period from 2015 to 2025 will increase from 15.41 to 75.44 billion. This development leads to the fast growth of IoT malware. Different from Intel x86 malware, IoT malware is light-

[1]Statista (2018). Internet of Things - the number of connected devices worldwide 2015 - 2025

weight, which can be deployed in many small devices such as smart-phones, routers, and devices in smart-home ecosystems. Although the computing power of each IoT device is low, for a huge number of IoT devices, they can collaborate for making large-scale attacks (e.g., BotNet). Although almost IoT malware does not include obfuscation techniques (due to rare system update), analyzing of indirect jump will be needed to understand its control behavior. To overcome this problem, it is necessary to apply Dynamic Symbolic Execution to reconstruct its Control Flow Graph, then use the generated graphs to perform detection or classification tasks. Dynamic Symbolic Execution technique requires a binary emulator and path constraints generation over each execution. However, ARM architecture has many Cortex series; each one contains an instruction set with plenty of individual instructions. This diversity takes a huge engineering effort to manually implement the Dynamic Symbolic Executor for ARM. Therefore, a method to systematically extract the formal semantics of ARM instructions will be very useful since it saves a lot of human efforts. Our ultimate goal is proposing a method to semi-automatically extract formal semantics of ARM. This method must be generalized for being able to be extended to other architectures without big modifications.

Symbolic Execution [5] is an old, powerful, and popular method to analyze and/or verify software. It has been developed mostly for high-level programming languages, such as Java and C. Recently, the symbolic execution tools for binaries gradually increase, e.g., MiAsm [6], McVeto [7], CoDisasm [8], BE-PUM [9], and KLEE-MC [10], but most of them target on x86. Considering the evolving threat of IoT malware, extending such tools to various instruction sets, e.g., ARM, MIPS, and PowerPC becomes important. The difficulties on handling binaries lie on (1) the operational semantics is intricate to human and (2) the number of instructions is often large, e.g., > 1000 for x86. However, contrary to high-level programming languages, good news are:

- A binary program has a simple semantics framework as transitions on the environment consisting of a memory, a stack, registers, and flags.
- The instruction set has a rigid natural language specification.
- Since many debuggers and emulation environments are prepared, the ambiguity of natural language specifications can be resolved by testing.

We target on ARM, of which the specifications is available on ARM Developer Website [11]. Since ARM is a RISC-based processor family, it has rather few instructions ($\simeq 60$ - 300). However, it has three Cortex series: M for microcontrollers (e.g., IoT devices), A for rich operating systems (e.g., Android), R for real-time systems (e.g., LTE modems). Furthermore, each has many variations, e.g., 16 in Cortex-A, 9 in Cortex-M, and 5 in Cortex-R, which are still increasing.

This thesis proposes a method to systematically extract the formal semantics of instructions from their natural language specifications collected from ARM Developer Website.

For each instruction, first, we apply some natural language processing techniques to retrieve information of its arguments, its flags updates, and its actions. After that, the formal semantics is described as a Java method in an extension of BitSet class obtained by instantiating these information to a dynamic template. This template represents the semantics framework as a transition on quadruplets of the flags, the registers, the memory, and the stack.

Since the flags update is a boolean operation, a similarity analysis with model sentences often works, e.g., x86 specifications at Intel Developer Network [12]. A more challenging task is the semantics extraction of the actions; different from Intel Developer Network for x86, the specifications at ARM Developer Website does not provide the pseudocode description. We manually prepare rewriting rules as a semantic interpretation that converts a normalized syntax tree of a sentence to a Java code fragment in a bottom-up manner. At last, the generated Java method is automatically tested whether the result matches with a popular ARM emulator, e.g., $\mu Vision$ [13].

Note that, instead of intending a fully automatic extraction, we hope to reduce human effort by automatically handling rather simple but many instructions. Then, when developing a formal method tool, human can concentrate only on the most complex part. Our experiment is performed on 6 Cortex-M variations: M0, M0+, M3, M4, M7, and M33. By preparing 205 semantics interpretation rules (and additional 23 syntax normalization rules) on phrases, we have successfully extracted the formal semantics of 692 instructions among 1039 collected specifications, and 662 instructions have passed the automated conformance testing. With 35 manually implemented methods, a dynamic symbolic execution tool for ARM Cortex-M variations has started to work.

1.2 Problem Statement

ARM is a RISC based CPU and the number of instructions is relatively small. However, it has many variations call Cortex (e.g., Cortex-A, Cortex-M, Cortex-R). In this research, we focus on Cortex-M series since it was used in plenty of IoT devices. The ARM Developer website[2] provides the natural language specifications of Cortex-M series. Some of them have been written in structured forms with natural language description, some have been written totally in natural language and have been enclosed in a PDF file. In fact, extracting structured data from PDF file is a challenge because we just can gather almost the information by plain text even it is written inside a table in this PDF file. Forming these text into structured data requires a lot of efforts. Therefore, for the first experience, we focus on six series: Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, and Cortex-M33, which are written in structured form.

The table below shows some instructions in the official ARM Cortex-M7 instruction set

[2]https://developer.arm.com/

document. The full set can be found on ARM Developer Website for Cortex-M7 [3].

Mnemonic	Operands	Brief description	Flags
`AND, ANDS`	`{Rd,} Rn, Op2`	Logical AND	N,Z,C,V
`ASR, ASRS`	`Rd, Rm, <Rs|#n>`	Arithmetic Shift Right	-
`B`	`label`	Branch	-
`BFC`	`Rd, #lsb, #width`	Bit Field Clear	N,Z,C
`BFI`	`Rd, Rn, #lsb, #width`	Bit Field Insert	N,Z,C
`BIC, BICS`	`{Rd,} Rn, Op2`	Bit Clear	-
`BKPT`	`#imm8`	Breakpoint	-
`BL`	`label`	Branch with Link	-
`BLX`	`Rm`	Branch indirect with Link and Exchange	N,Z,C
`BX`	`Rm`	Branch indirect and Exchange	-
`CBNZ`	`Rn, label`	Compare and Branch if Non Zero	-
`CBZ`	`Rn, label`	Compare and Branch if Zero	-
`CLREX`	`-`	Clear Exclusive	-
`CLZ`	`Rd, Rm`	Count Leading Zeros	-

Figure 1.1: A part of ARM Cortex-M7 instruction set

where each instruction has same structures of natural language description including some information as follows:

- **Mnemonic:** The short name of instruction.
- **Brief description:** A brief information about the purpose of this instruction.
- **Syntax:** Main components of instruction syntax, including parameters and preconditions that must be checked before execution.
- **Operation:** This is the most essential part of the instructions, contains some sentences describing how the instruction is explicitly executed.
- **Flags update:** The update status of flags are described here. It uses many synonyms to explain what flags will be changed after execution such as change, update, modify, and set.

[3]https://developer.arm.com/docs/dui0646/a/the-cortex-m7-instruction-set/instruction-set-summary

The problem can be stated as follows:

Input: Given an official instruction set document of a Cortex series collected from the ARM developer website (one of Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, and Cortex-M33).

Output: Extracting the formal semantics of these instructions, then generate a binary emulator and path conditions that will be used for the Dynamic Symbolic Execution for ARM.

Example 1.2.1. Detailed information of UMAAL instruction in Cortex-M7

Mnemonic	UMAAL
Brief description	Signed multiply with accumulate long
Syntax	UMAAL{cond} RdLo, RdHi, Rn, Rm
Operation	The UMAAL instruction multiplies the two unsigned 32-bit integers in the first and second operands. Adds the unsigned 32-bit integer in RdHi to the 64-bit result of the multiplication. Adds the unsigned 32-bit integer in RdLo to the 64-bit result of the addition. Writes the top 32-bits of the result to RdHi. Writes the lower 32-bits of the result to RdLo.
Flags update	This instruction does not affect the condition code flags.

Table 1.1: The official description of UMAAL instruction in Cortex-M7

The natural language specification of an ARM instruction taken from ARM Developer Website consists of five sections: *mnemonic*, *brief description*, *syntax*, *operation*, and *flags update*. From these information, we aim to extract this formal semantics

$$\frac{\begin{array}{c} R_{pc} = k; instr(k) = umaal\ rdlo\ rdhi\ rn\ rm; R_{rdlo} = lo; R_{rdhi} = hi; \\ R_{rn} = n; R_{rm} = m; a = m * n + lo + hi; hi' = a \gg 32; lo' = (a \ll 32) \gg 32; \end{array}}{\langle F, R, M, S\rangle \to \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_{rdlo} \leftarrow lo'; R_{rdhi} \leftarrow hi'], M, S\rangle}\ [\text{UMAAL}]$$

by using some sentence-level NLP (natural language processing) techniques: (1) the syntax analysis, (2) the similarity analysis, and (3) the semantic interpretation. The instruction name and arguments are directly extracted from the *syntax* section by (1), the flag update information is extracted from the *flags update* section by (1) and (2), and the actions are extracted from the *operation* section by (1), (2), and (3), ignoring the *mnemonic* and the *brief description* sections.

Our expected output for this example is the Java executable code of UMAAL for Cortex-M7. This code contains the semantics transition of ARM instructions including path condition updates.

```
public void UMAAL(Character l, Character h, Character n, Character m,
   Character suffix, Character cond) {
   env.arithmeticMode = ArithmeticMode.BINARY;
   if (cond == null || env.checkCond(cond)) {
      char[] flags = new char[]{};
      BitVec result = null;
      result = mul(val(n),val(m));
      result = add(result,val(h));
      result = add(result,val(l));
      write(h,shift(result,Mode.RIGHT,32));
      write(l,shift(shift(result,Mode.LEFT,32),Mode.RIGHT,32));
      if (suffix != null && suffix == 's') {
         if (result != null) {
            env.updateFlags(flags, result);
         }
      }
   }
}
```

Figure 1.2: The generated Java code of UMAAL instruction in Cortex-M7

Solution Overview: Figure 1.3 shows the flow of our method, in which two are manually prepared:

(A) 35 initial functions for the binary emulator, which are used as basic methods in the Java dynamic template. Note that, these functions both update the environment and path condition through each transition.

(B) 228 rewrite rules consisting of 23 syntax normalization rules and 205 semantic interpretations rules where the left-hand sides of rules are collected automatically and the right-hand sides are prepared manually.

To have an optimal performance of the automatic extraction with respect to the manual preparation, we first select some ARM instructions, of which their descriptions consist of frequently appearing phrases. Then, fewer rewrite rules cover more instructions. Currently, the total 228 rewrite rules cover 662 instructions among 1039 collected specifications. As the result, by instantiating extracted information to the Java dynamic template, a method that formally describes the semantics of UMAAL is obtained. The dotted boxes indicate the insertions of the extracted information to the template.

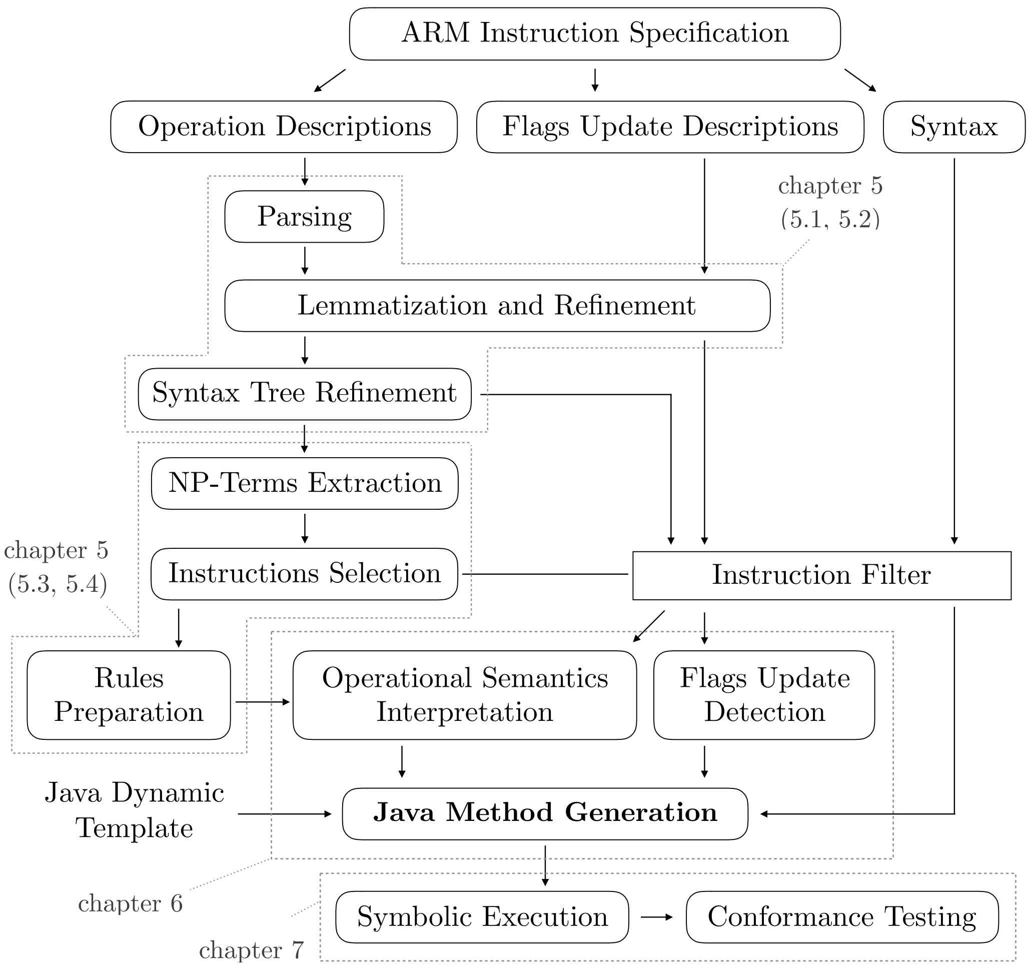


Figure 1.3: The Flow of Semantics Extraction

Difficulties

There are two major difficulties compared to x86 instruction specification. First, ARM instruction specifications lack of pseudo-code descriptions. Pseudo-code is an informal description of how the program is executed. It enables human to understand the semantics of program without knowledge of any explicit programming languages. In code generation, pseudo-code plays an important roles in describing the execution process of programs. In the work of Yen et al. [12], the pseudo-code in Intel x86 instructions are quite structured. For ARM instructions, instead of pseudo-code, the operations description is written totally by natural language in English. It makes more challenges to analyze and extract information because of the ambiguity and complex structure of natural language. In addition, the operation's description in ARM consists of many separated sentences, in which some sentences describe an operation. Moreover, it may use the result from the previous operation. Hence, to combine all sentences and transform it to a formal unified operation is also a challenge.

In addition, flags changes are also described by natural language only. The table below shows some kinds of expressions for flags-change description.

Description	**Implication**
This instruction does not change the flags	Does not update flags
This instruction does not affect the condition code flags	
The V flag is left unmodified.	
This instruction updates the N, Z, C and V flags according to the result.	Update specific flags based on description
Updates the N and Z flags according to the result. Does not affect the C and V flags.	

Table 1.2: Typical flags modification description of ARM instructions

As for Intel x86 instruction description, ARM also uses many different types of synonyms to describe flags-change. It makes more challenging to correctly detect how flags are affected after execution. Naturally, many synonyms can be used in order to describe such implications. For instance, two sentences *"The V flag is left unmodified"* and *"This instruction does not affect the V flag"* have different words, but indicate the same meanings.

1.3 Related Work

There are several model checking based approaches [2, 3, 4] to analyze malware. Different from high-level programming languages, binary code is not easy to obtain its control flow graph (corresponding to the difficulty of the disassembly). Thus, before model checking, the precise model needs to be prepared. Recently, the symbolic execution tools for binaries gradually increase, e.g., MiAsm [6], McVeto [7] , CoDisasm [8], BE-PUM [9], and KLEE-MC [10], but most of them are for x86. Furthermore, KLEE-MC and MiAsm first convert the binary code to intermediate assembly languages, like LLVM. Although current IoT malware rarely uses typical obfuscation techniques [1] of PC malware, it uses indirect jumps quite often. The traditional disassemble techniques like the *linear sweep* and the *recursive disassembling* (used in IDA Pro [14], Capstone [15]) are easily cheated by them. The use of intermediate assembly languages shares the problem. The symbolic execution of CoDisasm relies on MiAsm and has difficulties on handling indirect jumps. BE-PUM and McVeto on x86 directly apply the symbolic execution on binaries (with the one-step disassembly at the specified address) to handle indirect jumps precisely. The difference is that McVeto uses only the symbolic execution and the destination candidates of indirect jumps are analyzed statically, whereas BE-PUM uses the dynamic symbolic execution and the destinations of indirect jumps are decided by the concolic testing. Thus, the targets of McVeto is mostly limited to compiled binaries. We follow the same methodology of BE-PUM, but apply it to ARM Cortex-M.

The drawback of the methodology is that the implementation becomes heavy. Thus, we hope to have an automated support to extract the formal semantics of binary instructions. For a wider coverage of BE-PUM, the semantics extraction from the x86 specifications collected from Intel Software Developer's Manual has been tried [12]. It covered 299 x86 instructions among 530 collected specifications, and the 5 semantic bugs in the manual implementation of BE-PUM are reported. It relies on the pseudo code description for extracting the information of the actions (with the aid of manually implemented about 30 functions), and the similarity analysis using a sophisticated scoring based on WordNet [16] is an essential use to extract the information of the flags update. In our case, the specification of ARM does not have such a pseudocode description and we need to use the semantics interpretation.

1.4 Contributions

The specification of ARM instructions is written in English by natural language. This study focuses on analyzing and extracting formal specification from natural language description over six cortex series. Because of the ambiguity of natural language, it is not a straightforward task. Our main contributions consist of:

- First, we proposed a generalized method to extract the semantic of operations in instructions, then generate its formal executable code (represented in Java). This method can be extended to multiple platforms, as long as the rewriting rules for each platform are defined correctly. In this method, we also introduce a strategy for automatically select potential instructions to optimize the needed human efforts for implementation. In this method, we also provide a set of rewriting rules for ARM Cortex-M series, which can be modified for extending to other architectures.
- Second, we present an approach to automatically detect the flags-changes using an unsupervised learning algorithm called Latent Dirichlet Allocation and the Cosine similarity measure.
- Finally, we combine two methods above to develop a tool to completely generate the formal semantics of ARM instructions. In our work, this formal specifications are written in Java, consisting of both path conditions update and a binary emulator. This semantics which will be further used in the Dynamic Symbolic Execution for ARM.

1.5 Thesis Structure

This thesis is organized into nine chapters. Chapter 1 is the introduction; the main content of the next eight chapters are summarized as follows:

- **Chapter 2** presents some background knowledge of Natural Language Processing and Machine Learning techniques that are applied in the scope of this thesis.
- **Chapter 3** briefly introduces IoT malware and typical approaches of malware analysis. Besides, it also presents the approach to overcome obfuscation techniques called Dynamic Symbolic Execution (concolic testing) and the current tool BE-PUM for x86/Windows.
- **Chapter 4** first briefly talks about ARM architecture, then introduces the formal semantics of ARM, including both operational semantics and Java semantics. It also shows an overview of semantics extraction process.
- **Chapter 5** mentions about some prerequisites needed to be done before the semantics extraction process. It also illustrates how the potential instructions is selected and how we prepared rewriting rules for further generation task.
- **Chapter 6** explains how the dynamic symbolic executor for ARM is systematically generated. The generation process includes three tasks: operation code generation, flags change detection and path condition generation.
- **Chapter 7** presents a conformance testing method to verify the correctness of our implementation using Symbolic Execution.
- **Chapter 8** shows the practical experiments result over six Cortex series of ARM architecture, including the instruction selection strategy, and the generated instructions. In addition, some cases that still cannot be covered are discussed. Finally, a running example of the generated DSE tool is also demonstrated.
- **Chapter 9** summarizes the main contributions of the thesis and the advantages as well as remaining drawbacks. After that, some future works are also mentioned to suggest some directions to improve and extend our proposed method to other platforms.

Chapter 2

Preliminaries

This chapter presents some existing techniques used in our method. For extracting formal semantics of ARM, Natural Language Processing techniques are used to process text data collected from ARM Developer Website and perform the instruction normalization. We also use a Machine Learning algorithm called Latent Dirichlet Allocation for detecting flags modifications.

2.1 Natural Language Processing Techniques

In computer science, Natural Language Processing (NLP) is a research field that aims to enable computers to understand and be able to communicate with human by natural language. It is one of the indispensable factors for the interaction between human and machine. Recently, the increase of computing power and the availability of big data leads to the rapid growth of NLP. Some typical advanced problems in NLP are information extraction, machine translation, text summarization, and text generation. Applying NLP to automatically extract information from text has been received a lot of attention from the community because it saves a lot of human efforts. As our observation, this technique can be used to extract formal specification of ARM instruction from its natural language description. In this work, Syntactic Parser, TF-IDF score, and Cosine similarity have been applied.

2.1.1 Sentence Syntax Parsing

Syntax Parsing (known as Parsing for short) is a process to determine the syntactic structure of a sentence based on a given formal grammar. This technique has been widely used as the pre-processing task of many natural language processing algorithms before proceeding next steps. It inputs a sentence and a grammar, then outputs a syntax tree which represents the syntax of this sentence. The grammar used in Parsing can be a context-free or domain-oriented grammar. In this study, we used a context-free grammar.

Definition 2.1.1. A context-free grammar G is a reduction system over strings that uses a set of rules to rewrite string patterns. It consists of four components $\langle N, T, R, S\rangle$ where:

- N: a set of non-terminal symbols.
- T: a set of terminal symbols.
- R: a set of rewriting rules. Each rule is denoted as $a \to b$, where a, b are strings.
- S: a set of start symbols. Each element of S is a non-terminal symbol.

A syntax tree of a sentence S now will be generated by performing rule matching by replacing the left-hand side by the right-hand side of rules until the right-hand side contains only terminal symbols.

Example 2.1.1. Consider a context-free grammar G as follows:

```
S → NP VP
PP → P NP
NP → DT N | DT N PP | 'I'
VP → V NP | VP PP
DT → 'a' | 'the'
N → 'banana' | 'table'
V → 'see'
P → 'on'
...
```

where:

Notation	Meaning	Notation	Meaning
S	Sentence	PP	Prepositional Phrase
NP	Noun Phrase	DT	Determiner
VP	Verb Phrase	P	Prepositional
V	Verb	N	Noun

Table 2.1: Syntax notations in the grammar G

In this case, the components of G are:

- N: `NP, PP, VP, DT, ...`
- T: `'a','on','see','banana', ...`
- R: `S → NP VP, PP → P NP, ...`
- S: `S, ...`

Then, a syntax tree of the sentence *"I see a banana on the table"* based on G is:

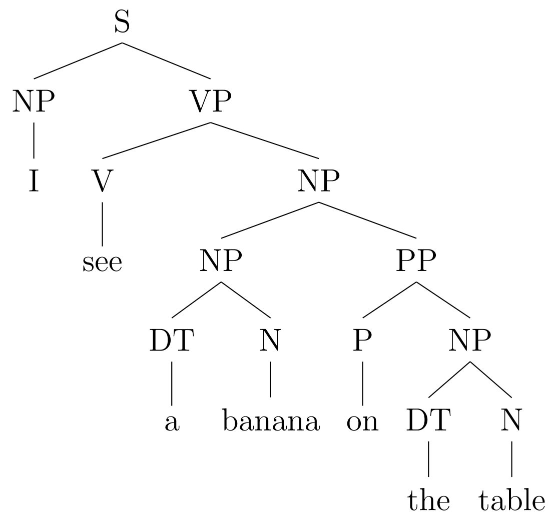


Figure 2.1: An example result of syntax parser

2.1.2 TF-IDF Score

Term Frequency - Inverse Document Frequency (TF-IDF) is a measure to estimate the importance of a word to a document over a corpus. It is usually used in text mining to refine sentences for the preprocessing task. Besides, it has been also applied to calculate the similarity between two document by using cosine distance between two sentence's TF-IDF vector. In this work, TF-IDF was used as an evaluation to remove unimportant words, acts as a pre-processing task. The reason why this task is essential is that almost unimportant words do not carry the meaning of the sentence. It can be removed to shorten the sentence and make it simpler. It is useful for the next steps of generating rewriting rules. The input of this task is a target document d and a corpus D. It outputs a real numbers vector that represents the score of each word in d.

Definition 2.1.2. Given a set of n documents:

$$D = \{d_1, d_2, d_3, \ldots, d_{n-1}, d_n\} \tag{2.1}$$

Each document i is a set of m_{i} words:

$$d_i = \{w_1^i, w_2^i, w_3^i, \ldots, w_{m_i-1}^i, w_{m_i}^i\} \tag{2.2}$$

Then:

- Term frequency of w_k^i is defined as:

$$tf(w_k^i) = \frac{f_{d_i}^k}{m_i} \tag{2.3}$$

- Inverse document frequency of w_k^i is defined as:

$$idf(w_k^i) = \log_e(\frac{n}{f_D^{w_k^i}}) \tag{2.4}$$

- TF-IDF of w_k^i is defined as:

$$tfidf(w_k^i) = tf(w_k^i) \times idf(w_k^i) \tag{2.5}$$

where:

- $f_{d_i}^k$: The number of occurrences of w_k^i in d_i.
- $f_D^{w_k^i}$: The number of document in D containing w_k^i.

The $tfidf(w_k^i)$ can represent the importance of a word w_k^i in the document d_i because:

1. If w_k^i appears many times in d_i, it is important for d_i. The bigger number of occurrences, the bigger $tf(w_k^i)$. It will led to the increment of $tfidf(w_k^i)$.
2. On a other hand, if w_k^i also appears many times in D, it is so common. The bigger number of occurrences, the smaller $idf(w_k^i)$. It will led to the decrement of $tfidf(w_k^i)$.

2.1.3 Cosine Similarity Measure

Cosine similarity is a real value reflecting the difference of two non-zero vectors based on the cosine of its spatial angle. To apply Cosine similarity for determining whether the semantic of two sentences are equivalent or not, we first need to transform two sentences s_1, s_2 into two non-zero vectors v_1, v_2, respectively. Then the value of Cosine Similarity between v_1 and v_2 is the distance between s_1 and s_2.

Definition 2.1.3. Let v_a and v_b are two non-zero real vectors with n dimensions:

$$v_a = (a_1, a_2, a_3, \ldots, a_{n-1}, a_n)$$

$$v_b = (b_1, b_2, b_3, \ldots, b_{n-1}, b_n)$$

The Cosine similarity between v_a and v_b is defined by:

$$sim(v_A, v_b) = \frac{v_a \cdot v_b}{\|v_a\|\|v_b\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{2.6}$$

2.2 Machine Learning Techniques

Machine Learning (a branch of Artificial Intelligence) is a set of algorithms that enables machines to automatically learn from previous experiences and after that, it can be able to adapt and predict new unseen data. Recently, Machine Learning is regularly used in many problems such as classification, regression, clustering, and detection. In fact, Machine Learning can be combined with other technologies to create more efficient algorithms for analyzing a huge amount of information. Based on how the machine learning algorithms "learn" from data, it can be classified into the following categories:

- *Supervised learning algorithm:* This is the algorithm which uses labeled training data to learn a generalized model, then uses this model to predict unlabeled testing data. Some famous supervised learning algorithms are Support Vector Machine, Neural Networks, Decision Tree, and Linear Regression.

- *Unsupervised learning algorithm:* Opposite to supervised learning algorithms, this kind of algorithm uses the unlabeled data to learn a generalized model for describing a latent structure behind the data. Although it cannot figure out the label of data, it still can divide data into some clusters. Some popular algorithms are K-means, K-nearest Neighbor, and Latent Dirichlet Allocation.

- *Semi-supervised learning algorithm:* This kind of algorithm uses both labeled and unlabeled data for the training process. Basically, it is used when we have a large set of unlabeled data and a small set of labeled data. In this case, labeled data will be used to improvement the accuracy of training process.

- *Reinforcement learning algorithm:* In some specific situation, the machine learning algorithm need to interact and adapt to its environment (e.g., games' AI bot). It uses the feedback and errors from the environment to improve itself, then after that, it can make better decisions.

In our works, a probabilistic unsupervised machine learning model called Latent Dirichlet Allocation (LDA) has been applied to represent a document by a distribution of topics. The basic idea of LDA is that each document is considered as a distribution of hidden topics d where each topic is a distribution of words. Then LDA tries to figure out d based on the training data.

2.2.1 Probabilistic Distributions

A probability distribution is a function describing the proportion of random variables that occurs in an event. For example, in the coin flipping scenario, assume the coin is fair, the probabilities of *head* and *tail* would be $0.5, 0.5$ respectively. It follows the Bernoulli distribution with the $\lambda = 0.5$. This section briefly introduces some probabilistic distributions used in the machine learning model that has been applied in our proposed method.

Poisson Distribution

Poisson Distribution is a discrete probability distribution, describes the average number of successful occurrences of an event e over a given time period t. Consider a random discrete variable N, if the average number of occurrences of N over t is λ, the probability of e occurs k times (k is a non-negative integer) is defined by:

$$p(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{2.7}$$

where $e \approx 2.71828$ (Euler's number)

Categorical Distribution

In many cases, the output of discrete random variables may be one value in a finite set. For example, when you roll a dice, the receiving face is one value in the set $\{1, 2, 3, 4, 5, 6\}$. In this case, we usually use Categorical distribution to describe random variables. Assume there are N possible outcomes, the outputs would be described by one element in the set $\{1, 2, \ldots, N\}$. Then, the Categorical Distribution is described by N non-negative parameters:

$$\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_N) \tag{2.8}$$

where:

$$\begin{cases} \lambda_i \geq 0 \\ \sum\limits_{i=1}^{N} \lambda_i = 1 \end{cases} \tag{2.9}$$

Each λ_i represents the probability of output to be i: $p(x = i) = \lambda_i$.
The probability density function of Categorical Distribution is defined as:

$$p(x) = Cat_x[\lambda] \tag{2.10}$$

Dirichlet Distribution

Dirichlet Distribution is used to describe the parameters of Categorical Distribution. It describes N continuous random variables $\lambda_1, \lambda_2, \ldots, \lambda_N$, where

$$\begin{cases} \lambda_i > 0 \\ \sum\limits_{i=1}^{N} \lambda_i = 1 \end{cases} \tag{2.11}$$

Using N positive parameters used to describe a Dirichlet Distribution: $\alpha_1, \alpha_2, \ldots, \alpha_N$, the probability density function is defined by:

$$p(\lambda_1, \lambda_2, \ldots, \lambda_n) = \frac{\Gamma(\sum\limits_{n=1}^{N} \alpha_n)}{\prod\limits_{n=1}^{N} \Gamma(\alpha_n)} \prod_{n=1}^{N} \lambda_n^{\alpha_n - 1} \tag{2.12}$$

where:

$$\Gamma(z) = \int_0^\infty t^{z-1} \exp(-t) dt \tag{2.13}$$

For short, it can be written as: $p(\lambda_1, \lambda_2, \ldots, \lambda_k) = Dir_{\lambda_1,\lambda_2,\ldots,\lambda_k}[\alpha_1, \alpha_2, \ldots, \alpha_K]$

2.2.2 Latent Dirichlet Allocation

In order to extract the characteristics of documents, in information retrieval and data mining, modeling the documents is an essential task before proceeding next steps. Compare to the words vectorization methods, documents represented by probabilistic distribution has many advantages. Latent Dirichlet Allocation (LDA) [17] is a generative probabilistic model over discrete data, which aims to discover the "*hidden topics*" in a corpus. In LDA, each document is represented by a distribution of hidden topics and a topic is a distribution of words. Because LDA is a bag-of-words model, the order of words does not matter. Beside being applied in text mining, it also is used in many other domains as long as the purpose is to capture the hidden structure of the dataset.

Notation

Assume we have a vocabulary ϑ, indexed by $\{1, 2, \ldots, V\}$.
A *word* is an element from ϑ, is represented by an one-hot vector. It means the $v$word is defined by a vector $w = (w^1, w^2, \ldots, w^V)$, in which $w^i = 1$ if $i = v$, otherwise, $w^i = 0$.
A *document* w is represented by a vector of N words, in which w_i is the ith word in the document:

$$w = (w_1, w_2, \ldots, w_N) \tag{2.14}$$

A *corpus* (or *collection*) is a set of M documents:

$$D = \{w_1, w_2, \ldots, w_M\} \tag{2.15}$$

Assumption

We need to assume a fixed number of topics, say K. Then we have K topics:

$$T = \{t_1, t_2, \ldots, t_k\} \tag{2.16}$$

Then assume the generation of document w in D follows these step:

1. First, decide the number of words N for D according to a Poisson distribution:
$$N \sim Poisson(\xi)$$
2. Second, choose a mixture of K topics for w according to a Dirichlet distribution:
$$\theta \sim Dir(\alpha)$$
3. Third, we next generate each words w_i in w by:
   - Pick a topic according to the multinomial distribution chosen in the second step: $t_i \sim Multinomial(\theta)$
   - Then use this chosen topic above to generate a word w_i according to a multinomial probability on the topic t_i

Then, LDA tries to learn the topic distribution of each document, and the words associated for each topics. The learning process is performed as follows:

1. For all document w in D, assign each word w_i in w to a random one in K topics.
2. After this step, we already have topic distribution of all w_i, and word mixtures of all topic. But this result is still not good. We need to improve it by performing some iterations:
   - For all document w in D:
   - For all word w_i in w:
   - For each topic t_j:
   - Calculate:
   $$p = p(t_j \mid w).p(w_i \mid t_j)$$
   where
     - $p(t_j \mid w)$: the proportion of words w_k in w in which $t(w_k) = t_j$
     - $p(w_i \mid t_j)$: the proportion of document in D containing w_i in which $t(w_i) = t_j$.

   After that, the w_i is assigned to the topic t_j with the new probability p.

After some iterations, when the state of the dataset is quite stable, the training can be stopped after some specific iterations or when the perplexity of model is good enough. This model now can be used for inferring topics distribution of a document.

Example 2.2.1. Assume we have following sample documents:

- s_1 : I love eating apple when walking with my dog at night.
- s_2 : I like some sports such as basketball, tennis, and walking.
- s_3 : My dog and her cat are playing with the balls.

- s_4 : Today we have apple, orange, and kiwi for a tea break after the tennis match.

Assume we choose the number of hidden topics that need to be discovered is 3 and each topic is a distribution of keywords such as:

- t_1 : apple (0.25), orange (0.2), kiwi (0.18), sports (0.1), dog (0.05), cat (0.04), ...
- t_2 : dog (0.4), cat (0.3), apple (0.15), basketball (0.05), ...
- t_3 : walking (0.35), basketball (0.2), tennis (0.2), orange (0.1), cat (0.05), ...

In fact, t_1, t_2, t_3 are hidden topics. It means we do not know exactly the label of them. However, t_1 could be interpreted as "*fruit*", t_2 could be interpreted as "*animal*", and t_3 could be interpreted as "*sport*". Now, each sentence s_i can be represented as a distribution of topics:

- s_1: t_1 (0.4), t_2 (0.3), t_3 (0.3)
- s_2: t_1 (0.1), t_2 (0.1), t_3 (0.8)
- s_3: t_1 (0.1), t_2 (0.75), t_3 (0.15)
- s_4: t_1 (0.8), t_2 (0.05), t_3 (0.15)

Chapter 3

IoT Malware Analysis

To satisfy the low power requirements, IoT devices tend to use lightweight processors. The most popular processor is being used is ARM. As the result, IoT malware is typically based on ARM architecture. Even though IoT Malware is not much complex as x86/windows malware, by our investigation, it also contains obfuscation techniques in order to hide its behaviors. It leads to some difficulties in detection and analysis because we cannot correctly trace its execution with commercial disassemblers like IDA Pro [14] and Capstone [15].

3.1 Obfuscation Techniques

Being obfuscated makes more difficulties for disassemblers to correctly trace the assembly code of malware. As a result, it has more opportunities to survive. Even commercial disassemblers like IDA Pro and Capstone are robust, they are still easily fooled by obfuscation technique such as indirect jump and overlapping instruction. In this section, we present some typical obfuscation techniques frequently appearing in IoT malware.

Indirect Jump

An indirect jump (also known as an indirect branch) is a control instruction, in which, instead of clearly determining the next instruction to be executed by an address, the target address is encoded itself and stored indirectly in memory or general-purpose register. Unless the instruction is executed, the specific value of the address to be jumped is unknown. The example below shows an indirect jump in an IoT malware *48cff3f21c* provided by Prof. Katsunari Yoshioka (Yokohama National University). At `9a50`, the `bx` instruction executes a jump operation to the address specified by the value stored in the register `lr`. To deal with this technique, it is necessary to use a binary emulator to get this specific value.

```
 ...
9a48:  e24bd00c sub sp,fp,#12
9a4c:  e89d6800 ldm sp,{fp,sp,lr}
9a50:  e12fff1e bx lr
9a54:  e1a0c00d mov ip,sp
9a58:  e92dd800 push{fp,ip,lr,pc}
...
```

Figure 3.1: Indirect jump in an IoT malware

Dead Code Insertion

This is a simple technique to modify the structure and appearance of malware, but does not affect its semantic and behaviors. By modifying itself code, it works quite well to protect against some traditional detection method in typical commercial antivirus software like bit-based signatures. In this technique, some ineffective instructions were inserted between the original instructions, then make the assembly code seems more complex, but actually it plays no role in the behavior of malware.

```
                                    ...
                                  d138:  add r3,r3,r0
  ...                             d13c:  nop
d138:  add r3,r3,r0               d140:  add r4,r3,#7
d13c:  add r4,r3,#7               d144:  mov r5,#5
d140:  mov r5,#5                  d148:  bl 8ecc
d144:  bl 8ecc                    d14c:  add r5,#1
d148:  bcs d5c4                   d150:  sub r5,#3
d15c:  ldr r5,[fp,#-56]           d154:  bcs d5c4
...                               d158:  add r5,#2
                                  d15c:  ldr r5,[fp,#-56]
                                  ...
```

Figure 3.2: Original malware assembly code and after dead code insertion

In this figure, we can easily real the problem at `d13c`, this is a *"no-operation"* instruction, means that doing nothing here. In addition, at `d14c`, `d150`, and `d158` the value of the register `r5` is modified three times, but after all, the value stored in `r5` is unchanged. Although these codes are inserted to make the program more complex, it just can modify the structure, but worths nothing for the behaviors of malware.

3.2 Typical Approaches

Static Analysis

Static analysis aims to analyze the executable binary files of malware without running it. Some typical methods of static analysis are signature based and behavior based.

- In signature-based approach, the hash value of binary files, or a byte sequence in binary files is used to check whether a new binary file is a malware or not by comparing the signature hash value of this file with the existing hash value stored in a centralized database. This database is regularly updated. This approach is used in many antivirus software because of its light-weight and correctness. But the biggest problems is that the number of unique signatures is huge, as well as modern malware usually use various polymorphic and metamorphic techniques.

- In behavior-based approach, the data flow and control flow of malware are statically explored, then it is used to describe the behavior of malware. These flows are usually represented by directed graphs. Analyzing malware behaviors now becomes checking these graphs. Basically, if malware is totally clear without any obfuscation techniques, this method may work well. However, modern malware is usually obfuscated.

Dynamic Analysis

Dynamic Analysis aims to analyze malware by actually running it. Based on the specific behavior and the way of how malware affects to the environment during runtime process, we can understand its functionalities and then, explore some indicators as its signature. Some typical indicators are API calls, connected IP addresses or domains, and behavior of downloading some files. Although almost dynamic analysis methods are slower than static analysis, it reflects more precise about malware behaviors. In fact, dynamic analysis and static analysis are usually combined together to improve the accuracy of detection and classification.

Machine Learning Approach

Malware detection based on Machine Learning approaches are attracting a lot of attention. In the past, when the machine learning model was not robust, and the data set is not big enough, this approach seemed to be inefficient. But currently, many methods gain impressive results. Some typical methods are introduced below:

- Malware as an image: This idea considers a malware as an image, then transforms malware into gray-scale images. After that, by using a Convolution Neural Network (CNN) over transformed images, the detection or classification are performed. For instance, in 2017, Jiawei Su et al. proposed a lightwight classification for IoT malware [18]. His approach converts a binary code to a fixed-size color image and uses AlexNet [19] to perform the classification task.

- Malware as a sequence: This idea considers a malware as a document with a sequence of byte, in which each word is a group of bytes. Then it applies some typical machine learning algorithms to perform the detection or classification. For instance, in 2017, a malware detection from raw byte sequences was introduced by E. Raff et al [20].

This method has a good performance with linear complexity dependence on the length of sequence.

Model-checking Based

To make the machine understand and work with real-world systems, it should be simulated by a model in the computer written by logic or mathematics. Malware is not an exception, it usually is represented by a model M, then the all the scenarios and behaviours of M are tested by using formal verification methods. A typical model for binary file is Control Flow Graph (CFG), which reflects its execution steps. After that, the malware classification and detection problems are equivalent to checking the represented model M. Based on model-checking, many algorithms have been proposed to detect malware. For instance, in 2018, Anh Viet Phan et al. introduces a graph-based Convolutional Neural Network approach [21] to perform software defects detection and malware analysis.

3.3 Malware Analysis Based on Control Flow Graph

One of the drawbacks of traditional malware analysis methods is that it can not overcome obfuscation techniques. To deal with this problem, Dynamic Symbolic Execution (concolic testing) is applied in order to explore the Control Flow Graph (CFG) of malware. A CFG is a representation model of the program execution process. It is a directed graph which node is a block without any jumps and each edge represents the path condition from the parent node. CFG can explore all feasible paths from the initial state to terminated state during the execution.

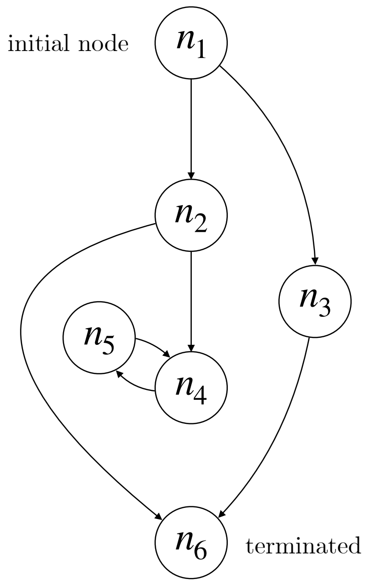
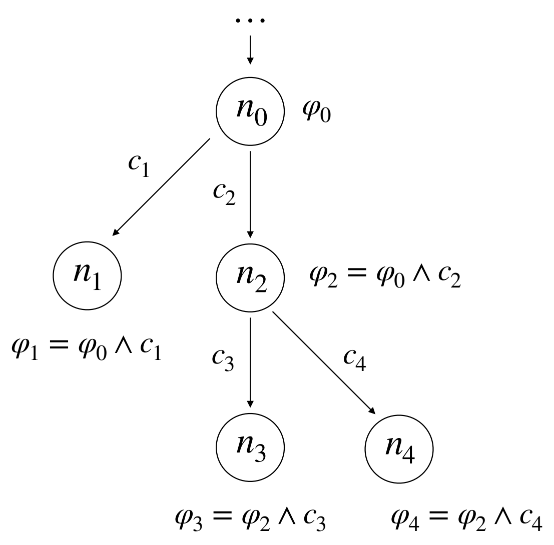


Figure 3.3: A Control Flow Graph of Binary File

This figure above shows an example of CFG. It can be interpreted that, node n_1 and n_2 may causes an $if - else$ statement, node n_4 and n_5 may causes a *while* loop. For binary files, the data instructions such as `mov, ldr, add` do not affect the jump operations, therefore it can be grouped into a node. If the execution reaches a jump instruction (e.g., `b` in ARM and `jmp` in x86), the branch is created by deciding the satisfiability of the current path constrain with a Theorem Prover.

3.3.1 Dynamic Symbolic Execution

Symbolic Execution [5] is a technique has been used regularly in software testing, to execute a program symbolically rather than running it with concrete input values. By using Symbolic Execution, all available scenarios during the execution can be caught. In this technique, inputs are assumed as symbolic values like α, β; then in every conditional branch, it adds the constraints in the path condition *pc*. By using this approach, the Symbolic Execution can run through all possible path of a program and do the testing all possible outcomes. This technique also is used in software security to detect potential vulnerabilities.



...
n_0
φ_0
c_1
c_2
n_1
n_2
$\varphi_2 = \varphi_0 \wedge c_2$
$\varphi_1 = \varphi_0 \wedge c_1$
c_3
c_4
n_3
n_4
$\varphi_3 = \varphi_2 \wedge c_3$
$\varphi_4 = \varphi_2 \wedge c_4$

Figure 3.4: Symbolic execution for automatic tests generation

The figure above illustrates how symbolic execution is applied to cover all possible scenarios of a program. Assume the path condition from the initial node of n_i is φ_i, and the condition from n_{i-1} to n_i is c_i. Then, for each path, the satisfiability of the path condition is checked (e.g, φ_1, φ_2, φ_3, φ_4) by *Z3*. If it is SAT, an instance of input values causing this path is collected as a test-case. There are two ways to explore next destinations in Symbolic Execution: Static Symbolic Execution (SSE) and Dynamic Symbolic Execution (DSE).

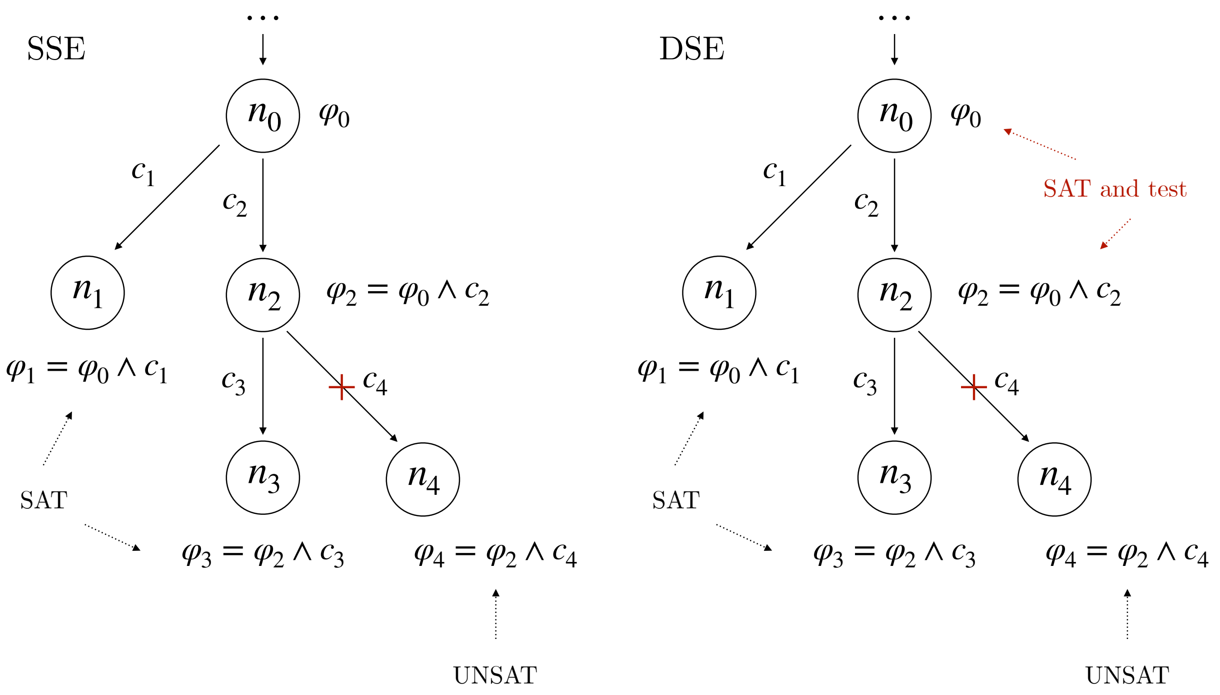

SSE
n_0 φ_0
c_1 c_2
n_1 n_2 $\varphi_2 = \varphi_0 \wedge c_2$
$\varphi_1 = \varphi_0 \wedge c_1$
c_3 c_4
n_3 n_4
SAT
$\varphi_3 = \varphi_2 \wedge c_3$ $\varphi_4 = \varphi_2 \wedge c_4$
UNSAT
DSE
SAT and test

Figure 3.5: Static Symbolic Execution and Dynamic Symbolic Execution

- **Static Symbolic Execution:** The next candidates are statically decided by checking the feasibility of each path $\varphi_2 = \varphi_1 \wedge c_2$, $\varphi_3 = \varphi_1 \wedge c_3$, and $\varphi_4 = \varphi_1 \wedge c_4$. One of the problems of SSE is that, for some specific conditions, the theorem provers cannot easily check the satisfiability of the candidates. For instance, if c_2 keeps the condition: $(x * x * x + y * y * y = z * z * z)$, in which x, y, z are integer numbers, it is really hard to decide whether it is SAT or UNSAT. In this case, the DSE should be used, in which a concrete instance of φ_0 should be used in order to test the feasibility of the path.

- **Dynamic Symbolic Execution (Concolic Testing):** This technique is the combination of concrete execution and static symbolic execution to overcome the drawback mentioned above. I also be used to explore obfuscation techniques like *indirect jump* in malware. In DSE, to decide the next candidates, the feasibility is checked by using a satisfiable instance of φ_1. It requires a binary emulator to get the value of the concrete variables.

To deal with obfuscation techniques in malware like *indirect jump*, we need to use DSE because at that time of executing jump instruction, its target is an expression of symbolic values that need to be dynamically determined by using a satisfiable instance of the current path constrain. For instance, if there is a jump instruction `b lr` at n_2, a satisfiable instance of the path condition φ_2 will be used to get the concrete value of `lr`, then builds the CFG based on this determined target.

3.3.2 On-the-fly CFG Generation

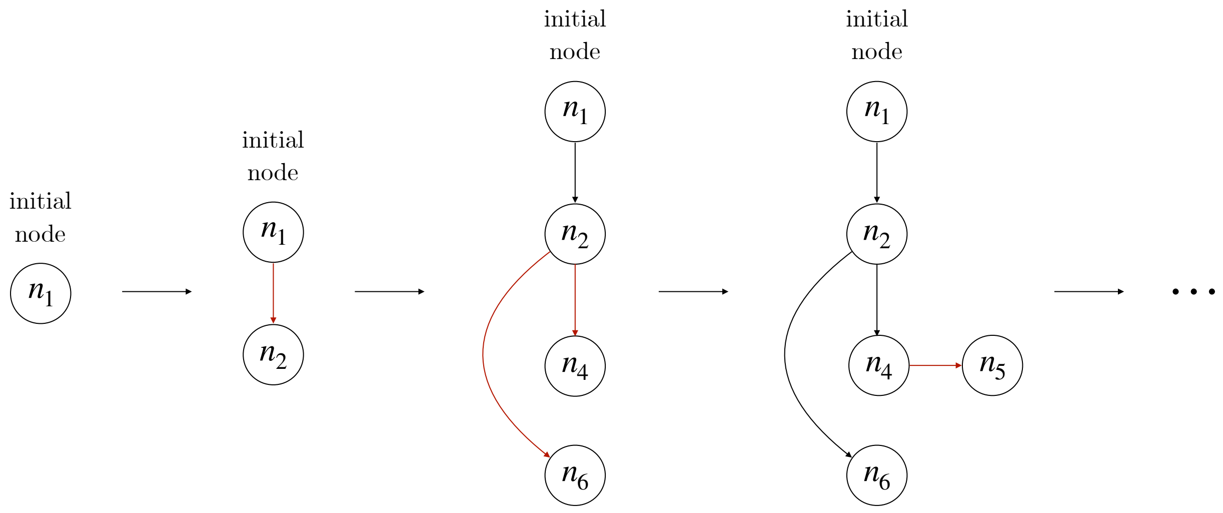


Figure 3.6: Control Flow Graph Generation by on-the-fly manner

The figure above illustrates how Control Flow Graph is generated by the on-the-fly manner. When an conditional instruction is reached, concolic testing will be applied to decide the feasible paths (red color in the figure). At each execution step, the state of a binary program is updated, including the environment of binary emulator (flags, registers, stack, and memory). This procedure is interrupted at the end of program or when reaching an unsupported instruction.

3.3.3 BE-PUM for x86/Windows

BE-PUM (Binary Emulation for PUshdown Model) [9] is a binary code analyzer concentrating on malware on Intel x86/Win32 architecture. BE-PUM inputs a binary file then applies the Dynamic Symbolic Execution technique (DSE) to generate the Control Flow Graph (CFG) of binary files in on-the-fly manner. For analyzing binary files, BE-PUM uses *JackStab 0.8.3* [22] as the disassembler, and *Z3 4.3* [23] as the theorem prover to perform test instances in the DSE process.

The figure[1] below shows the architecture of BE-PUM, which consists of three main elements: a CFG storage, a binary emulator, and a symbolic execution.

[1]These figures are redrawn from the original images in the paper [9]

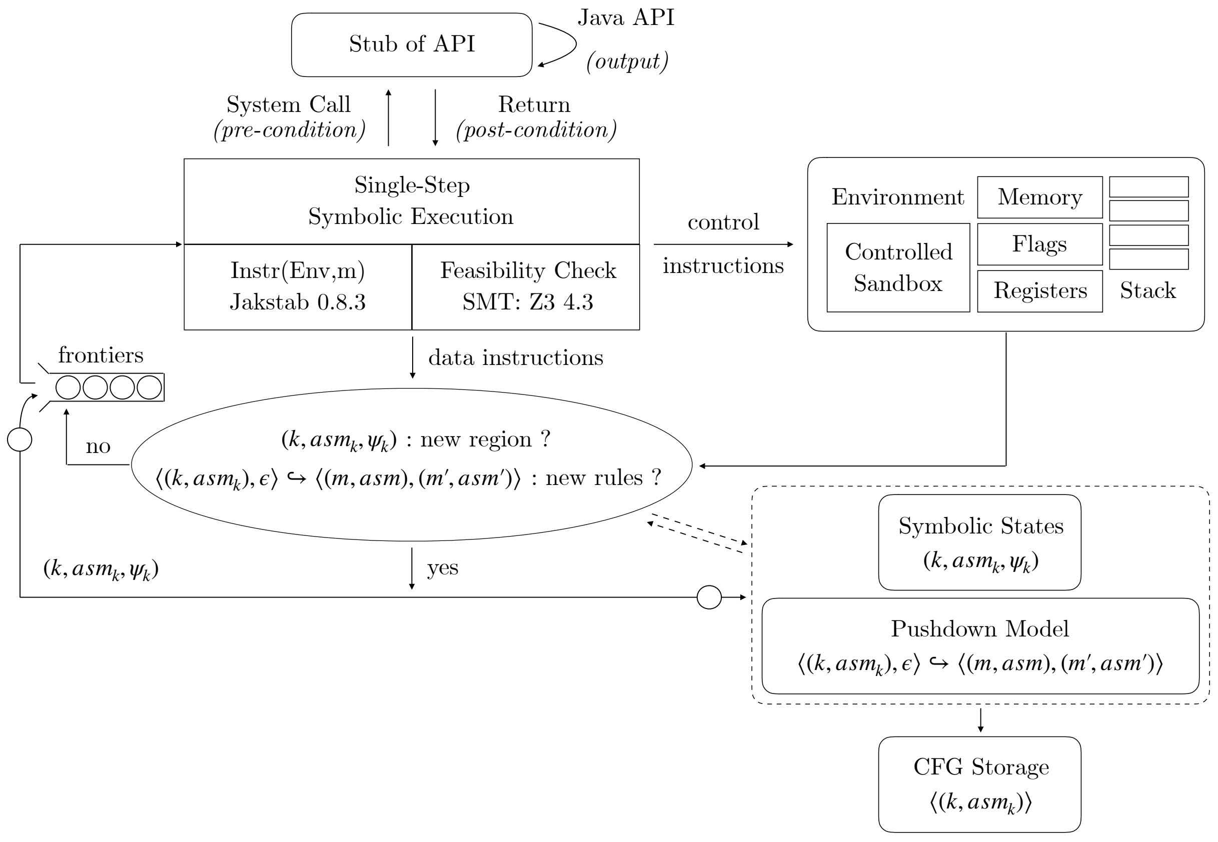


Figure 3.7: BE-PUM Architecture

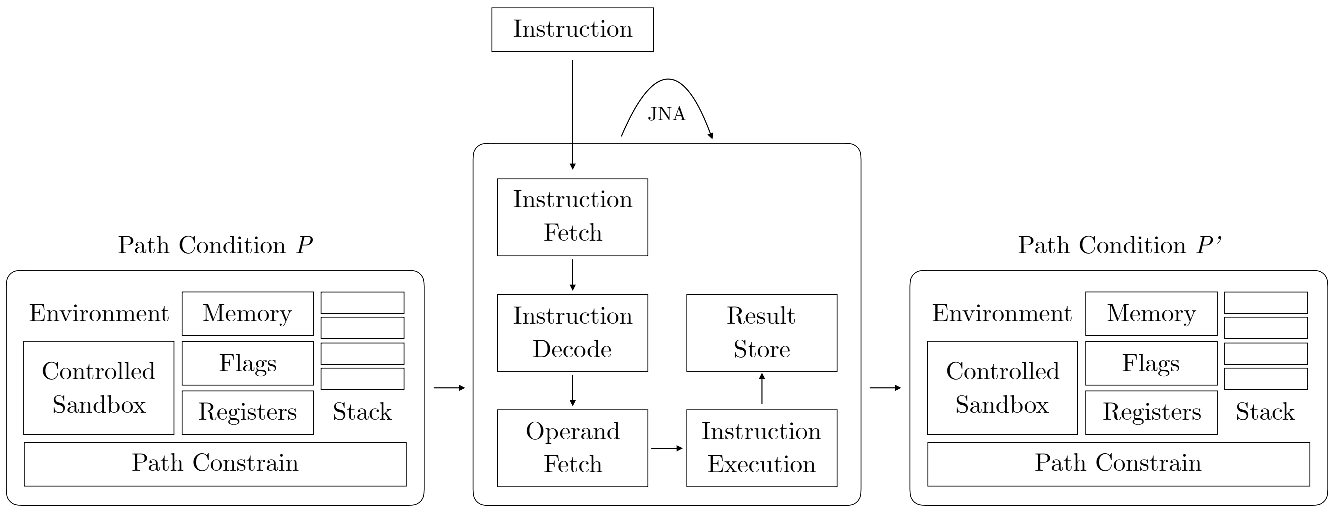


Figure 3.8: Binary emulator in BE-PUM

The binary emulator in BE-PUM executes the instructions. If an instruction is a branch operation, the binary emulator updates the path constraints and the environment in the *pre-condition P* to the *post-condition P'*. Otherwise, the binary emulator only updates the environment. In BE-PUM, path constraints keep a boolean expression of symbolic values, and the environment holds the states of Memory, Stack, Registers, and Flags.

Chapter 4

ARM Formal Semantics

4.1 ARM Processor

ARM (shortened of Acorn RISC Machines) is a computer processors family of RISC architecture. This section briefly introduces the main components of ARM, including Stack, Memory, Register, and Flags.

4.1.1 Architecture

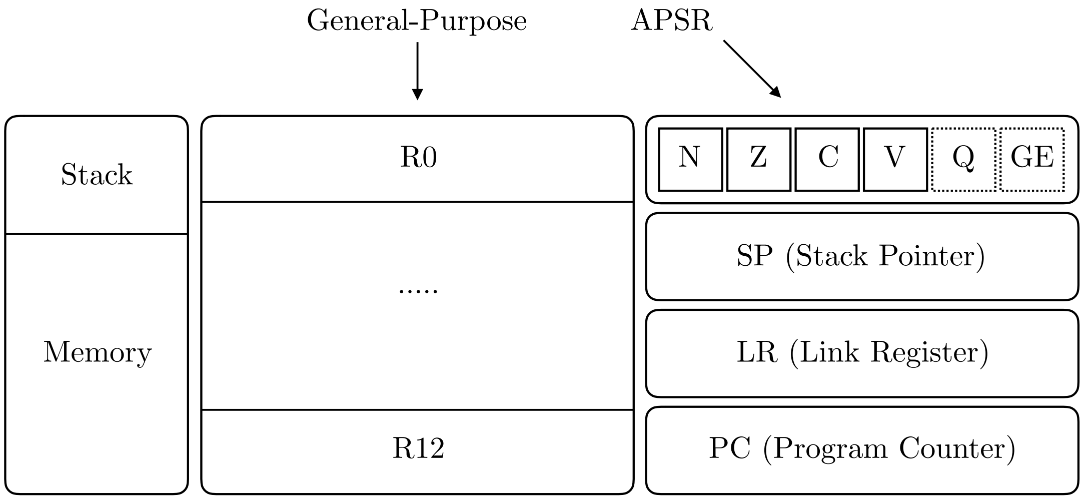


Figure 4.1: Components of ARM Architecture

This figure illustrates the main components in ARM architecture:

1. **Memory:** This is a physical device for temporarily stores information used during the computer operations.

2. **Stack:** Stack is a part of memory, stores temporary variables generated by the execution process.

3. **Register:** This is a place in the processor to hold data. There are five different types of registers in ARM Processors:
    - *General-purpose registers:* It contains 13 registers marked from R0 to R12.
    - *Stack Pointer (SP):* This is a register which points to the last value was stored in the stack.
    - *Link Register (LR):* The LR register keeps the address returned from a function call.
    - *Program Counter (PC):* (or instruction pointer) The PC register keeps the address of next instruction that needs to be executed.
    - *Application Program Status Register:* One APSR register. It keeps conditional flags (N, Z, C, V). In some specific versions of ARM, it also holds GE and Q flags.
4. **Flags:** Flags are binary values to store states in the executed operations. The value stored in flags is either *True (T)* or *False (F)*. An instruction may need to check the boolean value of flags before execution and after executing, it may also update some specific flags. In ARM, flags are stored in the APSR register.
    - Negative flag (N): This flag is set by an operation if the result is negative.
    - Zero flag (Z): If the result of an instruction is zero, it is set by $true$, otherwise $false$.
    - Carry flag (C): When an unsigned operation's result overflows the capacity of 32-bit register, this flag is set.
    - Signed Overflow flag (V): The flag works the same as the C flag, but for signed operations.
    - Q: This is one of the program status flags in the APSR. It is used to indicates overflows or saturation of instruction result in only the E variants ARM-v5 or later.
    - GE: The GE flags only exist in ARM-v6 and later. It can be set during the execution of parallel operations.

4.1.2 Cortex Series

The most popular architecture of ARM family is Cortex. It includes different series for various purpose. In general, it can be categorized into three main series: Cortex-A, Cortex-M, and Cortex-R.

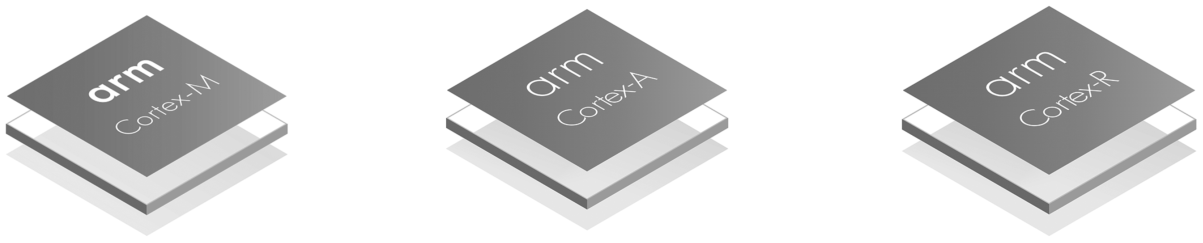


Figure 4.2: Popular Cortex series of ARM

Cortex-A

The ARM Cortex-A is the highest performance processor in ARM family. It is optimized for rich operating systems such as Android or Linux. It plays a role as the heart in the powerful technology products like smart-phones, tablet, laptop devices. The Cortex-A series can be categoried by three families: highest performance (A7X series), performance and efficiency (A5X), and lowest power (A3X).

Cortex-M

The ARM Cortex-M is the lowest power processors, which is optimized for real-time embedded processing and micro-controller uses. As a report from ARM[1], this family has already been shipped in tens of billions of device. The Cortex-M series can be categoried by three families: lowest power (M0, M0+, M23), performance efficiency (M3, M4, M33, M35P), and highest performance (M7).

Cortex-R

The ARM Cortex-R is the family of ARM architecture that is optimized for real-time applications. Not only offers high performance, it also satisfies the requirements of real-time applications such as solid-state drive controllers.

4.1.3 Instructions

Each Cortex series has a different number of instructions. Even some shared instructions are used in many series, there are still some differences between other architectures. For example, the instruction `SSAX` in the Cortex-M4 has the same operation as `SSAX` in Cortex-M7, but the flags update of `SSAX` in Cortex-M4 is *"This instruction does not affect the condition code flags"* and the flags update of `SSAX` in Cortex-M7 is *"This instruction set the APSR.GE bits according to the results"*. Because we do not know exactly when they are different, we still need to process all instructions. In 5.2, an effective strategy is introduced to decide which one should be implemented.

[1]https://arm.com/products/processors/cortex-m

Variation	Number of instructions
Cortex-M0	63
Cortex-M0+	63
Cortex-M3	129
Cortex-M4	244
Cortex-M7	261
Cortex-M33	297
Total	1039

Table 4.1: Number of collected instructions over six architecture

The table above shows the number of instructions for each Cortex-M variation that we have successfully collected from ARM Developer Website. Each ARM instruction consists of: `$name$suffix{$cond} $params` where:

- `$name`: Instruction name.
- `$suffix`: Conditional suffix (optional). This is the precondition that need to be checked before execution. There is 15 different values of suffix, as described in 4.2.
- `$cond`: Flags update condition (optional). In general, if the `$cond` appears, this instruction updates the APSR flags based on the result of the operation.
- `$params`: Instruction's parameters. It can contains some conditional params separated by |. For instance, in the instruction `ADD{S} {Rd,} Rn, <Rm|#imm>`, the third parameter can either be a value stored in the register `Rm` or a specific value `#imm`.

Example 4.1.1. An Cortex-M7 instruction: `UMAALNE RdLo, RdHi, Rn, Rm`

In this example, the instruction name is `UMAAL`. The parameters are `RdLo`, `RdHi`, `Rn`, and `Rm`. The flags update condition is omitted. It means the flags will not be modified after execution. The conditional suffix is `NE` means that, this instruction needs to check this condition before execution. The table below shows all possible conditional suffix in ARM and its meaning.

Suffix	**Meaning**	**Condition**
EQ	Equal	Z
NE	Not equal	$\neg Z$
CS or HS	Carry set	C
CC or LO	Carry clear	$\neg C$
MI	Negative	N
PL	Positive or zero	$\neg N$
VS	Signed overflow	V
VC	No signed overflow	$\neg V$
HI	Unsigned higher	C $\wedge \neg Z$
LS	Unsigned lower or same	$\neg C \wedge Z$
GE	Signed greater than or equal	$N = V$
LT	Signed less than	$N \neg = V$
GT	Signed greater than	$\neg Z \wedge N = V$
LE	Signed less than or equal	Z $\wedge N \neg = V$
AL (or omitted)	Always executed	None

Table 4.2: ARM Conditional Suffix[2]

4.2 Formal Semantics of ARM

There are several works of the formal semantics of binaries, especially for x86, e.g., operational semantics for self-modifying programs [24]. The formal semantics of binary is a basis for the implementation of binary code analyzers, e.g., BINCOA [25], MiAsm [26], McVeto [7], CoDisasm [8], BE-PUM [9] KLEE-MC [10], Jakstab [27], BAP [28]. Although the semantics of binaries is intricate for human, the semantics framework is quite simple, which consists of four ingredients: *registers*, *flags*, *memory*, and *stack* (a part of the memory). We omit the multi-threads and the weak memory model, since our target, IoT malware, is mostly a sequential user-mode process. An execution of each instruction is regarded as a transition on the quadruplets as in the figure below:

[2]https://community.arm.com/

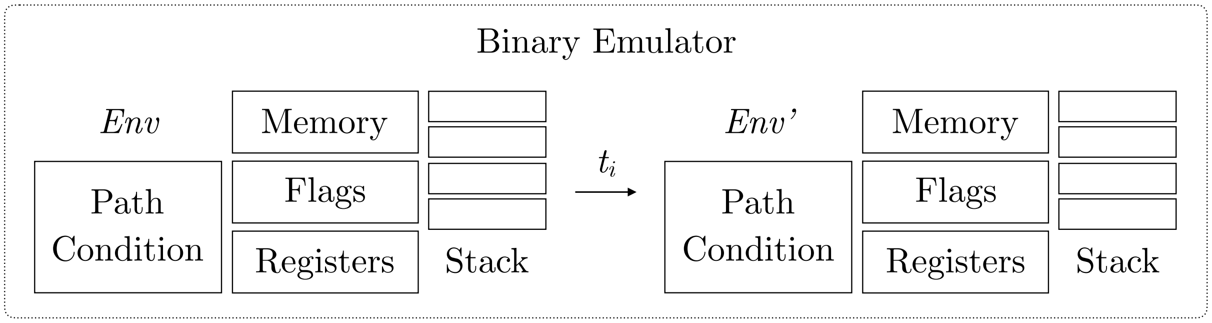


Figure 4.3: The semantics transition framework of ARM

4.2.1 Abstract Environment

Definition 4.2.1. The environment model $E = \langle F, R, M, S \rangle$ of the 32-bits ARM Cortex-M binary code consists of:

- F: a set of 6 flags: $F = \{N, Z, C, V, Q, GE\}$
- R: a set of 17 registers:

  $$R = \{r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, sp, lr, pc, apsr\}$$

  where *apsr* is a special register storing the values of all flags N, Z, C, V (also includes Q, GE in some particular versions of ARM).
- M: a set of n contiguous memory locations: $M = \{m_0, m_1, \ldots, m_{n-1}\}$
- $S(\subseteq M)$: a set of k contiguously allocated memory to store the stack: $S = \{s_0, s_1, \ldots, s_{k-1}\}$ with $k < n$.

At the beginning, each the components of environment keeps an initial symbolic values. After that, the values of environment is updated through each execution as a semantics transition. Note that, the register *pc* is pointing to the address of next instruction that needs to be executed and the register *SP* is pointing to the top of stack *S*.

4.2.2 Operational Transitions

$$\frac{R_{pc} = k;\ instr(k) = b\ i;\ R_i = m}{\langle F, R, M, S \rangle \to \langle F, R[pc \leftarrow m], M, S \rangle}\ [\mathrm{B}]$$

$$\frac{\begin{array}{c} R_{pc} = k; instr(k) = subs\ i\ j\ h; R_j = m; R_h = n; a = m - n; \\ N' = (a < 0); Z' = (a = 0); C' = (a \geq 2^{32} - 1); V' = (a \leq -2^{31}) \vee (a \geq 2^{31} - 1) \end{array}}{\langle F, R, M, S \rangle \to \langle F[N \leftarrow N'; Z \leftarrow Z'; C \leftarrow C'; V \leftarrow V']; R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow a], M, S \rangle}\ [\mathrm{SUBS}]$$

$$\frac{R_{pc} = k;\ instr(k) = mov\ i\ j; R_j = m}{\langle F, R, M, S\rangle \rightarrow \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow m], M, S\rangle}\ [\text{MOV}]$$

$$\frac{\begin{array}{c} R_{pc} = k;\ instr(k) = ands\ i\ j\ h;\ R_j = m; R_h = n; a = m\ \&\ n; \\ N' = (a < 0); Z' = (a = 0); C' = (a \geq 2^{32} - 1); V' = (a \leq -2^{31}) \vee (a \geq 2^{31} - 1) \end{array}}{\langle F, R, M, S\rangle \rightarrow \langle F[N \leftarrow N'; Z \leftarrow Z'; C \leftarrow C'; V \leftarrow V'], R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow a], M, S\rangle}\ [\text{ANDS}]$$

$$\frac{R_{pc} = k;\ instr(k) = neg\ i\ j;\ R_j = n}{\langle F, R, M, S\rangle \rightarrow \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow !n], M, S\rangle}\ [\text{NEG}]$$

$$\frac{\begin{array}{c} R_{pc} = k;\ instr(k) = cmp\ i\ j;\ R_i = m; R_j = n; a = m - n; \\ N' = (a < 0); Z' = (a = 0); C' = (a \geq 2^{32} - 1); V' = (a \leq -2^{31}) \vee (a \geq 2^{31} - 1) \end{array}}{\langle F, R, M, S\rangle \rightarrow \langle F[N \leftarrow N'; Z \leftarrow Z'; C \leftarrow C'; V \leftarrow V'], R[pc \leftarrow k + \mid instr(k) \mid], M, S\rangle}\ [\text{CMP}]$$

$$\frac{R_{pc} = k;\ instr(k) = ldr\ i\ j;\ M_j = m}{\langle F, R, M, S\rangle \rightarrow \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow m], M, S\rangle}\ [\text{LDR}]$$

This figure above shows some examples of the operational semantics of ARM instructions, based on the description on ARM Developer Website: *b, subs, neg, and, ldr, mov, cmp.* That mean the indirect jump, the subtraction, the negation, the bitwise AND, the load, the move, and the comparison respectively. For instance, the "*subs i j h*" first takes the subtraction of two values stored in the registers j and h, and stores the result $a = m$ - n to the register i. The flags are updated such that:

- N is set by *true* if a is negative, otherwise *false.*
- Z is set by *true* if a is not zero, otherwise *false.*
- C and V are set by *true* if a overflows for unsigned and signed operations, respectively.

4.2.3 Java Specifications as Semantics

Our motivation of the formal semantics is an automatic generation of the dynamic symbolic execution tool for ARM (similar to CoDisasm [8], BE-PUM [9], MiAsm [6], Klee-MC [10]). The formal semantics of each ARM instruction is represented as a Java method of **`BitVec`** class, which is obtained by instantiating this dynamic template:

```
public void $name($params, Character suffix, Character cond) {
   env.arithmeticMode = $arithmeticMode;
   if (cond == null || checkCond(cond)) {
      char[] flags = new char[]{$flags};
      BitVec result = null;
      $execCode
      if (suffix != null && suffix == 's') {
         if (result != null) {
            env.updateFlags(flags, result);
         }
      }
   }
}
```

Note that:

- **`BitVec`** is a new class defined as a pair $\langle bs, s\rangle$, where bs is a **`BitSet`** value (a supported class in Java) representing the 32 bits BitVector, and s is a string value in SMT format to store either an expression (for the registers, the memory, and the stack) or a formula (for the flags).
- The blank variables need to be filled are:
  - `$name`: The instruction name.
  - `$params`: The arguments of the function.
  - `$arithmeticMode`: Show either the floating-point or the binary arithmetic.
  - `$flags`: The list of flags to be updated.
  - `$execCode`: The code for actions.
- Default parameters of the template are:
  - `suffix`: The suffix of the flags update. If the s suffix occurs, the flags might be updated.
  - `cond`: The conditional suffix of the instructions, showing the pre-condition.
- The environment `env` is the quadruplet of the flags, the registers, the memory, and the stack.
- Prepared functions:
  - `updateFlags`: Update flags if their names occur in `$flags`.
  - `checkCond`: Check the pre-condition provided by `cond`.
  - Corresponding to BitVector theory of SMT solvers, 35 basic methods are manually implemented, e.g., the arithmetic operators *add, sub, mul*, the bitwise operators *and, shift, xor*, the data operators *write, load, store*, and else *clz, saturate, rotate.*

    Since some operators on binary are primitive, they need to be manually implemented as basic functions for the binary emulator. After that, all generated Java methods work based on them. In total, 35 basic functions are prepared in 7 categories:

1. *Environment:*
   - `checkCond`: Check the pre-condition before execution.
   - `updateFlags`: Update the flags based on the result of operators.
2. *Jump Operator:*
   - `b`: Jump to an address or a value stored in a register (indirect jump).
3. *Bitwise Operators:*
   - `and`: Bitwise AND.
   - `or`: Bitwise OR.
   - `xor`: Bitwise XOR.
   - `not`: Bitwise NOT.
4. *Arithmetic Operators:*
   - `abs`: Absolute value of a BitVec value.
   - `div`: Division of two BitVec values.
   - `mul`: Multiplication of two BitVec values.
   - `add`: Addition of two BitVec values.
   - `sub`: Subtraction of two BitVec values.
   - `max`: The bigger value in two BitVec values.
   - `min`: The smaller value in two BitVec values.
   - `sqrt`: Square root of a BitVec value.
5. *IO Operators:*
   - `write`: Write a BitVec value to a register.
   - `val`: Get the BitVec value stored in a register.
   - `load`: Load the BitVec value in a memory location to a register.
   - `store`: Store a BitVec value in a register to a memory location.
   - `pop`: Pop a BitVec value from the stack.
   - `push`: Push a BitVec value to the stack.
6. *Bit-based Operators:*
   - `comp`: Complement of a BitVec value.
   - `neg`: Negation of a BitVec value.
   - `sat`: Saturate a BitVec value.
   - `shift`: Shift a BitVec value.
   - `rot`: Rotate a BitVec value.
   - `rev`: Reverse a BitVec value.
   - `signedExt`: Signed Extend a BitVec value.
   - `zeroExt`: Zero Extend a BitVec value.
7. *Others:*
   - `clz`: Count the number of leading zeros in a BitVec value.
   - `round`: Round a BitVec value.
   - `cmp`: Compare two BitVec values.
   - `clearBitfield`: Clear BitField of a register.

  * `copyBitfield`: Copy BitField to a register.
  * `convertEndian`: Convert between 16-bit signed big-endian data and 32-bit signed little-endian data.

Fixed-size BitVector Theory

Using SMT format to represent path constraints is an effective way, however, linear and nonlinear arithmetic theories cannot cover the path constrains solving on ARM due to some special operators of ARM instructions (e.g. shift, rotate, count_leading_zeros). In these cases, fixed-size bitvector theory is an appropriate choice. Detailed specifications of operations can be found at SMT-LIB [3] or Z3-Guide [4]. In brief, it can be described by a set of operators as follows, where `a` and `b` are two BitVector values:

1. Basic Bitvector Arithmetic
   - Addition: `(bvadd a b)`
   - Subtraction: `(bvsub a b)`
   - Unary Minus: `(bvneg a)`
   - Multiplication: `(bvmul a b)`
   - Unsigned Remainder: `(bvurem a b)`
   - Signed Remainder: `(bvsrem a b)`
   - Signed Modulo: `(bvsmod a b)`
   - Shift Left: `(bvshl a b)`
   - Logical Shift Right: `(bvlshr a b)`
   - Arithmetical Shift Right: `(bvashr a b)`
2. Bitwise Operations
   - Bitwise OR: `(bvor a b)`
   - Bitwise AND: `(bvand a b)`
   - Bitwise NOT: `(bvnot a)`
   - Bitwise NAND: `(bvnand a b)`
   - Bitwise NOR: `(bvnor a b)`
   - Bitwise XNOR: `(bvxnor a b)`
3. Predicates over Bitvectors
   - Unsigned Less or Equal: `(bvule a b)`
   - Unsigned Less Than: `(bvult a b)`

[3]http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml
[4]https://rise4fun.com/z3/tutorialcontent/guide

- Unsigned Greater or Equal: `(bvuge a b)`
- Unsigned Greater Than: `(bvugt a b)`
- Signed Less or Equal: `(bvsle a b)`
- Signed Less Than: `(bvslt a b)`
- Signed Greater or Equal: `(bvsge a b)`
- Signed Greater Than: `(bvsgt a b)`

However, there are some operations of ARM instructions still cannot be covered by the basic operators in Fixed-Size Bitvector Theory. In these cases, we need to transform these operations to a combination of basic operations. For instance, the `mostSignificant(a)` can be transformed to `shiftRight(a,16)` where `shiftRight` is a basic operator supported by BitVector theory in SMT solvers and 16 is a half of length of a register (assume we are using 32-bit architecture). Then, the SMT format of this operator will be `(bvlshr a #b010000)`. By the same manner, `leastSignificant(a)` can be transformed to `shiftRight(shiftLeft(a,16),16)`.

In addition, some special cases even contains loops inside. For instance, the `CLZ r` instruction aims to count the number of leading zero of the value stored in a register *r*. The simplest implementation of `CLZ` requires a loop inside:

```
int clz(int r){
    int count = 0;
    while (r > 0) {
        r >> 1;
        count++;
    }
    return 32 - count;
}
```

Figure 4.4: The simplest implementation of CLZ

Because Theorem Prover cannot accept loops, we need to declare new function to unfold the loop inside `CLZ`. Fortunately, the number of loop is statically identified because we assuming the target architectures are 32-bit. Therefore, the loop now is unfolded by using 32 statements as follows:

```
int clz(int r){
    int count = 0;
    int new_r;
    new_r = r >> 1;
    if (new_r > 0) count++; else return new_r;
    ... < 30 times more >...
    new_r = r >> 1;
    if (new_r > 0) count++; else return new_r;
    return new_r;
}
```

Figure 4.5: The unfolded form of CLZ

After that, it can be easily written in SMT format to input to a theorem prover for checking the satisfiability of path constrains.

```
(declare-const r0 (_ BitVec 32))
(declare-const c0 (_ BitVec 32))
... same declarations for r1, c1 ... r31, c31
(declare-const r32 (_ BitVec 32))
(declare-const c32 (_ BitVec 32))
(declare-const z (_ BitVec 32))
(declare-const m (_ BitVec 32))
(define-fun clz ((x (_ BitVec 32))) (_ BitVec 32)
(if (and
  (= r0 x) (= z #x00000000) (= m #x00000001) (= c0 #x00000020)
  (= c1 (ite (bvsgt (bvashr r0 m) z) (bvsub c0 m) c0))
  (= r1 (ite (bvsgt (bvashr r0 m) z) (bvashr r0 m) r0))
  ... same declarations for c2, r2, ... , c31, r31
  (= c32 (ite (bvsgt (bvashr r31 m) z) (bvsub c31 m) c31))
  (= r32 (ite (bvsgt (bvashr r31 m) z) (bvashr r31 m) r31))
) c32 #x00000021))
```

Figure 4.6: The SMT format of CLZ

This figure above shows an example of unfolded methods `CLZ`. After defining it in SMT format, this function now can be used for further checking. In general, to deal with all unsupported methods, the corresponding SMT format of them are also declared.

4.3 Semantics Extraction Overview

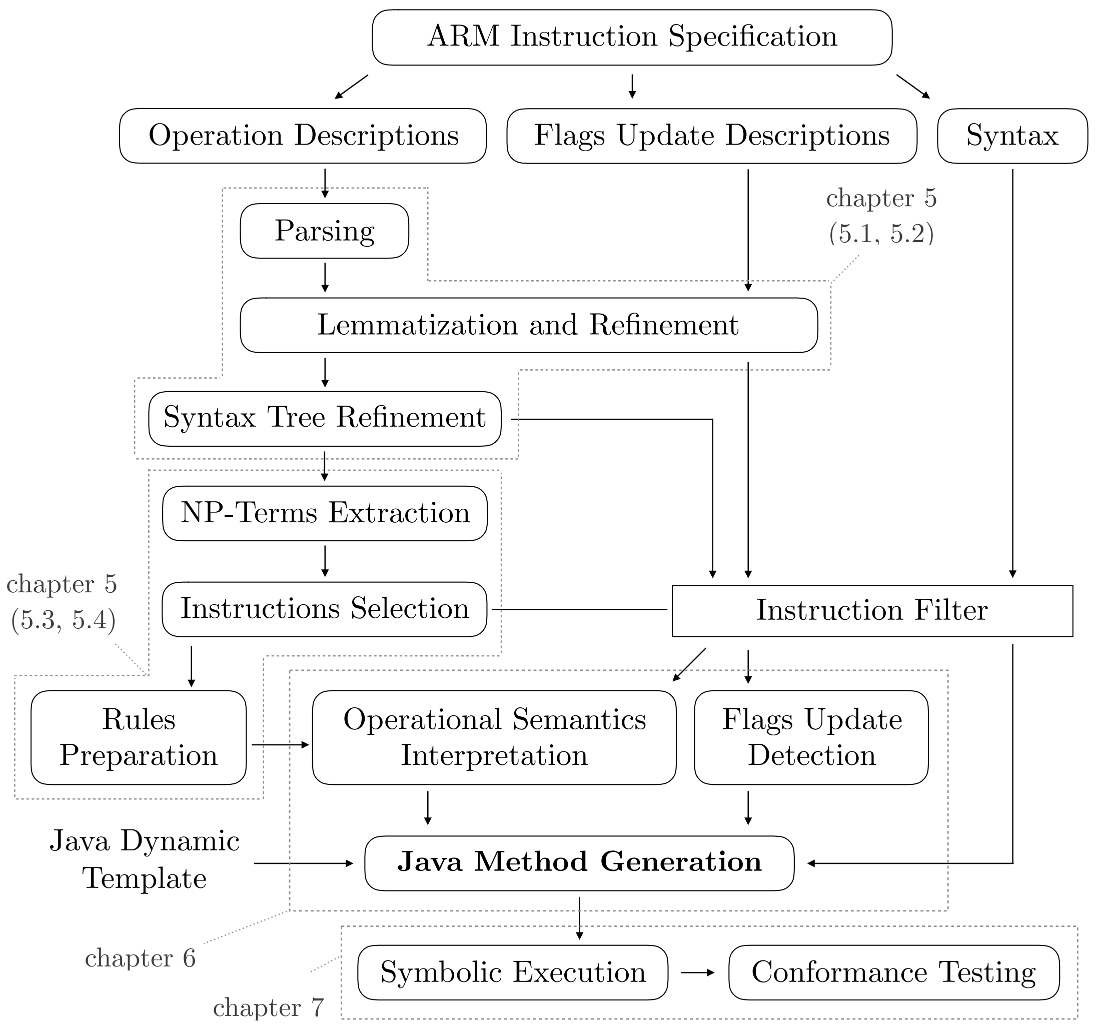


Figure 4.7: Semantics Extraction Overview

The figure above illustrates how semantics extraction is performed. It has two main parts: *Prerequisites* and *Dynamic Symbolic Executor Generation*. The first part aims to prepare prerequisites for the generation, which includes rewriting rules and selected instructions. In this part, some NLP techniques are applied. After that, the second part generates the dynamic symbolic executor for ARM, which consists of a matching process to get executable code, and the flags changes detection. In fact, the path conditions are also generated inside the matching process. The next two chapters present the detailed procedures of these two parts.

Chapter 5

Syntax Normalization and Semantics Interpretation

Before automatic performing the semantics extraction, we need to perform some prior prerequisites, which requires using natural language processing techniques. After these tasks, the normalized form of instruction is obtained and the rewriting rules is also prepared.

5.1 Instructions Normalization

Before pushing instructions to the generation step, it need to be normalized to extract syntax structure, important words, and noun phrases. Given an instruction S, the aim of this task is extract syntax tree and NP-Terms of S. Figure 5.1 shows the normalization process. First, the instruction's operation description is splitted to individual sentences. After being processed by three steps, all sentences is combined together. At here, each instruction contains all syntax tree and NP-Terms of sentences. To apply NLP on each sentence of the operation section in the specification, three steps are performed: the *parsing*, the *lemmatization*, and the *unimportant word removal.*

1. **Syntax Parsing:** This step performs parsing to get the syntax tree T of S.

2. **Syntax Tree Refinement:** In different situations, words in English is transformed into forms, but the meaning is kept. For instance, *"studied"* and *"studies"* carry the same meaning, but if we consider it as two separate words, it is not reasonable. Therefore, this step first perform *Word Lemmatization* to transforms words to its primitive form. After that, unimportant words are removed by using TF-IDF score.

3. **NP-Terms Extraction:** We call a noun phrase (NP) in the syntax tree is a *NP-Term.* This task inputs a syntax tree T and extract all *NP-Terms* from T.

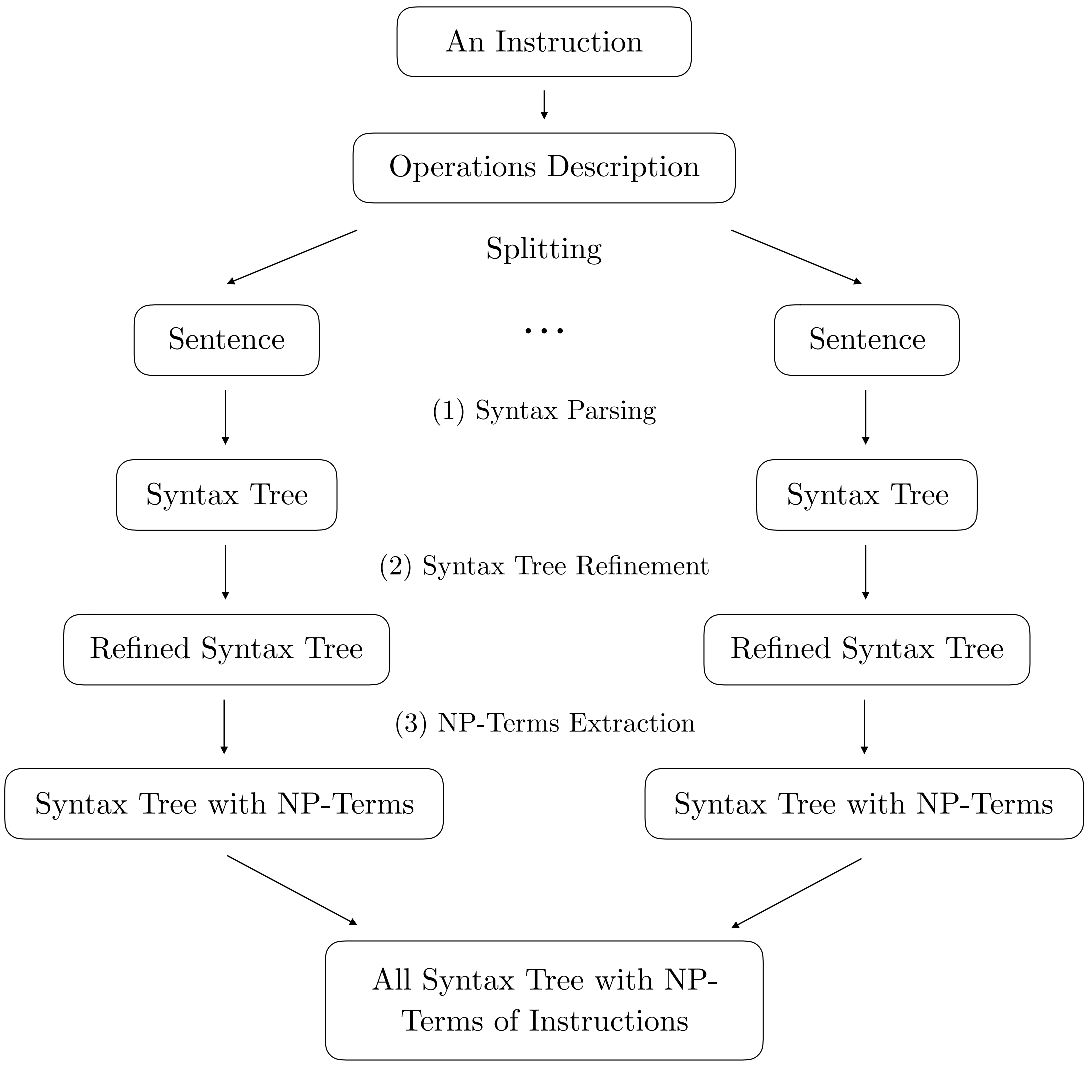


Figure 5.1: Instruction Normalization Procedure

Sentence Syntax Parsing

The parsing is the first step for the syntax analysis of a sentence. We use the standard tool *NLTK* [29], obeying to the built-in context free grammar of English. It outputs the syntax tree labeled with the classification of phrases, like **NP** (Noun Phrase). Fig. 5.2 shows the output of the sentence *"The UMAAL instruction multiplies the two unsigned 32-bit integers in the first and second operands"* in *UMAAL* specification. In fact, we can use our own grammar rules, but in this work, we use the default context-free grammar of *NLTK*, There are some grammar rules contained in this:

```
S  → NP VP
PP  → P NP | IN NP
```

```
NP → DT CD JJ JJ NNS | DT JJ CC JJ NNS
VP → VBZ NP | VP PP
DT → 'the'
NN → 'instruction'
VBZ → 'multiplies'
IN → 'in'
CD → 'two'
JJ → 'unsigned' | '32-bit' | 'first' | 'second'
...
```

Example 5.1.1. Based on the grammar rules above, the syntax tree of the sentence *S: "The UMAAL instruction multiplies the two unsigned 32-bit integers in the first and second operands"* can be generated:

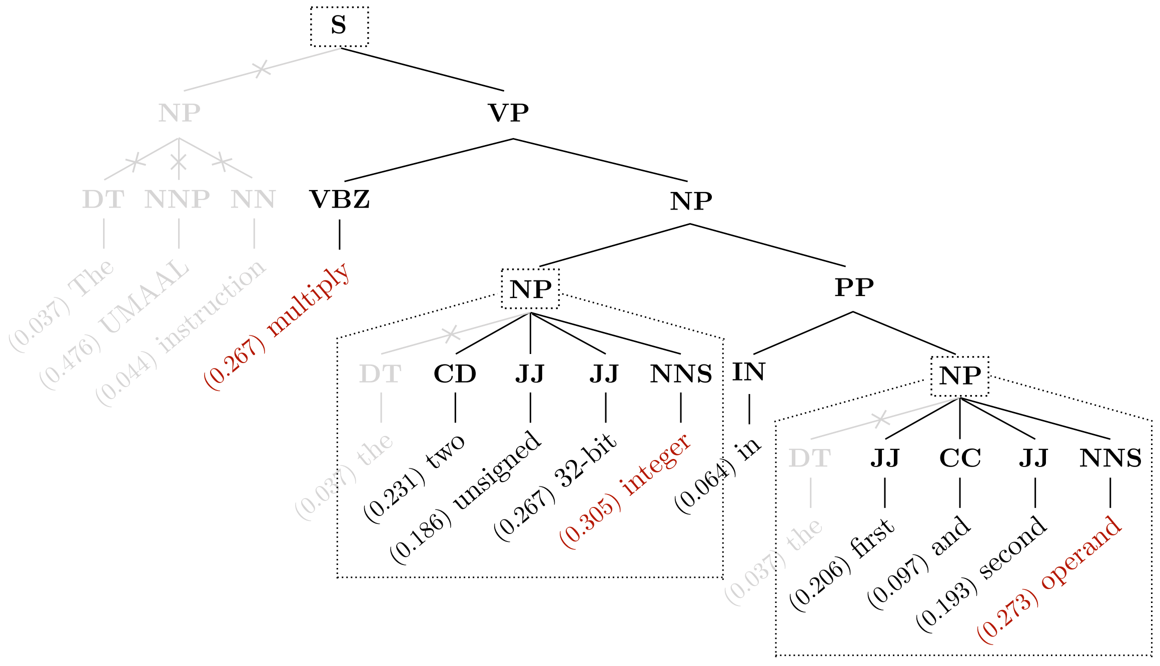


Figure 5.2: Example of the syntax tree, its refinement and TF/IDF scores

where:

Abbreviation	Meaning
S	Sentence
NP	Noun Phrase
VP	Verb Phrase
DT	Determiner
NNP	Proper Noun
NN	Singular Noun

Abbreviation	Meaning
PP	Prepositional Phrase
CD	Cardinal Number
JJ	Adjective
IN	Preposition
NNS	Plural Noun
CC	Coordinating Conjunction

Table 5.1: Syntax notations in the syntax tree of S

Lemmatization

A natural language like *English* introduces the variations on an expression, e.g., the conjugation, the synonym and the plural form. The lemmatization unifies them to the standardized form of a word. The tool *NLTK* also provides the lemmatization of English words. The red-colored words appearing at the leaves in Fig. 5.2 are the result of the lemmatization, e.g., *multiplies* $\rightarrow$ *multiply*, *integers* $\rightarrow$ *integer* and *operands* $\rightarrow$ *operand*.

Unimportant Words Removal

In this step, TF/IDF score is used to remove unimportant words, where a *term* is a word appearing in the operation section, and the whole documents are a set of the operation sections of all collected specifications. For instance, the TF/IDF scores of words in the sentence above is the values put together with leaves' label. After trying a few sentences, we set the threshold $h = 0.05$ to cut unimportant words. Note that, the name of the instruction is also removed. In Fig. 5.2, the grey edges are cut and if all the leaves are cut, the removal is propagated to the upper node.

NP-Terms Extraction

NP-Term and NP-Phrase

Definition 5.1.1. A *native NP-Term* is a subtree of the normalized syntax tree of which the root node is labeled "NP" and each non-root node is not labeled "NP". A *NP-Phrase* is either the phrase at the leaves of a native NP-Term or the (whole) sentence after replacing each native NP-Term with $\Box$.

After trials on about 100 sentences, we concentrate on subtrees with the root label *NP* (Noun Phrase). This seems the best compared with keeping or selecting the labels like *VP* (Verb Phrase) and *PP* (Propositional Phrase) for optimal choices of the NP-Phrase and the instructions. Due to the efficiency reason, we ignore intermediate subtrees with the root label NP in a refined syntax tree, and focus on the phrases of native NP-Terms and the whole sentences. Since the semantic interpretation is applied in a bottom-up manner on the refined syntax tree, the phrases of native NP-Terms are replaced with $\Box$, which means a hole to be instantiated. To identify the holes $\Box$'s, we enumerate them and give the indices.

Example 5.1.2. In Fig. 5.2, the subtrees surrounded by the dotted lines are native NP-Terms. The extracted NP-Phrases are: *"two unsigned 32-bit integer"*, *"first and second operand"*,and *"multiply* $\Box_1$ *in* $\Box_2$*"*.

This process inputs the refined syntax tree T, then outputs a list of *NP-Terms* L. To extract *NP-Terms* from T, we do the traversal over this tree by the bottom up manner. If a node N has the label *'NP'* or *'S'*, all its children nodes containing words or a blank hole $\Box$ are merged together and become a new *NP-Term.* This *NP-Term* is also added to L. After that, the subtree which N is the root is replaced by a blank hole $\Box$. Repeat this process until T becomes a blank hole $\Box$. Return L. In fact, we have try to use verb phrases (VP) or propositional phrase (PP) instead of noun phrase, but the result is worse. *NP-Terms* seems like the best choice.
Then, the following NP-Terms can be extracted from the refined syntax tree above.

1. first and second operand
2. two unsign 32-bit integer
3. multiply $\square_1$ in $\square_2$

After completing this step, all syntax tree with NP-Terms of S is generated. After that, its state is kept for further usage in semantics extraction without redoing the previous tasks.

5.2 Automatic Instructions Selection

As stated in 4.1.3, there are over 1000 instructions over six ARM Cortex series. Because some operations are written by a long and complex natural language description, a huge number of rules are required to cover it. However, these rules are usually rare, some are unique. For instance, among all instructions of six series, these instructions STLEX, VLLDM, and LDAEX in Cortex-M33 only appear one time and its description is very long and complex. In comparison with the gained benefit, the efforts needed to cover these cases is not worth it. Hence, it is necessary to have an effective strategy to decide which instructions should be processed, in order to save the human efforts as much as possible. This strategy inputs all instructions of six architectures, and an acceptance rate α then output a list of best selected instructions.

After having all extracted syntax tree and NP-Terms, we perform a selection strategy to decide which instruction should be processed.

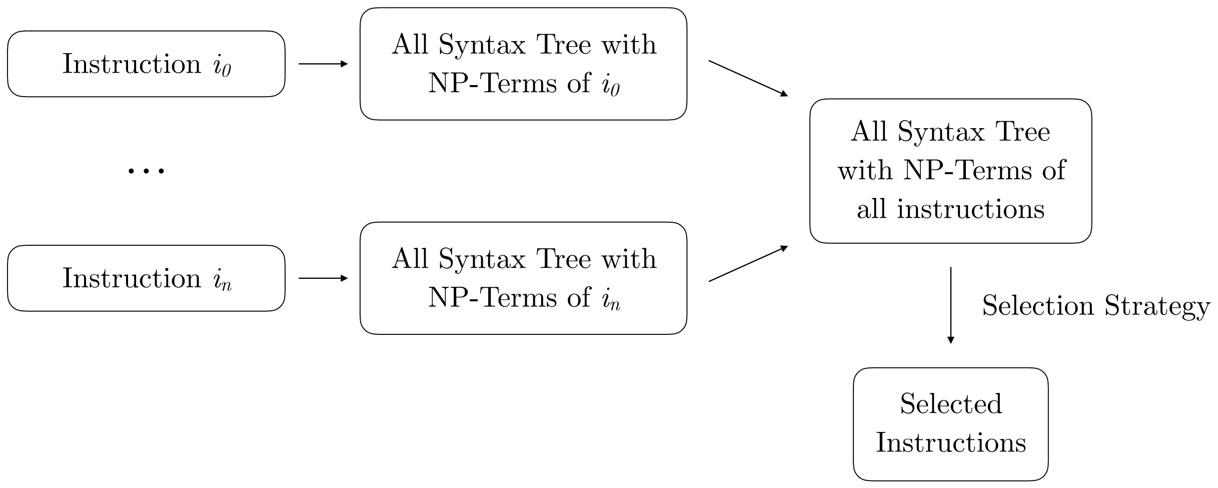

Instruction i_0
All Syntax Tree with NP-Terms of i_0
...
Instruction i_n
All Syntax Tree with NP-Terms of i_n
All Syntax Tree with NP-Terms of all instructions
Selection Strategy
Selected Instructions

Figure 5.3: Candidates selection from all instructions

We observe on the collected specifications of ARM instructions that:

- The same instruction may be shared with different variations, but sometimes they have different semantics. For instance, the instruction UASX appears both in M33 and M4 but has different flag updates.

- Some instructions appear only once among variations and have long and complex natural language descriptions. For instance, STLEX, VLLDM, and LDAEX in Cortex-M33 are such examples.

Our aim is that less effort of manually prepared semantics interpretation rules covers the semantics of more instructions, trying to have an optimal trade-off. Thus, the strategy for selecting NP-Phrase and instructions is:

1. First, estimate the TF/IDF score for all NP-Phrases in all sentences of the operation descriptions over n collected instructions.

2. Second, the importance of an instruction i is considered as the sum of TF/IDF scores of all NP-Phrases contained in i. We aim to select k instructions that maximize the sum of their importance scores divided by k. We set an acceptance rate α to be 65%, means $k \geq \alpha \times n$.

Definition 5.2.1. Instruction Selection Strategy: For a list of k chosen candidates, we use $\varphi(k)$ to measure the efficiency of the selection strategy over all instructions in six variations. The greater $\varphi(k)$, the better selected candidates. Let I is the set of all n collected instructions:

$$I = \{i_1, i_2, \ldots, i_n\}$$

where an instruction i consists of a set T_i including w NP-Terms:

$$T_i = \{\langle t_1, f_1\rangle, \langle t_2, f_2\rangle, \ldots, \langle t_w, f_w\rangle\}$$

where t_j is the j^{th} NP-Term, and f_j is the frequency of t_j in i. Let $p(t_j)$ is the proportional occurrence of t_j over all NP-Terms in I, the importance of i over I is defined as:

$$m_i = \sum_{j=1}^{w} p(t_j).f_j$$

Let M is the descending sorted set of all m:

$$M = sorted(m_1, m_2, \ldots, m_{n-1}, m_n)$$

Let M_q is the q^{th} value of M, k is the number of expected candidates, $\varphi(k)$ is then defined as:

$$\varphi(k) = \frac{1}{k}\sum_{q=1}^{k} M_q$$

Now, given a particular value of k, this strategy can effectively obtain an optimal trade-off by just taking the first k candidates to make $\varphi(k)$ as large as possible.

After performing the selection procedure, the number of selected instructions and selected NP-Phrase are 692 and 228, respectively. Among them, we manually observe the further need of the syntax normalization, e.g., $\langle$*sign bottom* | *bottom sign*$\rangle \rightarrow$ *sign bottom*, and 23 syntax normalization rewrite rules are added. The resting 205 NP-Phrases are regarded as a set of the left-hand sides candidates for semantic interpretation rules, which is manually prepared.

5.3 Rewriting Rules Preparation

Based on our investigation over ARM instructions, we observe that, some noun phrases in its operations' description occurs many times. For instance, as the table below, for over six ARM Cortex architectures (M0, M0+, M3, M4, M7, and M33), the term *"destination register"* and *"second operand"* appear 249 times and 156 times in the instructions description, respectively. Thus, it is reasonable to first, define some rewriting rules one time, then use it for many times. The most popular *Noun Phrases* over six architectures are described in the table below. Note that, the smaller instructions covered by one rules, the better performance of reducing human efforts.

Phrases	**Occurrences**	**Phrases**	**Occurrences**
destination-register	249	zero-extend-to-32-bits	90
second-operand	156	word-value	64
first-operand	129	offset-from-register-rn	64
memory-address	124	lowest-memory-address	62
top-halfword	94	highest-memory-address	62

Table 5.2: Most popular extracted NP-Terms

Therefore, by our observation, it is feasible to define some rewriting rules for the popular noun phrases one time, then use it many times. This figure below shows how rewriting rules are generated in our work. This process inputs a sentence S from operation description, then outputs some rewriting rules for this sentence.

Rewriting System

After selecting appropriate instructions to optimize the human efforts, we have already had the normalized syntax tree and extracted NP-Terms of them. These NP-Terms act as the left-hand side (LHS) of rewriting rules. We will now design the right-hand side (RHS) for each extracted NP-Terms. This task aims to prepared all rewriting rules for the semantics interpretation, which will be introduced later.

Definition 5.3.1. Our Rewriting System: Let a, b, u, v are strings. Our designed system is a reduction system over strings. A rewriting rules $r : a \to b$ is a transition from a LHS u to a RHS v. With a, b are any strings (including empty string), r can be extended to:

$$aub \to avb$$

Rules Categories.

Consider each string (e.g, uaw, ubw) is a *token*, each token is either an English sequence or abbreviation (e.g, *neg*). Let t, t_1, t_2 are tokens, bvs is a BitVec-class code statement. There are two types of rewriting rules:

1. Token Transformation:

$$t_1 \to t_2$$

2. Interpretations:

$$t \rightarrow bvs$$

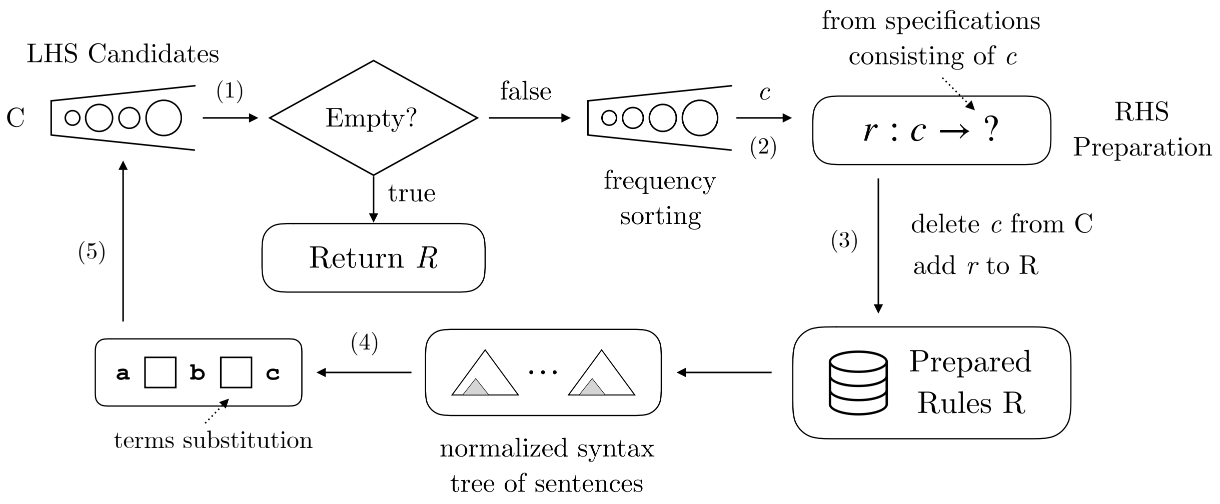
Each left-hand side of a rule can be categorized into 3 patterns below. A ground rule rewrites NP-Phrases of native NP-Terms, a left-linear rule rewrites NP-Phrases of the whole sentence, and a conditional rule unifies several rules that are quite similar.

- *Ground rules:*
  - Interpretation form: $a \rightarrow b$
  - Example: *first and second operand* $\rightarrow$ *rn, rm*
- *Conditional rules:*
  - Interpretation form: $a \langle\ b \mid c\ \rangle\ d \langle\ e \mid f\ \rangle \rightarrow g$
  - Example: $\langle$ *most significant* | *top* $\rangle$ $\langle$ *32-bits* | *32-bits of result* $\rangle$ $\rightarrow$ *shift(result, Mode.RIGHT, 32)*
- *Context-based rules:* $\Box_i$ is a hole to be instantiated by a string.
  - Interpretation form: $a \Box_1\ b \Box_2\ c \rightarrow d \Box_2\ e \Box_1\ f$
  - Example: *extract bits* [$\Box_1$:$\Box_2$] $\rightarrow$ *result = extract(xm,*$\Box_1$,$\Box_2$*)*

Note that, a rewrite rule can be both conditional and left-linear. For instance, $\langle$ *place* | *write* $\rangle$ $\langle$$\Box_1$*(result,32)* | $\Box_1$*(result,32) of result*$\rangle$ $\langle$ *in* | *to* $\rangle$ *rdhi* $\rightarrow$ *write(xhi,* $\Box_1$*(result, 32))*. Among 205 NP-Phrases for the semantics interpretation, the number of ground, conditional ground, left-linear, and conditional left-linear rules are 116, 69, 14 and 6, respectively.

Rules Preparation Process

Starting with the set of selected NP-Phrases as the initial set of the left-hand sides of the semantics interpretation rules, the right-hand sides of rules now are manually prepared. This figure illustrates this preparation procedure:



Choose an NP-Phrase with the highest frequency, its corresponding right-hand side is added by directly interpreting the specifications that contain this NP-Phrase. The specifications may include NP-Phrases that have not selected. They are newly added to the set of the left-hand sides of the rules.

(1) C is the set of the left-hand sides of rules that need to be interpreted.

(2) $c \in C$ with the highest frequency is completed as a rule $r : c \to u$.

(3) r is added to the rule set R, and c is deleted from C.

(4) Rules in R rewrite each in C. When a substitution to $\Box$ in $r \in R$ occurs, the rule r is updated with this substitution.

(5) Continue until $C = \varnothing$.

Example 5.3.1. Consider the procedure on a sentence in UMAAL specification. Assume a set of NP-Phrase: $C = \{c_1, c_2, c_3\}$ with c_1 : *"first and second operand"*, c_2 : *"two unsigned 32-bit integer"*, and c_3 : *"multiply* $\Box$ *in* $\Box$*"*. The highest frequency NP-Phrase is on c_2 and by interpret the specifications, we set r_2 : *first and second operand* $\to$ *rn, rm*). c_3 now is substituted such that c_3' = *multiply* $\Box$ *in rn,rm.* After the procedure completed, R consists of:

r_1**:** first and second operand $\to$ rn, rm

r_2**:** two unsigned 32-bit integer $\to$ val($\Box$), val($\Box$)

r_3**:** multiply val($\Box$), val($\Box$) in rn, rm $\to$ mul(val(rn), val(rm))

Chapter 6

Dynamic Symbolic Executor Generation

6.1 Generation Overview

As our observation, `$name` and `$params` can be easily extracted from the Mnemonic. We classify the executable code to two parts: *main-operation* and *flags-update*. The *main-operation* is the part that only takes effect on the register, memory, and stack of the environment ($(\langle R, M, S\rangle)$) while *flags-update* part takes only effect on the flags of environment ($(\langle F\rangle)$). The dynamic symbolic executor generation now focuses on three following problems:

1. **Operation Code Generation:** This part aims to generate the main operation code of instruction (*main-operation* part), then fill it to the blank `$execCode` in the dynamic code template. This generated code usually performs arithmetic or bitwise operations then update the binary emulator components such as memory, stack, and register.

2. **Flags Update Detection:** This part aims to detect whether a flag is changed or not (*flags-update* part). After that, it fills an array of modified flags to the blank `$flags`. For instance, `$flags = "N","Z","C"` means this instruction updates the N, Z, and C flags. The V, GE, and Q flags are left unmodified.

3. **Path Conditions Generation:** The CFG generation uses concolic testing to check the satisfiability of the path conditions from the initial node to the current node for deciding which execution branch is feasible. Therefore, along with executable code generation and flags change detection, path conditions also need to be generated.

By using the prepared template and rewriting rules, code generation process can be briefly described as follows:

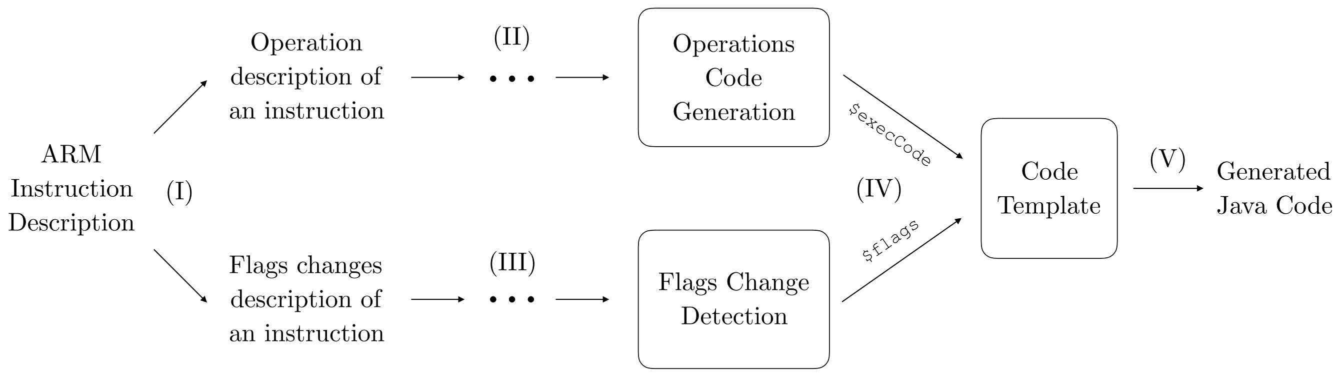


Figure 6.1: Java code generation overview for an instruction

Assume the ARM instruction need to be proceed is K.

1. First, at (I), the operation description and flags change description are extracted from the description of K.
2. Second, at (II), the executable code generation part is performed (detailed process is presented later). After this step, the `$execCode` is obtained. The detailed procedure of (II) is given in the section 6.2
3. Third, at (III), the flags changes detection part is performed (detailed process is presented later). After this step, the `flags` is obtained. The detailed procedure of (III) is given in the section 6.3
4. Next, at (IV), the obtained `$execCode` and `$flags` are filled to the code template which is prepared.
5. Finally, the full Java code for K is generated.

The next three sections present the detailed procedures of Executable Code Generation, Flags Change Detection, and Path Conditions Generation.

6.2 Operations Code Generation

Code generation from Operations Description inputs an instruction written in natural language describing an operation of an ARM instruction, then generate a corresponding formal Java code. For instance, if the sentence *"The UMAAL instruction multiplies the two unsigned 32-bit integers in the first and second operands"* is considered as the input, the expected output of this process is: `mul(val(rn),val(rm))`.

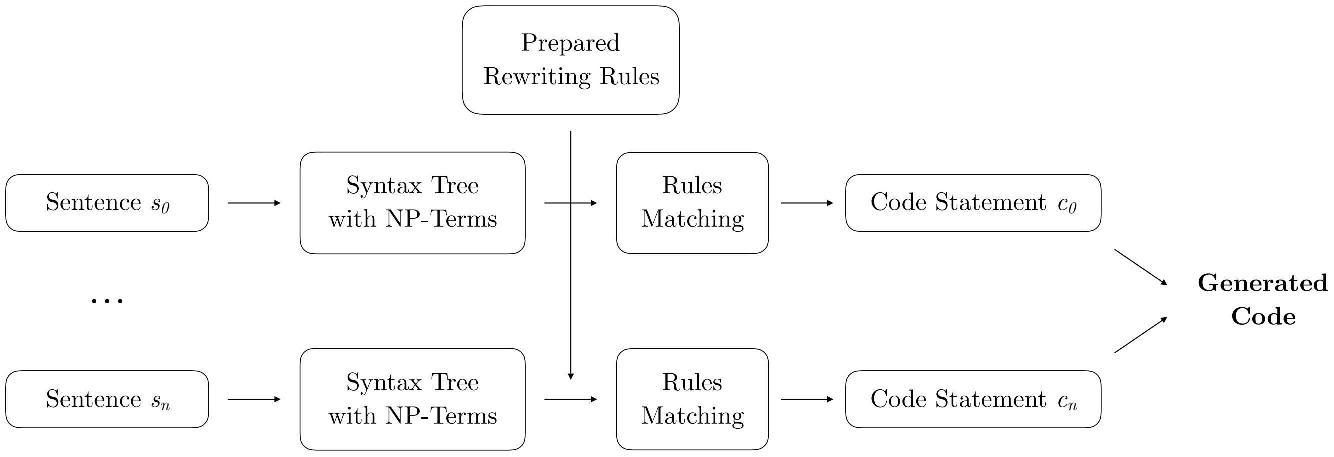


Figure 6.2: Code generation process for an instruction

The diagram above illustrates how the Java code is generated.

1. First, each sentence s_i of this instruction with its syntax tree and NP-Terms (already extracted) is performed bottom-up rules matching with the prepared rewriting rules. After that, a code statement c_i is generated.

2. After having all c_i, we combine all together to get the final generated code.

Rules Matching

This task aims to generate executable code from the refined syntax tree above, with the helps of prepared rewriting rules. The manner is, start matching nodes from the bottom to the top of the tree, if the term in this node is matched with the *LHS* of prepared rules, replace it by the *RHS* of the rules. Keep going until the root of the tree is reached.

Example 6.2.1. Let assume we have already had the following prepared rewriting rules:

1. first and second operand → rn, rm

2. two unsign 32-bit integer → val(□), val(□)

3. multiply val(□), val(□) in rn, rm → mul(val(rn), val(rm))

These rules may be occasionally already prepared. In such a case, the reductions are performed in a bottom-up manner as described in the figure above:

$$\begin{aligned}
& \text{multiply two unsigned 32-bit integer in } \underline{\text{first and second operand}} \\
\overset{r_1}{\to}\ & \text{multiply } \underline{\text{two unsigned 32-bit integer}} \text{ in rn, rm} \\
\overset{r_2}{\to}\ & \underline{\text{multiply val(□), val(□) in rn, rm}} \\
\overset{r_3}{\to}\ & \text{mul(val(rn), val(rm))}
\end{aligned}$$

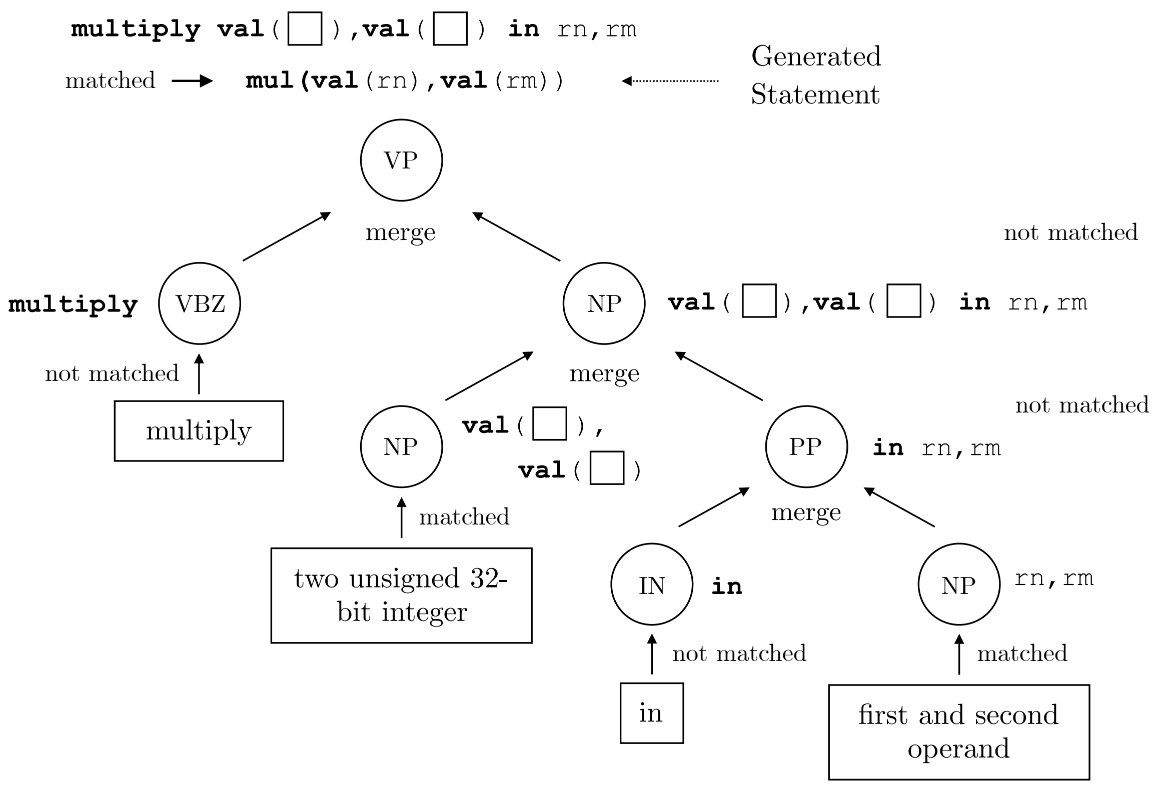


Figure 6.3: Matching process to generate formal Java code for an operation

6.3 Flags Update Detection

The table below shows typical expressions for the flags update descriptions appearing in the specifications.

Description	**Implication**
This instruction does not change the flags	Does not update flags
This instruction does not affect the condition code flags	
The V flag is left unmodified.	
This instruction updates the N, Z, C and V flags according to the result.	Update specific flags based on description
Updates the N and Z flags according to the result. Does not affect the C and V flags.	

To detect whether a flag is updated, a topic modeling method called Latent Dirichlet Allocation (LDA) [17] is used to estimate the topics distribution of a sentence s and the (unique) model sentence: m = *"update affect set change modify"* as two real numbers vectors $\overrightarrow{v_s}$, $\overrightarrow{v_m}$, respectively. After that, the similarity between s and m are evaluate by a similarity measure over $\overrightarrow{v_s}$ and $\overrightarrow{v_m}$. The flags "modified" if $sim(\overrightarrow{v_s}, \overrightarrow{v_m})$ does not exceed a threshold t; otherwise, "unmodified". As the preprocessing step of the sentences in the flags update section, we apply the same lemmatization and the unimportant word removal in Section 5.1. The unsupervised model training uses all the preprocessed sentences in the flags update section of all collected ARM instruction specifications. The figure below illustrates the procedure:

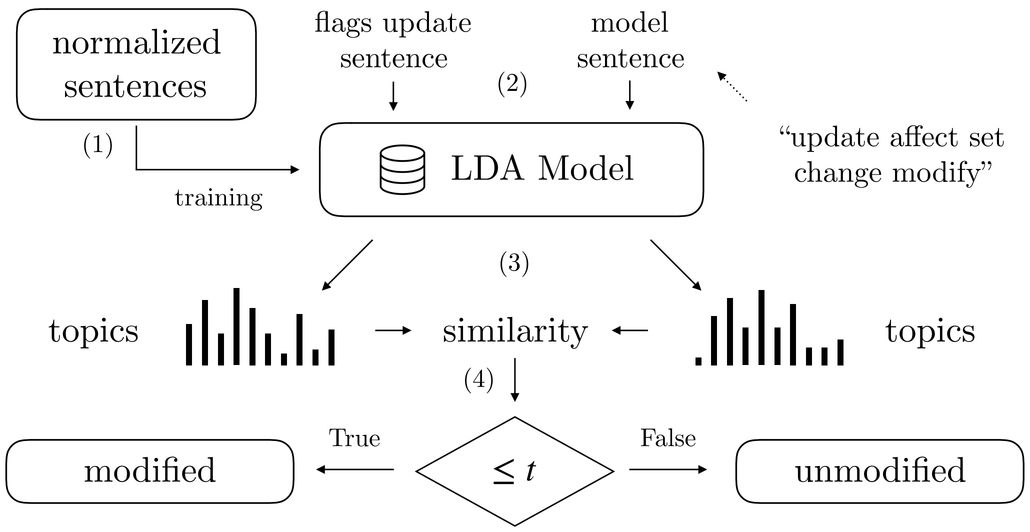


Figure 6.4: Flags Update Detection

We use the implementation of LDA in sklearn library [30] with a set of hyperparameters: $\alpha = 0.1, \beta = 0.1, ntopics = 10, twords = 10, niters = 2000$ and the Cosine similarity as the similarity function. The threshold is set $t = 0.85$.

Example 6.3.1. Suppose that two sentences s_1 and s_2 need to be classified into two classes "modified" and "unmodified". Assume the model sentence is m; v_1, v_2, and v_m are vector representations by LDA model of s_1, s_2, and m respectively. We have:

- s_1*:* "This instruction updates the N, Z, C and V flags according to the result". In our experiment, v_1*=(0.81998589, 0.02000604, 0.02, 0.02000347, 0.02, 0.02000072, 0.02000053, 0.02000209, 0.02000125, 0.02)*
- s_2*:* "Does not affect the C or V flags". In our experiment, v_2*=(0.05, 0.05, 0.05, 0.05, 0.05, 0.55, 0.05, 0.05, 0.05, 0.05)*
- *m:* "update affect set change modify". In our experiment, v_m*=(0.02000251, 0.2199939, 0.02, 0.2199923, 0.02, 0.41999946, 0.02000674, 0.02000319, 0.02000191, 0.02)*

By applying the Cosine similarity, we have $sim(s_1, m) \approx 0.0834$ and $sim(s_2, m) \approx 0.868$. Because $sim(s_1, m) \leq 0.85$ and $sim(s_2,) > 0.85$, s_1 and s_2 should be classified to *"modified"* and *"unmodified"* respectively.

6.4 Path Conditions Generation

Symbolic Execution executes program and generates the pre-condition and post-condition as *path condition* [5]. A path condition is a conjunction of constrains in jump instructions from the entry point to the current instruction. The path conditions need to be generated because we need to check its satisfiability in concolic testing to decide which path is feasible.

Path Conditions

The Symbolic Execution store the symbolic state of a program execution by a set of $\langle n_i, a_i, \varphi_i \rangle$ where n_i is a node and φ_i is the precondition of the path reaching to n_i from the initial node (the program entry). By checking by a theorem prover (SMT Solver), if φ_i is *satisfiable*, the path n_i is *feasible*.

Definition 6.4.1. In CFG, each node i is a tuple of its location n_i, current assembly instruction a_i, and the path conditions φ_i from the initial node to i:

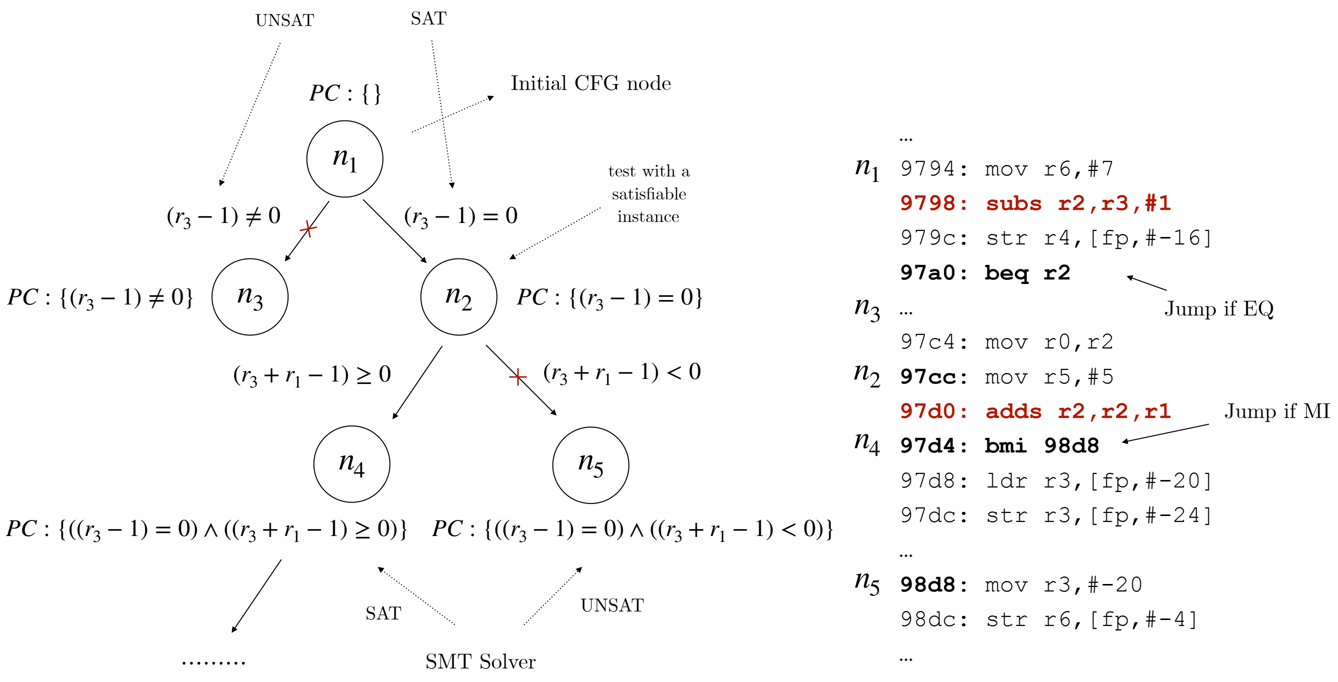
- $\langle n_0, a_0, \varphi_0 \rangle$ is the initial node (entry point) of the CFG
- $\langle n_i, a_i, \varphi_i \rangle$ is the i^{th} node

The path conditions is updated through each step of the execution:

$$\begin{cases} \varphi_0 = true \\ \\ \varphi_{i'} = \varphi_i \wedge c_i \end{cases} \tag{6.1}$$

where i is the previous instruction of i' and c_i is the path condition from node i to i'. Please note that the condition c_i might depend on the environment's variables. For example, the instruction `beq` checks the condition `Z = true`, where `Z` is the flag Z. However, the flags Z might be changed based on the data instruction (e.g, `subs r0 r1`). In that case, we also need to update the condition of Z: $z = ((r0 - r1) = 0)$. This procedure will continue until the program execution interrupts (at the end of program or when meeting an unsupported instructions). In our implementation, the path conditions is updated inside the manually defined methods.

Example 6.4.1. The figure below shows an example to illustrates how path conditions are generated through the execution on ARM.



```
...
9794: mov r6,#7
9798: subs r2,r3,#1
979c: str r4,[fp,#-16]
97a0: beq r2
...
97c4: mov r0,r2
97cc: mov r5,#5
97d0: adds r2,r2,r1
97d4: bmi 98d8
97d8: ldr r3,[fp,#-20]
97dc: str r3,[fp,#-24]
...
98d8: mov r3,#-20
98dc: str r6,[fp,#-4]
...
```

Figure 6.5: Path conditions update through the execution on ARM

The CFG in the left-hand side represents the execution of the code in the right-hand side. Through the execution, the path conditions and CFG is generated as follows:

1. n_1: Assume n_1 is the initial CFG node. First, this node is added to CFG without any conditional checking. After that, next instruction at `9798` is executed. At that time, the value of r_2 is updated: $r_2 = r_3 - 1$. Because this is `subs`, the flags is updated based on the result of this instruction. It means, $Z = (r_3 - 1) < 0$ (in example, we only consider flag Z because the next jump instruction only needs Z). Next, the instruction `beq` checks the condition $Z = true$ (because of the suffix `eq`. It means, the path constrain here will be $(r_3 - 1) < 0$. Now, theorem prove was used to decide which branch is satisfiable. Let's assume that right branch is SAT and left branch is UNSAT. Because `beq r2` is an indirect jump, the jump target must be decided by testing an satisfiable instance of PC. Let's assume here the theorem prover returns $r_2 = 97cc$, then the execution now continues to node n_2 and update path conditions at n_2: $PC = \{(r_3) - 1 = 0\}$.

2. n_2: At n_2, the next instruction $adds r_2, r_2, r_1$ is executed. Therefore, the value of r_2 is updated: $r_2 = r_2 + r_1 = r_3 - 1 + r_1$. Because this is `adds`, the flags are updated based on the result of this operation. It means, $N = (r_3 + r_1 - 1) < 0$. Then, the right-hand side branch conditions is now assigned to $(r_3 + r_1 - 1) < 0$ and the left-hand side branch conditions is now assigned to $(r_3 + r_1 - 1) \geq 0$. The next instruction is `97d4`, checking the condition $N = true$ (suffix `mi`) means checking the new path constrain: $((r_3 - 1) = 0) \wedge ((r_3 + r_1 - 1) < 0)$. Assume that it's UNSAT, then now the execution follows the left-hand side and test it with a satisfiable instance of the path condition: $((r_3 - 1) = 0) \wedge ((r_3 + r_1 - 1) \geq 0)$.

3. The execution will interrupted at the end of program or catching an unsupported instruction.

After running through all instructions in the binary file, the Dynamic Symbolic Execution returns the CFG of binary files, which reflects its behaviors. From its behaviors, further techniques may be applied to detect whether this binary file is a malware or not. However, if at a node, the SMT Solver returns *unknown* or the instruction is not already supported, this process is interrupted.

Chapter 7

Conformance Testing

Software testing is the procedure to check the correctness of a program by comparing the similarity of actual result and expected output. Test cases generation is one of software testing problems, which aims to automatically generate all possible program scenarios without the involvement of human. To verify the correctness of the generated Java methods, a conformance testing is performed by comparing the execution results between the generated Java method and the binary emulator $\mu Vision$ [13] (which supports many ARM variations). Since the number of instructions over six ARM architectures is huge, manual testing is very time-consuming and requires a lot of human efforts in both writing test cases and perform the test. This chapter presents our testing process to (1) automatically generate test-cases, then (2) proceed the automated tests by combining the change of binary emulator's environments between our implementation and a trusted ARM Debugger (We use ($\mu Vision$) [13]). For automatically generate test cases, in this work, we used Symbolic Execution technique. After that, a theorem prover was used (Z3) [23] to check the satisfiability of all possible conditional paths. If satisfiability of a path is SAT, we use this concrete values as the input parameters of the test case. To generate test data for full coverage, we apply a symbolic execution tools based on Java Pathfinder (JPF) named JDart [31] on generated Java methods.

7.1 Automatic Tests Generation

As stated in 3.3.1, symbolic execution can be used to explore all feasible path of program. Therefore, for each feasible path, a test case is generated. By testing over all test cases, the correctness of the program can be warranted. We will illustrate how Symbolic Execution works in this example to generate test cases.

Example 7.1.1. Assume the following function *foo* is needed to test, the aim is to generate test-cases to cover all feasible path of *foo*.

```
public boolean foo(int a, int b, int c){
   int x = 5;
   int y = 1;
   if (a == b){
      x = x - 1;
      y = y + x;
   } else {
      if (c == a + b){
         x = x + y;
         if (a * b == 36) {
            y = y - x;
         }
      } else {
         x = x - y;
      }
   }
   return x > y;
}
```

Figure 7.1: An example of function need to be tested

Solve: Let α, β, γ are the symbolic variables representing three *foo*'s parameters a, b, c respectively. The execution tree of *foo* is generated as below:

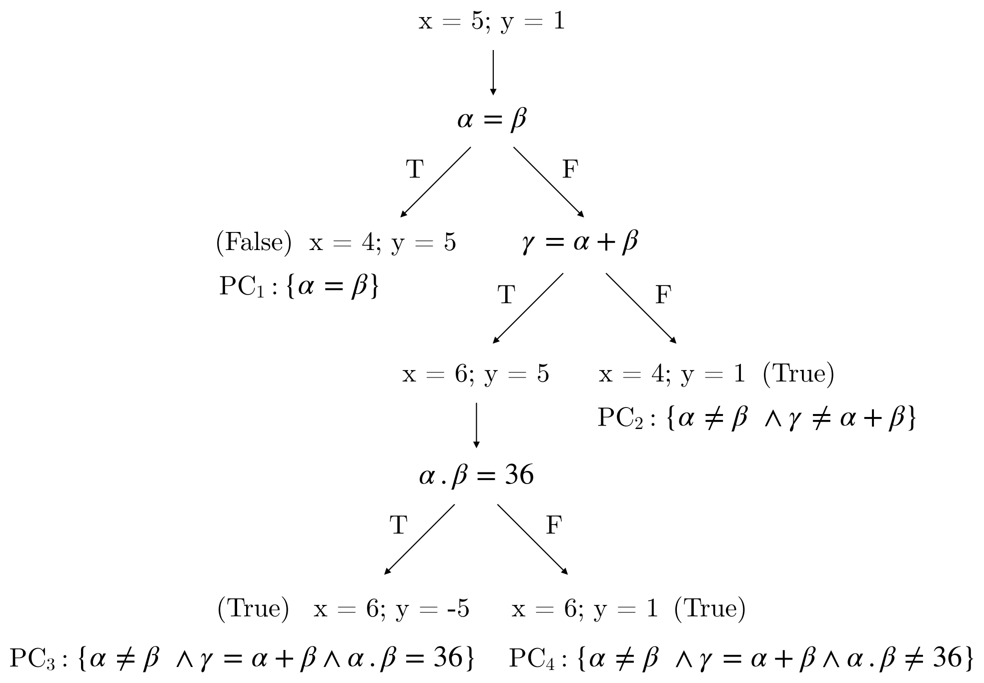

x = 5; y = 1
$\alpha = \beta$
T
F
(False) x = 4; y = 5
$PC_1 : \{\alpha = \beta\}$
$\gamma = \alpha + \beta$
T
F
x = 6; y = 5
x = 4; y = 1 (True)
$PC_2 : \{\alpha \neq \beta \wedge \gamma \neq \alpha + \beta\}$
$\alpha . \beta = 36$
T
F
(True) x = 6; y = -5
x = 6; y = 1 (True)
$PC_3 : \{\alpha \neq \beta \wedge \gamma = \alpha + \beta \wedge \alpha . \beta = 36\}$
$PC_4 : \{\alpha \neq \beta \wedge \gamma = \alpha + \beta \wedge \alpha . \beta \neq 36\}$

Figure 7.2: Execution tree of *foo*

where:

- T, F: branch conditions
- $(True)$, $(False)$: results of *foo* function.
- PC_i: path conditions from the initial node.

Symbolic Execution generate 4 possible path conditions. Using a theorem prover (e.g., $Z3$), 4 pairs of input values and expected result of test-cases are:

1. For PC_1: $\langle(\alpha = 0, \beta = 0, \gamma = 0), false\rangle$
2. For PC_2: $\langle(\alpha = 0, \beta = 1, \gamma = 2), true\rangle$
3. For PC_3: $\langle(\alpha = 1, \beta = 36, \gamma = 37), true\rangle$
4. For PC_4: $\langle(\alpha = 60, \beta = -34, \gamma = 26), true\rangle$ $\square$

7.2 Test Case Structure

To evaluate the correctness of our generated code, our idea is comparing the environments of binary emulator between our implementation and $\mu Vision$ ARM debugger. Note that, the environment includes the states of *flags* (F), *registers* (R), *memory* (M), and *stack* (S). Hence, A test-case consists of:

1. Input variables:
    - *Java Source Code (JSC):* The target generated Java executable code.
    - *Parameters (PRS):* A set of parameters with concrete values generated by symbolic execution.
    - *Pre-Environment (PrEnv):* The environment of binary emulator before execution.
2. Expected output:
    - *Post-Environment (PoEnv):* The environment of binary emulator after execution.

7.3 Testing Procedure

Assume all possible code for instructions over 6 architectures are generated and pushed into a stack S. This diagram illustrates how the testing process is performed:

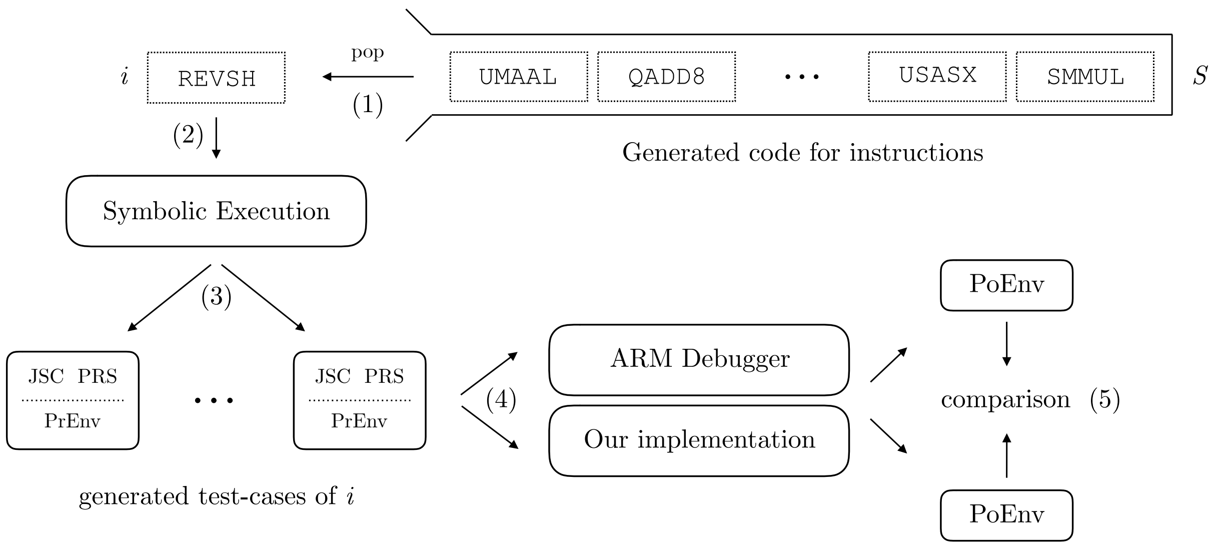


Figure 7.3: Conformance Testing Procedure

1. First, at (1), an instance i is pop from the stack S, then push through the Symbolic Execution at (2).
2. After that, at (3), this code is analyzed by symbolic execution, then all possible test-cases of i are automatically generated. Note that, each test-case consists of four components. For each test-case:
   - At (4), we push it through $\mu Vision$ and our implementation. This step outputs two *PoEnv* state of $\mu Vision$ and our implementation.
   - Next, at (5), we compare the two *PoEnv* above. If the two *PoEnv* are similar, we can conclude that, with this test-case, the implemented function works correctly.
3. Finally, if there is no wrong test-case in i, it can be conclude that, this function is implemented correctly.

Chapter 8

Experiments

This section presents the experiment result of our proposed method over six ARM Cortex series (M0, M0+, M3, M4, M7, and M33). First, the efficiency of the Instruction Selection Strategy is evaluated. After that, we conduct experiments on how effective the rewriting rules cover instructions and evaluate the generated Java code by using test cases generated from the conformance testing procedure. Finally, we discuss some ignored cases and failed cases to clarify why it cannot be covered by our current method.

8.1 Instruction Selection Strategy

To evaluate the efficiency of the automatic instruction selection strategy, we conduct the experiment over 1039 ARM instructions collected from ARM Developer Website, over six architecture. The black line R describes the average number of rules that need to be prepared per an instruction. The blue line C shows the percentage of covered instructions.

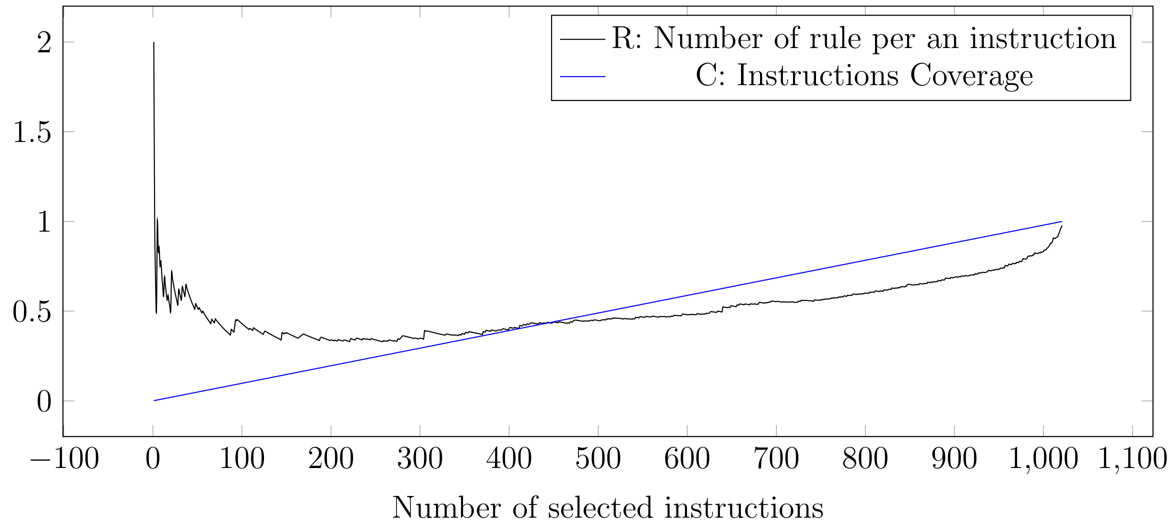


Figure 8.1: Instruction Selection Strategy Performance

The figure illustrates that, for the very first selected instructions, the value of R decreased dramatically. After taking 258 instructions, R reaches the smallest value. Although at that point, the value of R is minimum, but the number of covered instructions is quite small (24.83%). After that, the value of R tends to increase until it reach the value 0.9785 at the end. This result can be interpreted that, the automatic Instruction Selection Strategy works well because it chooses the best candidates, then make the R decrease. After that, the remaining candidates is not good enough. As the result, the value of R increases.

Note that, the acceptance rate C must be provided at the beginning. In our implementation, we choose the rate $C = 65\%$, it means our strategy tries to selects N best instructions to minimize the value of R as long as $N \geq 65\%$ total number of instructions. With the $C = 0.65$, for average, we only need to write 0.34 rules to successfully generate an instruction implementation. At this point, 692 instructions are generated (66.60%) and 662 (63.72%) instructions are verified.

8.2 Successfully Generated Instructions

This table shows the number of selected and generated instructions where:

- **Variation:** The ARM Cortex-M Architecture.
- **Collected:** Total instructions collected from ARM Developer Website.
- **Selected:** Total instruction selected by our Selection Strategy.
- **Generated:** Number of generated instructions.
- **Verified:** Number of instruction passed the conformance testing.

Variation	Collected	Selected	Generated	Verified
Cortex-M0	63	44 (69.84%)	44 (69.84%)	41 (65.08%)
Cortex-M0+	63	44 (69.84%)	44 (69.84%)	41 (65.08%)
Cortex-M3	129	80 (62.02%)	80 (62.02%)	74 (57.36%)
Cortex-M4	244	167 (68.44%)	167 (68.44%)	161 (65.98%)
Cortex-M7	261	178 (68.20%)	178 (68.20%)	172 (65.90%)
Cortex-M33	279	179 (64.16%)	179 (64.16%)	173 (62.00%)
Total	1039	692 (66.60%)	692 (66.60%)	662 (63.72%)

Table 8.1: The number of generated instructions over six Cortex series

The table above shows the detailed experiment result. After applying the proposed method for 1039 collected ARM instructions of 6 variations, the Java methods for 692 instructions (66.60%) are generated by using 228 rewrite rules. Among 692 generated Java methods, 662 methods (63.72%) have passed an automated conformance testing. It can be argued that, the instruction selected by our Instruction Selection Strategy is quite balanced among six architectures. The number of generated instructions equals to the number of selected instructions because rules are prepared for all selected instruction by our strategy. However, the number of verified instructions is obviously smaller than generated instructions because there are some failed cases. We will discuss in the next sections.

8.3 Discussion

Ignored Cases

We observe that some instructions are described by very long and complicated sentences. Theoretically, it is possible to analyze this case, but it will take a huge effort to define rules for them. Therefore, these cases should be ignored. In fact, our Instruction Selection Strategy removed them automatically.

Mnemonic	QSAX
Brief description	Saturating Subtract and Add With Exchange
Syntax	QSAX{cond} {Rd}, Rm, Rn
Opreration	Subtracts the bottom halfword of the second operand from the top highword of the first operand. Adds the bottom halfword of the source operand with the top halfword of the second operand. Saturates the results of the sum and writes a 16-bit signed integer in the range $-2^{15} \leq x \leq 2^{15} - 1$, where x equals 16, to the bottom halfword of the destination register. Saturates the result of the subtraction and writes a 16-bit signed integer in the range $-2^{15} \leq x \leq 2^{15} - 1$, where x equals 16, to the top halfword of the destination register.
Flags update	This instruction does not affect the condition code flags.

Table 8.2: The ignored case of QSAX instruction in Cortex-M7

Failed Cases

There are 30 cases that failed to pass the conformance testing. Failures are classified into two reasons.

1. **Wrong flags change detection:** For flags change detection task, there are total 1436 individual sentences needed to analyze, include various types of description. It also uses many synonyms such as *update, change, affect, modify, and set.* Our experiment correctly detect 1428 sentences among this set. The cases that we missed are the complex synonyms like *"left unmodified"* in the sentence *"The V flag is left unmodified"* in the instruction `RORS` of Cortex-M0+. When the complex synonyms like *"left unmodified"* in the flag update section of the instruction RORS (Cortex-M0 and Cortex-M0+), the similarity analysis fails.

Mnemonic	RORS
Brief description	Rotate Right
Syntax	RORS {Rd,} Rm, Rs
Opreration	RORS moves the bits in the register Rm to the right by the number of places specified by register Rs.
Flags update	This instruction updates the N and Z flags according to the result. The C flag is updated to the last bit shifted out, except when the shift length is 0. The V flag is left unmodified.

Table 8.3: The failed case of RORS instruction in Cortex-M0+

2. **Wrong order of sentences in instruction description:** Some instruction descriptions do not have a correct order of sentences. It is hard to detect and reverse its order. Therefore, currently our method cannot deal with this problem. For instance, the instruction *STRB* descriptions has two sentences, but its order is not correct. The correct one should be "The STRB instruction zero extend a register unsigned value then store it to memory". Our method interprets each sentence in order in the operation section. Thus, the confusing order of the sentences leads failure. For instance, the operation section of the instruction STRB (Cortex-M7) intends *"The STRB instruction* zero extend *a register unsigned value, then store it to memory"*, but with the opposite order.

Mnemonic	STRB
Brief description	Store Register for two unsigned byte
Syntax	STRB{cond} Rt, Rt2, [Rn],#offset
Operation	The STRB instruction store a register unsigned byte value to memory. Zero extend to 32 bits on loads.
Flags update	This instruction does not change the flags.

Table 8.4: The failed case of STRB instruction in Cortex-M7

8.4 Running Example of the Generated DSE Tool

We show a simple case demonstrating how our Dynamic Symbolic Execution (DSE) tool explores the destination of an indirect jump in ARM Cortex-M. In this example, the indirect jump appears in the instruction **`bmi r1`**, where **`mi`** is the conditional suffix and **`r1`** is the destination of indirect jump. Note that, this DSE uses the semantics of ARM instructions extracted by our proposed method.

```
-----[ INIT ]-----
+ Flags:
+ Register (32-bit):
-----[ ADDS R0 #15 ]-----
+ Flags:
  - N:0 (bvslt (bvadd r0 #x0000000f) #x00000000)
  - Z:0 (= (bvadd r0 #x0000000f) #x00000000)
  - C:0 (bvugt (bvadd r0 #x0000000f) #xFFFFFFFF)
+ Register (32-bit):
  - R0  :(bvadd r0 #x0000000f)
-----[ ADDS R1 #38850 ]-----
+ Flags:
  - N:0 (bvslt (bvadd r1 #x000097c2) #x00000000)
  - Z:0 (= (bvadd r1 #x000097c2) #x00000000)
  - C:0 (bvugt (bvadd r1 #x000097c2) #xFFFFFFFF)
+ Register (32-bit):
  - R0  :(bvadd r0 #x0000000f)
  - R1  :(bvadd r1 #x000097c2)
-----[ ADDS R3 R0 #12 ]-----
+ Flags:
  - N:0 (bvslt (bvadd (bvadd r0 #x0000000f) #x0000000c) #x00000000)
  - Z:0 (= (bvadd (bvadd r0 #x0000000f) #x0000000c) #x00000000)
  - C:0 (bvugt (bvadd (bvadd r0 #x0000000f) #x0000000c) #xFFFFFFFF)
+ Register (32-bit):
  - R0  :(bvadd r0 #x0000000f)
  - R1  :(bvadd r1 #x000097c2)
  - R3  :(bvadd (bvadd r0 #x0000000f) #x0000000c)
-----[ SUBS R4 R3 R1 ]-----
+ Flags:
  - N:1 (bvslt (bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2)) #
      x00000000)
  - Z:0 (= (bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2)) #x00000000)
  - C:0 (bvugt (bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2)) #
      xFFFFFFFF)
+ Register (32-bit):
  - R0  :(bvadd r0 #x0000000f)
  - R1  :(bvadd r1 #x000097c2)
  - R3  :(bvadd (bvadd r0 #x0000000f) #x0000000c)
  - R4  :(bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2))
-----[ NEG R5 R3 ]-----
+ Flags:
  - N:1 (bvslt (bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2)) #
      x00000000)
  - Z:0 (= (bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2)) #x00000000)
  - C:0 (bvugt (bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2)) #
      xFFFFFFFF)
+ Register (32-bit):
  - R0  :(bvadd r0 #x0000000f)
  - R1  :(bvadd r1 #x000097c2)
  - R3  :(bvadd (bvadd r0 #x0000000f) #x0000000c)
  - R4  :(bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (bvadd r1 #x000097c2))
  - R5  :(bvneg (bvadd (bvadd r0 #x0000000f) #x0000000c))
-----[ BMI R1 ]-----
-> Indirect jump detected !
-> Checking path constrains by z3 (bvslt (bvsub (bvadd (bvadd r0 #x0000000f) #x0000000c) (
    bvadd r1 #x000097c2)) #x00000000) ...
   SAT instance: sat
   (model
   (define-fun r0 () (_ BitVec 32)
    #x000097a6)
   (define-fun r1 () (_ BitVec 32)
    #x00000000)
   )
-> Jump to #x000097c2
```

Figure 8.2: An example demonstrating how our DSE generates path conditions through executions and explores the destination of an indirect jump

Chapter 9

Conclusion and Future Directions

9.1 Result and Conclusion

This thesis proposed a systematic approach to extract formal semantics from natural language specifications of ARM Cortex-M instructions (from 6 variations, M0, M0+, M3, M4, M7, and M33). In the experiment, the semantics of 692 instructions is extracted from 1039 collected specifications from ARM Developer Website, after preparing 228 rewrite rules for the semantics interpretation and 35 initial methods used in the Java template. Among them, 662 (63.72%) have passed the automated conformance testing. The extracted semantics of each instruction is presented as a Java method of an extension of BitSet class. With the surrounding libraries, these Java methods give the implementation of the symbolic execution and the binary emulator, and we obtain the preliminary version of the dynamic symbolic execution of ARM Cortex-M for free. We confirmed that it correctly traces indirect jumps in small examples. We expect that our methodology can be a standard approach to implement the dynamic symbolic execution of binaries on other platforms, such as other Cortex of ARM and MIPS. Our ultimate goal is to reduce the human efforts on the implementation of Dynamic Symbolic Executor for ARM, which is used to analyze malware under obfuscation techniques. This study has contributed three main modules:

1. *Semantics Interpretation:* We have already developed a module to generate Java executable code for operations of ARM instructions in multiple platforms. The number of instruction now can be covered is approximately 63.72%. Because the remaining instructions consist of very long, complex and not popular instructions, at this moment, they are temporarily ignored.

2. *Flags Update Detection:* We have presented a method to automatically detect which flag is updated after execution by using an unsupervised machine learning algorithm called Latent Dirichlet Allocation. This algorithm works well for short and sparse sentences, which typically occur in flags changes descriptions of ARM instructions.

3. *Path conditions generation:* Beside generating the operation code, the path conditions are also generated. It is used in the satisfiabilities checking of the dynamic symbolic executor for ARM.

4. *A set of interpretation rules:* We also provide a set of rewriting rules for ARM Cortex-M series. To extend this method to another platform, these rules must be prepared manually.

Our result shows that: With only 228 prepared rules, 692 instructions are successfully generated and 662 instructions are verified. The rule/instruction ratio is approximately 0.34, it means for average, we need 0.34 rules to cover an instruction. Since rewriting rules is quite short and easy to prepare, our method reduces a lot of human effort on the implementation of Dynamic Symbolic Executor for ARM Cortex series over 6 architectures (M0, M0+, M3, M4, M7, and M33). The total generated Java code is approximately 10800 lines.

Advantages:

1. *Our method is a semi-automatic approach:* Our method only requires preparing rewriting rules at the beginning (for the operations generation task), and a model sentence (for flags change detection task). After that, everything runs automatically.

2. *Our method is effective:* It can cover a large number of instructions with a small ratio of predefined rewriting rules per an instruction.

3. *Our method is generalized:* It can be extended to multiple platforms as long as the rewriting rules are prepared in advanced.

Drawbacks:

1. *Long and complex instruction:* Our method still cannot deal with the long and complex instructions like QSAX. If this kind of instruction is added, the average efforts needed to prepare all rules is much larger, but the number of covered instructions is not much bigger.

2. *Requires human effort on preparing rules:* Because our method still requires the prepared rewriting rules, it still needs human efforts.

9.2 Future Directions

Up to now, out implementation can analyze and extract formal semantics for six ARM Cortex series including *Cortex-M0*, *Cortex-M0+*, *Cortex-M3*, *Cortex-M4*, *Cortex-M7*, and *Cortex-M33*. Even the total number of instruction in six mentioned architectures is quite big (over 1000), it is still a small set in all available instructions that needs to be processed. In the future, we intend to continue this study to enlarge the capacity of our method, make it be able to cover more architectures, as well as to deal with long and complex instructions. The first two next directions are:

1. *Extend to more series:* As our observation, ARM developer website does not contain structured document for other series like Cortex-A and Cortex-R. Only PDF specification can be found on this website. However, extracting structured data from a PDF file is a complex process and requires a huge effort because it contains tons of natural plain text. In the future, we are going to consider it as a pre-processing task. Because our proposed method is generalized for multiple platforms, after doing the pre-processing for PDF file, we can continue extracting formal semantics process in the same manner.

2. *Extend to other architecture:* Currently, our method supports only ARM architecture. in the future, it is feasible to extend our methodology to more architecture like MISP with a few modifications.

In the future, our plan is:

- Complete the dynamic symbolic execution tool and try experiments on real-world IoT malware.
- Try the methodology of the systematic semantics extraction on other platforms.
- ARM Developer Website does not provide structured documents for Cortex-A and R (only PDF files are provided). After performing the preprocessing for PDF files, we hope to apply our methodology for them in the same manner.

Bibliography

[1] I. You and K. Yim, “Malware obfuscation techniques: A brief survey,” in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on.* IEEE, 2010, pp. 297–300.

[2] F. Song and T. Touili, “Pushdown model checking for malware detection,” *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 2, pp. 147–173, 2014.

[3] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, “Proactive detection of computer worms using model checking,” *IEEE transactions on dependable and secure computing*, vol. 7, no. 4, pp. 424–438, 2010.

[4] J. Kinder, S. Katzenbeisser, C. Schallhart, Veith, and Helmut, “Detecting malicious code by model checking,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2005, pp. 174–187.

[5] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] F. Desclaux, “Miasm: Framework de reverse engineering,” *Actes du SSTIC. SSTIC*, 2012.

[7] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps, “Directed proof generation for machine code,” in *International Conference on Computer Aided Verification.* Springer, 2010, pp. 288–305.

[8] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, “Codisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2015, pp. 745–756.

[9] N. M. Hai, M. Ogawa, and Q. T. Tho, “Obfuscation code localization based on cfg generation of malware,” in *International Symposium on Foundations and Practice of Security.* Springer, 2015, pp. 229–247.

[10] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.” in *OSDI*, vol. 8, 2008, pp. 209–224.

[11] “Arm developer website, https://developer.arm.com/.”

[12] Y. Nguyen L. H., “Automatic extraction of x86 formal semantics from its natural language description,” Master’s thesis, School of Information Science, March 2018.

[13] "μvision, http://keil.com/mdk5/uvision/."

[14] "Ida pro, https://www.hex-rays.com/products/ida/."

[15] "Capstone engine, http://www.capstone-engine.org/."

[16] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[17] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[18] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of iot malware based on image recognition," *arXiv preprint arXiv:1802.03714*, 2018.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[20] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole exe," *arXiv preprint arXiv:1710.09435*, 2017.

[21] A. V. Phan, M. Le Nguyen, Y. L. H. Nguyen, and L. T. Bui, "Dgcnn: A convolutional neural network over large-scale labeled graphs," *Neural Networks*, 2018.

[22] J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 2009, pp. 214–228.

[23] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2008, pp. 337–340.

[24] G. Bonfante, J.-Y. Marion, and D. Reynaud, "A computability perspective on self-modifying programs," in *7th IEEE International Conference on Software Engineering and Formal Methods-SEFM 2009*, 2009.

[25] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in *International Conference on Computer Aided Verification.* Springer, 2011, pp. 165–170.

[26] F. Desclaux, "Miasm: Framework de reverse engineering," *Actes du SSTIC. SSTIC*, 2012.

[27] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *International Conference on Computer Aided Verification.* Springer, 2008, pp. 423–427.

[28] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification.* Springer, 2011, pp. 463–469.

[29] S. Bird and E. Loper, "Nltk: the natural language toolkit," in *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions.* Association for Computational Linguistics, 2004, p. 31.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, “Scikit-learn: Machine learning in python,” *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[31] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman, “Jdart: A dynamic symbolic analysis framework,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2016, pp. 442–459.